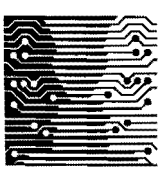
Facilitating the Modelling and Automated Analysis of Cryptographic Protocols

A DISSERTATION SUBMITTED TO THE DEPARTMENT OF
COMPUTER SCIENCE FOR THE DEGREE OF MASTER OF SCIENCE
AT THE UNIVERSITY OF CAPE TOWN

July 2001

Written by
Elton Saul

Supervised by
Dr Andrew Hutchison



Data Network Architectures Laboratory
Department of Computer Science
University of Cape Town
South Africa

Electronic versions of this thesis, published conference and journal papers as well as the SPEAR II application can all be downloaded from the SPEAR II website at http://www.cs.uct.ac.za/Research/DNA/SPEAR2. The author, Elton Saul, can be e-mailed at esaul@cs.uct.ac.za, while his supervisor, Dr Andrew Hutchison, can be contacted at hutch@cs.uct.ac.za.

Dedicated to my Father, who has always been
there to guide me in every situation
I have found myself.

Abstract

Multi-dimensional security protocol engineering is effective for creating cryptographic protocols since it encompasses a variety of design, analysis and deployment techniques, thereby providing a higher level of confidence than individual approaches. SPEAR II, the **S**ecurity **P**rotocol **E**ngineering and **A**nalysis **R**esource II, is a protocol engineering tool built on the foundation of previous experience garnered during the SPEAR I project in 1997. The goal of the SPEAR II tool is to facilitate cryptographic protocol engineering and aid users in distilling the critical issues during an engineering session by presenting them with an appropriate level of detail and guiding them as much as possible. The SPEAR II tool currently consists of four components that have been created as part of this dissertation and integrated into one consistent and unified graphical interface: a protocol specification environment (GYPSIE), a GNY statement construction interface (Visual GNY), a Prolog-based GNY analysis engine (GYNGER) and a message rounds calculator.

The GYPSIE environment is specifically geared towards the rapid, effective and accurate construction of cryptographic protocols and functions as the core enabling interface of the SPEAR II application. By utilizing three levels of abstraction presented through different views, the GYPSIE environment is able to present a protocol engineer with an appropriate impression of a cryptographic protocol and its operation. The *High-Level View* describes the overall flow of messages, using a formalism based on MSCs and SDL to represent the message passing specification, while the *Navigator View* appears adjacent to the High-Level View and summarizes the contents and structure of a protocol. The *Component View* is invoked from the High-Level View and allows a user to view and edit the contents of protocol messages, each message being displayed as a hierarchical tree. Among other features, GYPSIE includes support for subprotocols, features extensive undo and redo capabilities, is able to export a specification to text, LaTeX and Prolog, and includes an API set that allows for the incorporation of further engineering modules within the SPEAR II application.

GYNGER is a Prolog-based analyzer that performs automated analysis of protocols by using the GNY modal logic. The analysis engine improves on previous automation efforts and employs a forward-chaining approach to mechanize the tedious application of GNY inference rules, allowing all derivable GNY statements to be generated quickly, accurately and efficiently. To conduct an analysis with GYNGER a protocol engineer needs to specify a protocol's messages, initial assumptions and target goals in a Prolog-style GNY syntax. The GNY rule set is then imported and employed in the analysis, after which a proof is generated in an *English-style* GNY syntax for every successful goal that was specified. This English-style proof lists all of the statements involved in the derivation of the successful goal, indicating the postulates that were used and the premises which were employed in the postulate's application. Some advantages of GYNGER include the fact that it implements seventy-two of the GNY inference rules, incorporates syntax to represent shared secrets being used as identifiers, supports the 'never-originated-here' binary operator and features a flexible syntax for representing functions.

The Visual GNY environment was created to facilitate GNY-based protocol analysis and works in close conjunction with GYPSIE. In essence, Visual GNY functions as a user-friendly interface to the GYNGER analyzer. GNY statements necessary for an analysis are constructed using the Visual GNY interface and then passed on to GYNGER. Results from GYNGER are then returned to the Visual GNY environment and displayed appropriately. The novel aspect of the Visual GNY environment is that it represents GNY statements using a tree-like approach. All statements of the same type form part of the same tree structure, a heterogeneous set of GNY statements being represented by a collection of separate trees. This representation technique combined with pop-up menus which aid in statement construction, helps users to easily create GNY statements which are always syntactically correct. To use the Visual GNY environment, users do not need to be acquainted with the GNY syntax and notation, however, they must be familiar with the semantics and concepts underlying the logic to use it effectively. Some advantages of the Visual GNY environment include the ability to export structured trees to LaTeX, text and Prolog and view them as English-style text, suggested GNY statement completion and grouping of related sets of GNY statements by principal.

The message rounds calculator receives a message passing specification from GYPSIE and then returns the messages that can be sent together in parallel. This information helps to ensure that the most efficient protocol design in terms of message rounds can be deployed at the implementation stage, since the number of rounds resulting from the protocol model can be compared to the optimal number required for the protocol class into which the specification falls. The SPEAR I tool carried out limited synchronous message rounds calculations. Synchronous rounds calculations assume that a principal can only transmit a given message once he has received all of the previous messages in the specification which were destined for him. Optimal rounds calculations assume that message transmission can take place asynchronously. SPEAR II implements both *synchronous* and *optimal* message rounds calculations and is thus able to assist protocol engineers to a greater degree in bridging the gap from design to actual implementation.

All of the tools incorporated within the SPEAR II application have been tested to ensure that they fulfil their objectives. In experiments conducted with the GYPSIE environment, it was found that 75% of the protocol specifications created by users were completely error free, and that each of the individual specifications took approximately ten minutes to construct. The specifications used in these experiments contained four messages and twenty-eight components on average. When using the Visual GNY environment to create GNY statements, users had a 98% accuracy rate and all of the created statements were syntactically correct. GYNGER and the rounds calculator were both tested on various protocols and found to work as expected. In total, fifteen published protocols have been analyzed with the SPEAR II application. These analyses include well-known protocols such as the Yahalom, Wide-Mouth Frog, Needham-Schroeder Public-Key, Otway-Rees, Kerberos and Andrew Secure RPC Handshake Protocols. In essence, this dissertation has resulted in the creation of a graphically-based application that implements a multi-dimensional framework incorporating cryptographic protocol modelling, automated GNY-based analysis and message rounds performance analysis. The focus has been to remove as much of the complexity and tedium surrounding protocol engineering as possible and to provide a user-friendly, effective and powerful environment that can be used to facilitate the creation of secure cryptographic protocols.

Acknowledgements

This work would not have been possible, or nearly as much fun, without the support and assistance of a number of individuals and organizations:

- My supervisor, Dr Andrew Hutchison, for his expert supervision, guidance, motivation, patience and support throughout the past two years.
- The head of the DNA Research Group, Prof. Pieter Kritzinger, for his generosity and support of this work, associated travels and conferences.
- My mother, father and brother for their constant support, culinary supplies, encouragement and love, especially in the face of late nights and irregular working hours.
- Tracey, for your support, encouragement, friendship and dancing lessons lavished on me throughout the past two years. You made the time spent on this thesis pass very quickly indeed!
- Colin, Rita and Simon for your friendship and standing out among the rest in your support and genuineness. I'm glad to count you as my good friends.
- The friends from my cell group for their commitment and steadfastness. Caragh, Cristan, Kelly and Shaun, I really appreciate it!
- All of the Masters students for being such a great bunch of guys to work with and providing an environment conducive to quality and creativity. I'd especially like to thank the DNA boys, Mike, Ian, Lourens, Farrel, Andy, Yakomba and Thomas, for their friendship, criticism and fresh ideas.
- All of the anonymous referees and organizers associated with the ISMSSS 1999, SATNAC 1999, SEC 2000, SATNAC 2000, SACSIT 2000, WISE 2 and ISMSSS 2001 conferences and workshops for affording me the opportunity to present portions of this work to my peers.
- All of the willing and unwilling forty or so SPEAR II beta testers who helped to stress the system and put it through its paces.
- The Computer Science system administrators, Sam and Matthew, for constantly keeping our machines and network up and running. Let's also not forget the hard-working secretaries, Eve, Mary, Bernie and Zubaida for shielding me from paperwork and administrative activities.
- The National Research Foundation and Center of Excellence for sponsoring this work.

To all of you, I would like to express my sincere thanks and appreciation! It has been a pleasure working with every one of you.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

"It is hard to simulate the behaviour of the devil; one can always check that a protocol does not commit the old familiar sins, but every so often someone comes up with a new and pernicious twist."

— Ross Anderson and Roger Needham, "Programming Satan's Computer"

The use of open and unreliable computer networks is rapidly increasing as more companies and individuals connect to local and global networks. Internet applications such as the World Wide Web, instant messaging and e-mail have been rapidly adopted in the past few years to facilitate business processes and personal communication. At the time of writing, Internet research firm Jupiter Media Metrix[1] estimated that about 18.4 million people in the United States use Microsoft's instant messenging service, about 25.5 million people use AOL's free service, while Yahoo! Messenger has 11.8 million users. Although such growth is generally viewed as a beneficial trend, the associated security issues have often been ignored or simply lagged behind the expansion of network usage. In particular, the advancement and confidence in key technologies such as electronic banking, secure communication and electronic commerce has often been hindered by security problems associated with open networks. As a result of the development and construction currently taking place in the networking field, it is an opportune time to develop tools to support in the creation, analysis and maintenance of secure communication infrastructures.

Nua Surveys[2] estimates that the number of internet users in November 2000 numbered approximately 407.1 million, and more specifically within selected regions: 3.11 million in Africa, 113.14 million in Europe and 167.12 million in Canada and the United States. It has been estimated that business-to-consumer e-commerce will generate 52, 76 and 108 billion US dollars in the years 2001, 2002 and 2003 respectively. Business-to-business e-commerce will generate 499, 843 and 1331 billion US dollars over the same period. Within the next decade, the vast majority of companies will be forced onto the Internet by competitors who are reducing costs and expanding their customer base by exploiting this market. Market sectors such as financial services, computer hardware and software, travel, books, music and flowers are experiencing rapid growth in online sales [58]. For example, in January 1999, Dell Computer Corporation was selling an average of 14 million US dollars of equipment online per day [7], while Amazon.com has become the third largest bookseller in the United States, despite only being in business since 1995 [7]. Such enormous growth, scope and finance create a huge imperative for secure communication between companies and their customers, especially in light of the fact that security concerns have forced consumers to be cautious of using the Internet for electronic commerce [82].

[1] http://www.jmm.com

[2] http://www.nua.ie/surveys

Cryptographic protocols are used to provide security during communication sessions. Since messages that are sent across open networks are vulnerable to interception and manipulation by unknown entities, the security of these networked systems is crucial to protect the interests of all of its users. One of the most important characteristics of a networked system is the distributed nature of the communicating entities, also known as principals. A protocol is a set of rules that is used to define an exchange of messages between two or more of these principals. In particular, cryptographic protocols make use of security techniques to achieve goals such as confidentiality, authentication, integrity and non-repudiation. However, the fact that strong cryptographic algorithms exist does not guarantee the security of a communications system [75]. In fact, it is widely recognized that the engineering of security protocols is a very challenging task, since protocols which appear secure can contain subtle flaws and vulnerabilities that attackers can exploit [2]. Some examples of well-known protocols that have been found to be insecure include Microsoft's PPTP protocol [62], an early version of Netscape's SSL Protocol [2], the CCITT X.509 Protocol [1] and the Needham-Schroder Public Key Protocol [55]. Essentially, what is needed are tools and techniques that can effectively aid designers in creating, verifying and implementing solid, secure and reliable cryptographic protocols.

A number of proven and workable techniques currently exist to aid in the protocol engineering process [31, 56]. *Inference-construction methods* utilize modal logics similar to those which have been developed to monitor the evolution of belief and knowledge in distributed systems. Some popular modal logics are BAN [1], GNY [33] and SVO [79]. *Attack-construction methods* construct probable attack sets based on the algebraic properties of the algorithms employed in a protocol. These methods have to examine large state spaces to isolate possible attacks. Discussions and implementations of this approach can be found in [57, 24, 29, 46, 52]. *Proof-construction methods* attempt to avoid the exponential searches of attack construction methods by formally modelling the actual computations performed in protocols and then proving theorems about these computations. Information related to this approach can be found in [76, 66, 65, 9]. Besides the development of formal analysis methods, individuals have also put forward practical principals which govern how a protocol should be effectively designed and implemented to avoid certain types of attacks and flaws [2, 37, 40].

1.1 Multi-Dimensional Protocol Engineering

Specialized tool support for formal methods can significantly aid protocol engineers in creating and implementing cryptographic protocols which do not leak information, achieve their goals and are immune to replay attacks. In fact, protocol design and analysis has become so advanced and complex that we cannot perform certain analyses by hand as they take too long and tend to become tedious and error-prone over time. Tool support would also aid in the automatic generation of source code, thus helping to prevent careless errors that often creep into protocol implementations. Each of the techniques currently available to the security community is not capable of detecting every possible flaw or attack against a protocol when used in isolation. However, when used in combination with other formal methods, these techniques all complement each other and allow a protocol engineer to obtain a more accurate overview of the security of a protocol which she has designed. In effect, what we require is a unified approach to protocol engineering, one which combines a number of protocol engineering dimensions into one application that is consistent and easy to use, aids in the rapid construction, analysis and implementation of security protocols, and helps designers to distil and focus on the critical issues in the engineering process [5]. Such a *multi-dimensional* approach would fulfil a useful and fundamental role as an integrator and enabler in the discipline of security protocol design.

There will always be a need to design new protocols or integrate and upgrade existing ones. As e-commerce becomes more pervasive, we will witness the creation of even more cryptographic protocols which seek to facilitate the secure negotiation and exchange of goods and finances, as well as the authentication of individuals involved in these transactions. By supplying protocol engineers with a multi-dimensional protocol engineering tool, we equip them with an easily accessible array of proven techniques with which they can construct, test and deploy cryptographic protocols in a controlled, orderly and carefully planned manner. Such a tool would also be able to aid in the analysis and verification of existing protocols. Each of the engineering dimensions would realize a given technique and would also be appropriately automated, receiving the necessary input to begin its operation from the user or other engineering modules embedded within the tool. A benefit of multi-dimensional protocol engineering is that it would make protocol engineering and analysis operations accessible to a wider range of individuals, since it would remove the requirement that protocol designers need to experts in a large number of specialized engineering techniques. For example, in the case of automatic code generation the tool would be able to produce high-quality Java source code for each principal involved in the protocol without having the requirement that the user be proficient in the Java programming language.

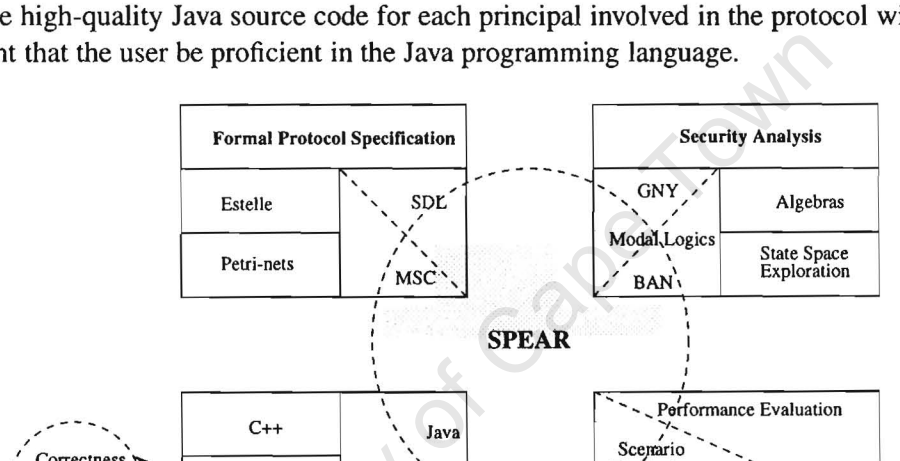


Figure 1.1: SPEAR I conceptual overview.

We believe that by encouraging protocol designers to make use of proven construction and analysis techniques throughout the protocol engineering process, the number of flawed security protocols that are produced could be significantly reduced. Multi-dimensional security protocol engineering is an effective approach for creating and deploying cryptographic protocols, since it encompasses a variety of analysis techniques, thereby providing a higher security confidence than individual approaches can achieve. A tool that employs a multi-dimensional engineering approach should be consistent and easy to use so that the techniques it implements are straightforward to apply. An example of a multi-dimensional protocol engineering tool is the SPEAR I application [5]. SPEAR I provides developers of cryptographic protocols with an environment in which to design, analyze and generate security protocols. In order to provide multi-dimensional support, SPEAR I enables protocol specification via a graphical interface in the style of Message Sequence Charts (MSCs) [44] and SDL [43], security analysis based on the BAN cryptographic logic [1], simple performance analysis using synchronous message rounds calculations [35] and Java source code generation. The fusion of these approaches allows the user to create cryptographic protocols from within the controlled environment of SPEAR I, with the specific aims of producing secure and efficient protocol designs and supporting the 'production' process.

Figure 1.1 contains a conceptual overview of the SPEAR I Framework, illustrating the dimensions incorporated therein, as well as the techniques within these dimensions that have been proposed and implemented. To use the SPEAR I application in a protocol engineering project, a user must first specify the protocol using the *Formal Protocol Specification* dimension. Once the specification is complete, Java source code can be automatically generated by using the *Code Generation* dimension. Scenario simulation that helps one to model the execution of the protocol is part of the *Meta-Execution* dimension, while performance analysis is carried out with synchronous round calculations. To determine whether a protocol achieves its goals, simple analysis may be conducted using the BAN modal logic, which is part of the *Security Analysis* dimension. All of the different techniques within each dimension work together and in some cases obtain information from each other. For example, the code generation engine requires details about the format, structure, source and destination of messages to create the message passing routines. This information is obtained from the *Formal Protocol Specification* dimension. The SPEAR I project was essentially a proof-of-concept implementation carried out over an eight month period and served to illustrate the viability and usefulness of a multi-dimensional approach. However, as a result of its prototypical nature, it was not solid or robust enough for commercial or academic use.

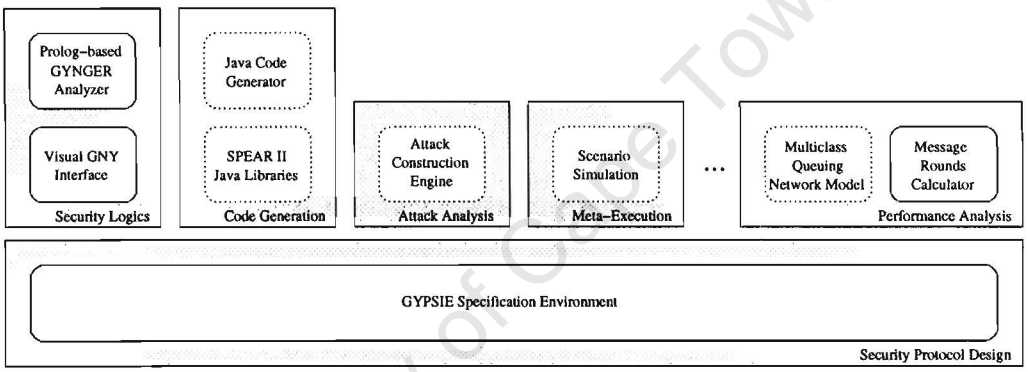


Figure 1.2: The current scope and ambitions of the SPEAR II Framework.

1.2 The SPEAR II Framework

The SPEAR II Framework [71] is based on concepts developed during the SPEAR I project and is essentially the continuation of this work. In Figure 1.2 we present a layered view of the SPEAR II Framework. Completed modules within this framework are indicated by solid outlines, while future modules are denoted by dotted outlines. The diagram clearly shows that the *Security Protocol Design* dimension serves as a basis for the other dimensions and that all of the SPEAR I dimensions are incorporated within the SPEAR II Framework. Notice that an *Attack Analysis* dimension has been added to the framework and there is also a separate *Performance Analysis* dimension. Within each dimension a collection of engineering techniques that have been included within the framework are shown. As an addition to the *Performance Analysis* dimension, we anticipate the use of multiclass queuing networks to carry out performance analysis, much like what was done in [8] for the SET protocol. If a code generation dimension is present, then this further opens up the possibility for controlled protocol executions either for the purpose of actual performance measurements (complementing the *Performance Analysis* dimension) or the re-creation of possible attacks (augmenting the *Attack Analysis* dimension). We have omitted a number of SPEAR I modules from the diagram due to space considerations, but these aspects can still be incorporated within the SPEAR II Framework when required.

Because of architectural and language differences between the SPEAR I and SPEAR II applications, the Java code generation, BAN analysis and scenario simulation features are not available in the current SPEAR II implementation. In fact, the SPEAR II project is a complete rewrite and reformulation of the SPEAR I application and aims to develop an extensive and commercially viable multi-dimensional security protocol engineering tool. As part of this dissertation, we have developed the GYPSIE [72] module, which is part of the *Security Protocol Design* dimension, the Visual GNY [73, 74] and GYNGER[3] modules, which are both part of the *Security Logics* dimension, and the message rounds calculator, which is part of the *Performance Analysis* dimension. The GYPSIE module is a graphical protocol specification environment that is similar to the SPEAR I interface, but incorporates a number of enhancements. Specifications created in GYPSIE are used as input to the Visual GNY, GYNGER and rounds calculator modules. The Visual GNY module is a graphical environment that can be used to construct GNY logic statements [33] which are used by the Prolog-based GYNGER GNY analyzer. Results from a GNY analysis conducted by GYNGER are returned to the Visual GNY environment and appropriately displayed. The message rounds calculator can determine both synchronous and optimal rounds in a protocol.

1.3 Scope and Objectives

The primary objective of this dissertation is to develop tools and techniques which will facilitate rapid, accurate and rigorous engineering of cryptographic protocols. Essentially, our aim is to distance protocol engineers from the syntactical element of protocol design and analysis, so that they can focus more on the associated semantics and distil any critical issues that may arise. Our work will take place in the context of the SPEAR II Framework, and all of the concepts that we develop will be implemented and incorporated therein. In essence, we will be working on and expanding the *Security Protocol Design*, *Security Logics* and *Performance Analysis* dimensions of the SPEAR II Framework. The framework will be realized through the creation of the SPEAR II application. The primary goals which we have set out to achieve are as follows:

1. Develop a graphically-based protocol specification environment that will allow protocol designers to easily construct extensible models of security protocols. Since this design environment is part of the SPEAR II *Security Protocol Design* dimension, these models will have to be able to serve as a basis for a number of diverse protocol-engineering operations which will be implemented by other SPEAR II modules. The graphical environment has been named '*GYPSIE*', which stands for 'Graphical Protocol Specification Environment'. Besides the obvious user interface requirements of the design environment, GYPSIE will also include an extensive API set that can be used to query a number of issues related to a protocol model. This API set will be crucial for integrating other SPEAR II engineering modules within the SPEAR II application.

2. Incorporate and implement a message rounds calculator within the SPEAR II Framework. This calculator will determine both synchronous and optimal rounds, each round consisting of the set of messages which can be sent and received in parallel. The rounds calculator will be embedded within the SPEAR II application and will use the GYPSIE API to obtain the message passing specification. No other information besides the messages and their embedded components will be required for this calculation. The resultant rounds will be displayed in an appropriately constructed dialog box within the SPEAR II graphical user interface.

[3]Named after Dean Kamen's top-secret GINGER invention (see http://ginger.patentcafe.com).

3. Integrate another graphically-based environment within the SPEAR II application which protocol engineers can use to construct and collate GNY logic statements applicable to a given security protocol. These statements will be used to specify the initial and target beliefs and possessions of principals as well as extensions to be appended to message components. An environment of this nature has not yet been created within the security community and the challenge in this case is to create it so that individual users do not get bogged down in the GNY syntax while using it. Instead, it must function as an enabler, freeing protocol engineers to focus on the important issues in an analysis. This environment will be known as '*Visual GNY*' and will work in close conjunction with GYPSIE to facilitate the exchange of information in the protocol specification.

4. Implement a GNY-based protocol analyzer to derive all possible GNY statements applicable to a given protocol and to determine whether a given protocol achieves its design objectives. The analyzer will be a standalone program, however in the context of the SPEAR II Framework it will work in close conjunction with the Visual GNY environment, receiving all of the input needed for analysis from it. When an analysis is complete, the results will be output to file, allowing the Visual GNY environment to retrieve, parse and display the results so that a user can view the outcome of the analysis. There are a number of implementation languages which can be used to create the analyzer, however we have decided to settle with Prolog [30], since it is simple to use and well suited for creating the forward-chaining [70] analysis system that we envisage. The analyzer has been given the name '*GYNGER*'.

5. Conduct experiments with users to ascertain the suitability of the GYPSIE and Visual GNY environments. These experiments will seek to determine the extent to which protocol engineers are able to easily use the environments and be productive therein. Experiments with the GYPSIE environment will focus on determining whether it does indeed facilitate accurate, efficient and effective protocol modelling, while the Visual GNY experiments will seek to examine how well its graphical metaphor aids users in constructing GNY statements. The GYNGER environment will be examined by carrying out analyses on well-known protocols and then comparing the results to what we expect. The message rounds calculator will be examined in a similar manner. In total, we expect to test the *Security Logics* dimension of the completed SPEAR II application on fifteen published cryptographic protocols.

To a large extent, the major thrust of this dissertation is the development of a completely graphically-based environment that will facilitate the GNY-based analysis of cryptographic protocols. There are a number of other security logics available, such as BAN [1], AT [3] and SVO [79]. However, for the purpose of this project, we have specifically selected the GNY modal logic. In part, this choice is due to the existing investment and interest that we already have in GNY-based analysis. The GNY logic is used extensively in postgraduate security courses offered by the University of Cape Town's Computer Science Department, and we had already spent time priming it for automation and inclusion in SPEAR II. While lecturing GNY principles, we have found that many students considered it to be tedious and cumbersome. However, these opinions were mainly derived from wrestling with the syntax and manually applying the eighty-eight inference rules to develop proofs. In this regard, GNY is a logic system in need of automation and a graphical approach to facilitate its effective use and application. The choice of security logic does not affect the integration of future logics within the *Security Logics* dimension of the SPEAR II Framework. In fact, we wish to encourage future work that will integrate more logics within the system, perhaps even harmonizing the input environments for each logic in the process.

Thus, at the conclusion of this dissertation we will have developed a graphically-based application (SPEAR II) based on the SPEAR II Multi-Dimensional Protocol Engineering Framework within which a security protocol engineer can easily model a security protocol (using the GYPSIE environment) and specify its associated GNY logic preconditions and target goals (using the Visual GNY environment). When using the SPEAR II application, a protocol engineer will be able to seamlessly interface with a custom-built Prolog-based GNY analyzer (GYNGER), which will be employed to determine whether the protocol specified in the GYPSIE environment attains all of the cryptographic goals which the engineer requires. GYNGER will also generate all possible GNY statements that can be derived to describe the protocol's final state. Results from GYNGER will be appropriately displayed in the Visual GNY environment. SPEAR II will also incorporate a message rounds calculator that will be able to generate the synchronous and optimal rounds for any given cryptographic protocol. This rounds calculation will aid in performance measurements. Finally, SPEAR II will be written so that it is extensible and allows further protocol analysis, engineering and design techniques to be incorporated.

1.4 Dissertation Outline

This dissertation is essentially divided into two parts. The first focuses on the background and development of the GYPSIE protocol design environment, while the second focuses on the specification and automation of the GNY analysis process. These two parts are covered by five chapters, the first two chapters discussing the facilitation of protocol design and modelling, and the last three describing the facilitation of automated protocol analysis. A brief summary of each of the chapters appears below:

Chapter 2 presents a background to the field of security protocol modelling environments. In particular, we discuss four well-known modelling environments and then present a comparative summary of their salient features. The main aim of this chapter is to familarize the reader with some of the different protocol engineering tools that are available within the security community.

The GYPSIE protocol modelling environment is described in **Chapter 3**. We begin this chapter by developing a set of requirements that we believe that a flexible and generic security protocol modelling environment should implement. Thereafter we describe the core features of GYPSIE and the message rounds calculator. User experiments which we conducted are then described and the results are discussed. The questionnaire that was used during the experiments can be found in **Appendix C**. We will also elaborate on some of the implementation details related to the GYPSIE portion of the SPEAR II application by describing the class hierarchy, file formats, memory management and the API set.

A GNY logic primer is presented in **Chapter 4**. The aim of this chapter is to introduce the GNY modal logic. We will sketch the fundamental assumptions underlying GNY and introduce core definitions and concepts related to the syntax and semantics. Thereafter we will describe the inference rules, give hints for determining protocol goals and discuss modifications which have been made to the GNY postulate set. We will also work through two example analyses to show how useful results can be achieved from an analysis. All eight-eight of the GNY inference rules are listed in **Appendix A**.

The aim of **Chapter 5** is to describe the automated GYNGER analysis tool which we have developed using Prolog. We will describe how the analyzer uses the inference rule set and then present an informal proof to show that a finite number of conclusions will always be derived in a finite number of steps. The implementation details pertaining to the analyzer will then be described using fragments of source code. Four sample analyses that were first used to test the analyzer will be described and the results that GYNGER derived will be presented. The proofs generated during these analyses are listed in **Appendix B**.

Chapter 6 describes the Visual GNY environment which has been created to facilitate GNY-based protocol analysis. We briefly introduce the problems associated with manual GNY analysis, before moving on to describe some existing graphical approaches to BAN and GNY-based analysis. Thereafter, we describe the graphical metaphor which we have developed, followed by an overview of the Visual GNY environment. User experiments that were conducted in the environment will then be discussed. The questionnaire used in these experiments is available in **Appendix D**. This is followed by a discussion of key portions of the Visual GNY implementation, such as the class hierarchy, file formats and selected algorithms.

We conclude this dissertation in **Chapter 7** by summarizing the contributions of this work and proposing projects for future work. The highlights of the chapters pertaining to GYPISE, GYNGER and Visual GNY are all presented. Furthermore, in this chapter we allude to **Appendix E** which contains summarized results from eight GNY-based protocol analyses that were all conducted using only the SPEAR II application.

Chapter 2

Security Protocol Modelling Environments

"We've realized that the fundamental problems in security are no longer about technology; they're about how to use the technology. Security is a process, not a product."

— *Bruce Schneier, Counterpane Internet Security*

Security protocol designers often need to implement and analyze security protocols using modelling or engineering tools. In order to be of any use, these tools must have an interface or environment which the designers can use to specify the protocol under examination. Certain environments are specifically designed to work with a fixed set of analysis techniques or tools. These environments are not generic and cannot be used to create protocol specifications for tools outside of their intended scope. However, other environments exist which can create more general protocol specifications that are of use to a broader range of applications.

There are currently no standardized graphical environments or techniques for specifying security protocols so that they can be used as input to any formal analysis tools. Standardized formal techniques such as Estelle [18], SDL [43] and Message Sequence Charts (MSCs) [44] do exist, however, these are not specifically geared towards the rapid specification of security protocols. The aim of this chapter is to introduce a number of security protocol modelling environments and then present a comparative evaluation of these systems. The environments that will be examined are the Convince Toolset [39, 17, 50, 51], the Interrogator [24], SPEAR I [5] and CAPSL [28], since they are in our opinion the most prominent and well-referenced security protocol modelling environments available at present.

2.1 The Convince Toolset

Convince is an automated toolset developed by Steve Brackin in 1995 to facilitate the modelling and analysis of cryptographic protocols, particularly those supporting authentication. An analyst can use Convince to construct a model of a protocol and then verify whether it meets user-specified security goals through the application of the BGNY modal logic [13]. The toolset consists of a combination of commercial and public-domain software. The commercial Software Through Pictures (StP) system [42] serves as a front-end to the system, allowing an analyst to model a protocol using a combination of graphical and textual notations. The analysis process is implemented by a freely-available Higher Order Logic (HOL) [38] theorem prover from Cambridge University. The exchange of information between these two components is facilitated by the LEX and YACC parser construction tools [4].

An overview of the three significant modules of the Convince toolset is presented in Table 2.1. This table illustrates the purpose of each module, the theory and programming that was required to create it, and the output that it produces. By leveraging existing software and technologies, the creators of Convince were able to construct it within one person-year.

Application	Purpose	Programming	Output
StP/OMT	Graphical Interface	Scripts to Produce ISL	Protocol Model and ISL Specifications
LEX/YACC	Translation from ISL to HOL	ISL to HOL Translator	HOL Specifications
HOL	Formal Theorem Prover	BGNY Belief Logic Theory and Proof Procedure	Proven and Failed Goals

Table 2.1: An overview of the Convince Toolset.

The Convince toolset has been used, amongst other purposes, to model and analyze three SPX protocols [80], Tatebayeshi-Matsuzaki-Newman (TMN) [50] and three versions of Kerberos [78, 49]. Three simple protocols involving key-exchange functions have also been modelled. Each of these protocols were reportedly analyzed in a relatively short time. The four-principal version of the Kerberos protocol was modelled in about an hour and then had its properties proved in five minutes.

2.1.1 Software Through Pictures

The StP/OMT object modelling tool is a Computer Aided Software Engineering (CASE) tool that facilitates the development of software systems using the Object Modelling Technique (OMT) methodology [41]. This methodology was originally developed by James Rumbaugh in the late 1980s and has recently been enhanced through the efforts of Jacobson, Booch and Rumbaugh. The focus of OMT is primarily on the development of custom software implementations, however, the creators of Convince have adapted the methodology to facilitate the modelling and analysis of authentication protocols. The application of the OMT methodology is automated by StP/OMT through the provision of a set of graphical editors which can each be used to create OMT software models. Convince utilises the Use Case, Event Trace and Dynamic Model StP/OMT editors, each of which provide different levels of abstraction.

Use Case models are an extension to the original OMT formalism. Within Convince, Use Case models are used to provide a structure for organizing protocol specifications. A Use Case consists of one or more actors connected to one or more processes. Each process can be associated with one or more Event Trace models, each representing a specific protocol scenario. In Convince, actors within a Use Case represent principals in an authentication protocol.

The sequence of message transfers that comprise an authentication protocol are described using *Event Trace* diagrams. Each Event Trace consists of a context object, a set of object classes, and a set of directed line segments denoting message transfers between the classes. Each message transfer is labelled with a text string stating the nature of the message and the stage of the protocol at which the transfer occurs. In Convince, an object class is considered equivalent to a principal in an authentication protocol.

The manner in which a principal responds to the receipt of messages is defined by a *Dynamic Model.* This model takes the form of a finite-state machine which defines how an object class responds to input events. A Dynamic Model can be represented as a set of concurrent sequences of state transitions, or as a single state transition sequence. Message transfers that are produced by a class are represented as output

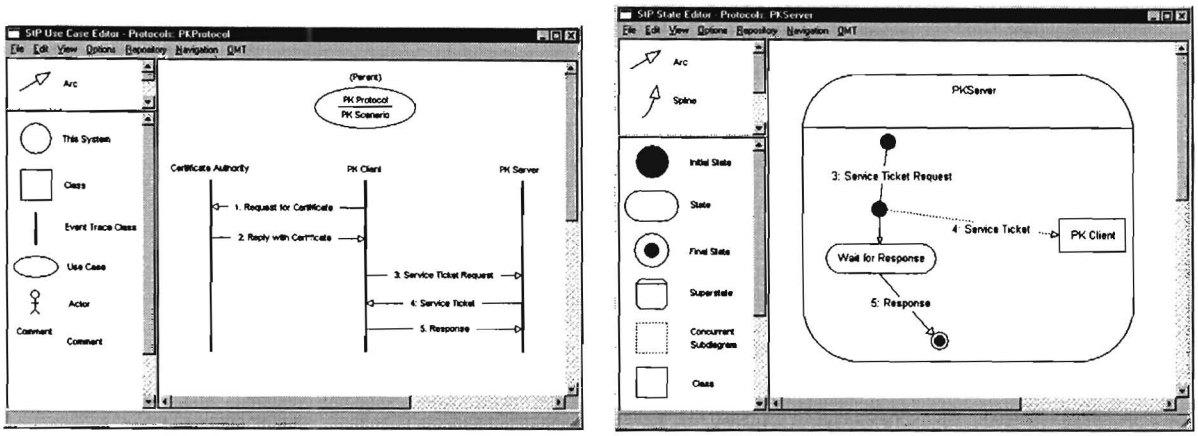


Figure 2.1: The StP Event Trace and Dynamic Model editors.

events.

In order to completely describe and verify the correctness of authentication protocols, the notation provided by StP/OMT was extended through the use of annotations. Each annotation provides additional protocol information related to the StP/OMT model elements. In Convince, the model elements which require annotations include principals, message transfers, context objects and principal states. A principal annotation denotes initial conditions, goal statements and the name to be used for the principal in message descriptions. This allows one to use longer and more descriptive principal names in OMT diagrams while using shorter, equivalent names in formulae. For message transfers, annotations represent the structure of messages conveyed between principals. Annotations associated with a principal's state correspond to statements in a belief logic. Finally, annotations associated with a context object within an Event Trace identify the names of principals, keys, cryptographic functions, hash functions and other functions and data referenced in other annotations.

2.1.2 Interfacing with the HOL Theorem Prover

The Interface Specification Language (ISL) [15] is a textual language whose syntax is a superset of the annotation syntax employed within StP/OMT. To generate the ISL specification of a protocol, the command option within the StP/OMT interface is used to extract the information specified in the model and produce the ISL output. The generated ISL specification has four major sections:

1. A set of *definitions* for data types.
2. A set of *initial conditions*.
3. The *protocol* description, described as a sequence of message transfers.
4. A set of *goals* detailing what the protocol should achieve.

In Convince, verification of an authentication protocol uses a HOL theorem prover. This necessitates translating the ISL specification into a HOL-compatible form prior to the proof process. A translator that was created with the LEX and YACC tools is used to perform this translation task before initiating the automatic proof process.

2.1.3 Specifying and Analysing a Protocol

To conduct a protocol analysis, a user first requires a description of the protocol. From this description, which is usually in text form, the user creates a protocol model by first defining the protocol elements within StP/OMT. The user then creates a Use Case diagram and associates it with a specific protocol scenario. This scenario is edited by using the Event Trace tool, which automatically creates a context object. In the Event Trace tool, the user adds vertical bars to represent principals, and labels them accordingly. A set of labelled directed line segments is added to denote the set of message transfers that occur as part of the protocol scenario.

After completing the Event Trace diagram, the user constructs a Dynamic Model for each of the principals. The start state is represented as a solid circle, intermediate states as rounded rectangles, and the end state as a bull's eye. Transitions between states are represented by directed lines whose labels denote the received events responsible for triggering the transitions. Message transfers that are initiated by the principal are represented as output events. These are associated with directed lines connecting a state transition to the principal who is the recipient of the message.

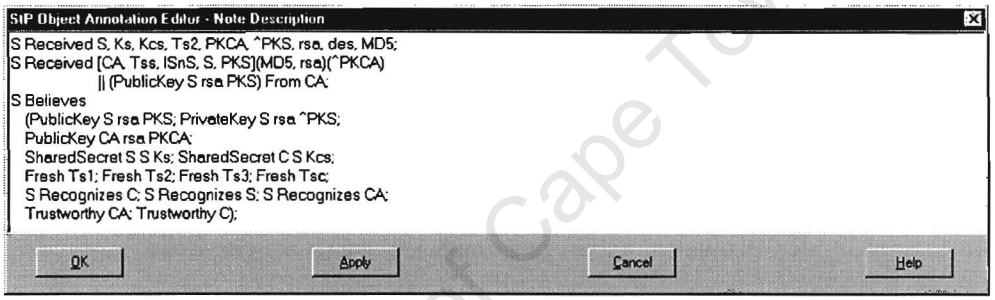


Figure 2.2: An StP annotation editor representing a principal's start state.

Generally speaking, the start state of a Dynamic Model corresponds to a subset of the initial conditions for the protocol. Accordingly, for each start state, the user provides annotations that represent the initial conditions of the corresponding principal. After adding the initial conditions to the model, the user provides annotations for the intermediate and end states. The annotations represent goals for the protocol which should become true once the protocol reaches a specific state.

Once the initial conditions, transactions, and goals have been input, the user directs Convince to convert the model to an ISL specification. This specification is then translated into a HOL-compatible form and the proof process is then invoked. The translator and HOL subsystems of Convince can also be invoked without the StP interface. In this case, the user must prepare or already possess an ISL specification in a text file. The name of this file is then passed to the translator as a command-line argument and the proof process is invoked.

Upon completing the proof process, Convince produces screen output, indicating whether it managed to prove all of the desired user-goals. If it cannot automatically prove a goal, it displays the goal to the user and then terminates the theorem proving process. ISL output files are produced describing proven and unproven goals. A goal failure can indicate that either the protocol's specified initial conditions are insufficient, or there is some other error in the formal description of the protocol, or the protocol is flawed. The user can repeat the modelling and analysis process, making changes to the model until he is satisfied that the protocol is flawed or does not have a flaw detectable by the HOL implementation of the BGNY logic.

2.2 The Interrogator

The Interrogator is a Prolog [25] program developed by Jonathan Millen, Sidney Clark and Sheryl Freedman in 1985. Using the Interrogator, a protocol engineer can search for security vulnerabilities in network protocols for automatic cryptographic key distribution. Given a formal specification of a protocol, the Interrogator searches for message modification attacks that defeat the protocol objective and reveal secret information. The current version of the Interrogator assumes that the penetrator is trying to learn private information, and the only way in which he can get that information is by reading a message in which it is transmitted as a data item. A black-box view of the Interrogator is simple: for input it receives a protocol specification and a target data item; its output is a message history, consistent with the protocol specification, showing how the penetrator could obtain the data item, if this is possible.

The Interrogator and its associated graphical interface were implemented using LM-Prolog on a LISP machine. The user interface takes advantage of the windowing, graphics and mouse capabilities of the LISP machine. Within the Interrogator, protocols are modelled using a state-transition approach, principals being represented as communicating finite-state machines. This method allows a wider class of protocols to be supported and permits variations in message sequencing. The Interrogator interface has two main components: a preprocessor that converts textual protocol specifications into an internal Prolog form, and a display interface for graphical user interaction. To conduct an analysis, a protocol is specified in a textual format, edited with normal LISP machine facilities, parsed and loaded. The interactive graphical display is then used to establish penetration objectives. If a vulnerability is found, it is displayed in the form of a message sequence, showing messages before and after modification by a penetrator.

The Interrogator has been developed to the extent where it has succeeded in finding a multiple-modification penetration of the Needham-Schroeder protocol [63] and some others with known vulnerabilities. Given a protocol specification and a target component to uncover, the Interrogator searches for a scenario involving penetrator actions which reveal the target. The history of messages sent and modified is displayed, allowing the user to examine how the penetration was carried out and evaluate it for feasibility and possible counter-measures. The search for penetrations is exhaustive within the allowed possibilities, but is it goal-directed and avoids search paths which are clearly futile. Execution times vary considerably depending on the initial assumptions and guidance given to the program. For example, the Needham-Schroeder example can take as little as thirty seconds or as long as two minutes to execute using uncompiled LM-Prolog on a Symbolics 3670.

2.2.1 The Preprocessor

The preprocessor carries out a straight-forward file-to-file conversion. Its input file is a textual protocol specification, while the output file contains transmit and receive clauses defining the finite state machine for each party in the protocol. A protocol specification is essentially a list of messages preceded by declarations of the symbolic constants used in the messages. Specifying a protocol with a message list means that one is specifying it by giving a normal history. Although this is common practice when describing protocols, it should be kept in mind that some protocols cannot be specified in this way. In particular, this approach would be inadequate for any protocol that offers a choice of responses at some point, where the decision does not depend on information deducible from the protocol.

The textual protocol specification consists of a number of fields which stipulate protocol data items, relations that exist between these items, data items that are known to an attacker, and the messages that are transmitted through the network. A sample specification is shown in the following example:

```
PROTOCOL example
CONSTANTS
    a, b, x, kdc : address
    ka, kb, kx, ck, oldck : key
    d : data
RELATIONS
    a, kdc : secret_key(a, ka)
    b, kdc : secret_key(b, kb)
    x, kdc : secret_key(x, kx)
KNOWNS
    a, b, x, kx, oldck, kb[oldck, a]
MESSAGES
    (* request ck *) a -> kdc: b
    (* generate ck *) kdc -> a: ka[ck, kb[ck]]
    (* forward ck *) a -> b: kb[ck]
    (* send data *) b -> a: ck[d]
END
```

The intent of most of the sections of the specification above are obvious, but two will be explained more clearly, namely KNOWNS and RELATIONS. Constants and expressions listed as KNOWNS are assumed to be known to the penetrator prior to the penetration attempt. Besides routine items such as the addresses of all principals and various data belonging to the penetrator as a legitimate network user, this list also contains any data that the penetrator might have recorded from some previous use of the protocol, such as the encrypted field `kb[oldck, a]`, and any data that is assumed to have been compromised, such as the old encryption key `oldck`. RELATIONS declarations are used to indicate relationships that are assumed to hold between certain pairs of constants, implying that if one value is known, the other can be calculated or looked up. Some relations are private and this is indicated by prefixing the relation with a list of addresses of those parties knowing it.

The parsing phase of the preprocessor is a rapid prototype, implemented with the help of the definite clause grammar facility supported by both the Edinburgh and the Uppsala implementations of Prolog. The conversion phase of the preprocessor takes the parse tree produced by the preprocessor and creates a state machine representation of the protocol, one machine per principal participating in the protocol. The preprocessor works its way through the messages in the protocol, constructing a transmit and a receive clause for each one. Besides the transmit and receive clauses for the protocol, the preprocessor also adds other clauses needed to complete the internal form of the protocol specification. It carries over the type declarations, relations and initial knowledge, and creates clauses for format declarations, and for the initial and final network state of the normal history.

2.2.2 The Display Interface

The display interface is a vehicle for setting up Interrogator runs and viewing the results. It provides a mouse-sensitive window for displaying protocol histories, selecting a message field as a penetration objective, and invoking the Interrogator. The window creation, manipulation, and display functions are written in Zetalisp. A facility exists for accessing LM-Prolog from Zetalisp and vice-versa, so that the display interface and Interrogator code can call each other. The following discussion will illustrate the capabilities of the user interface by tracing through a typical Interrogator session.

The display interface consists of a window which contains two subwindows within a larger frame. The upper window is a narrow command menu pane, while the lower window is a large display pane which is

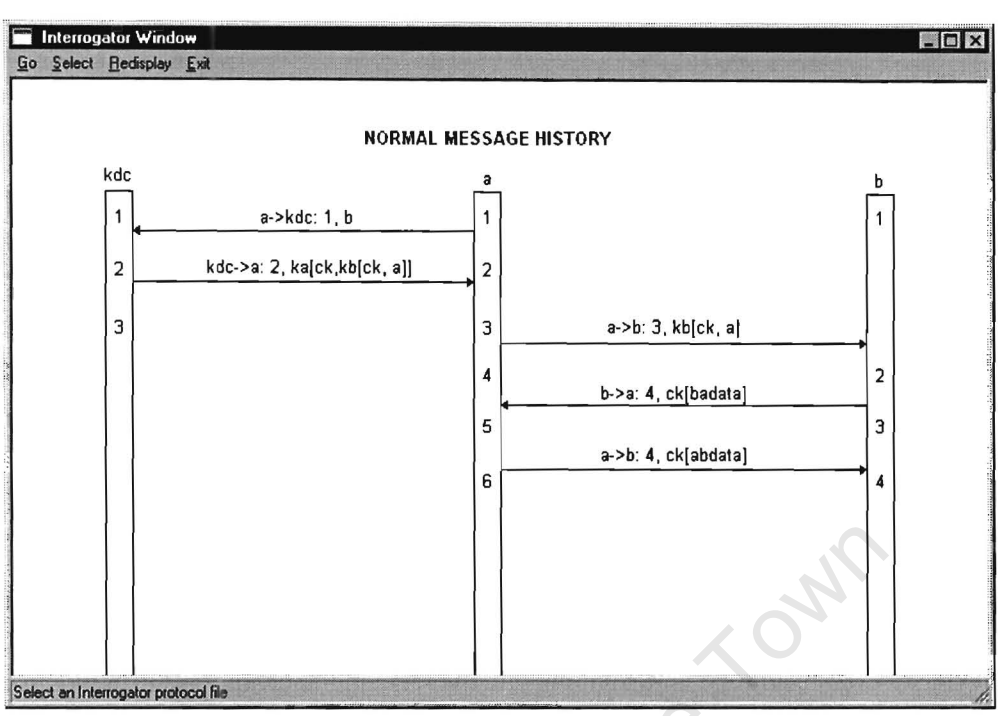


Figure 2.3: The Interrogator window showing a normal message history.

initially empty. The mouse-sensitive commands displayed in the upper window are listed and described below:

- *Go*: Invokes the Interrogator after the protocol and penetration objective have been indicated.
- *Select*: Displays a menu of protocols to analyse.
- *Redisplay*: Clears the display pane and redisplays the normal message history.
- *Exit*: Returns to LISP.

To use the Interrogator, the user first chooses a protocol via the *Select* command. If *Go* or *Redisplay* is picked first, a warning is displayed. The menu displayed by the *Select* command lists all of the files in a given directory that should only contain protocol files in internal Prolog form. Once the user has clicked on a file, the normal message history is determined and stored for later use in subsequent Interrogator activity. The display pane shows the normal message history in a graphical format. Each principal in the protocol is represented by a labelled column. Numbers inside each column indicate the state label for that principal, which changes as messages are transmitted or received. Messages are represented as labels on solid arrows showing the sequencing and direction of messages. Sometimes an arrow has to cross over a column between its source and destination. The columns are automatically ordered in the display so as to minimize such crossings.

After a normal message history has been depicted, the user must choose a message field as a penetration objective. Each message is mouse-sensitive and appears in reverse video when the mouse hovers over it. When the user clicks on a message, a temporary menu of the data fields within that message is created. This menu repeats the full message as its label. After the user has selected the penetration field, it is

displayed beneath the *Go* command. Once the user clicks on *Go*, the Interrogator searches for a solution message history by which it can obtain the penetration objective. During this processing phase, tentative and partial solution histories are shown in the display pane.

Some differences between the normal and penetrator message histories should be noted. No state numbers appear in the columns representing each principal and the messages are not mouse-sensitive. Vertically-orientated dashed lines drawn between each pair of communicating principals indicate the points of potential penetrator action in the protocol, when messages are travelling through the network between transmission and reception. Sent messages are drawn to the penetrator lines, while received messages are drawn from the penetrator lines. Message labels that are too lengthy to fit in the space provided are split into as many as three lines, where backslashes indicate the the last and first actual characters of the message's pieces. Message splitting also occurs when necessary in the normal history view.

At this point in the Interrogator session, the user has a number of options for further activity. He may click on *Go* and the Interrogator will continue to search for alternative penetrator solutions using the current data and state. The user may, instead, click on the *Redisplay* command, whereupon the display pane will be cleared and the normal message history for the protocol will be displayed once more. The messages will again be mouse-sensitive and the current state will be indicated in the columns. Here the user could click on *Go* again for a new solution with the same initial conditions, using *Redisplay* only to refresh his memory of the normal history. Alternatively, the user could select a new penetration objective, a new final state or a new protocol to analyse.

2.3 SPEAR I

SPEAR I, the Security Protocol Engineering and Analysis Resource, was developed by Paul de Goede, J.P. Bekmann and Andrew Hutchison in 1997 to aid in the design and analysis of cryptographic protocols. The two primary goals of SPEAR I are to enable secure and efficient protocol designs and to support the generation of protocol source code. SPEAR I offers developers of cryptographic protocols an environment in which to design, analyze and generate security protocols. Protocols are specified using a graphical user interface in the style of Event Trace diagrams. Security analysis based on the BAN cryptographic logic [1] is facilitated, while a meta-execution facility provides for the performance evaluation of cryptographic protocols from within a controlled environment. Java [45] production code can be generated once a protocol design has been analyzed.

2.3.1 An Overview of the User Interface

The SPEAR I interface consists of four main areas. The *design canvas* is used to create and specify a security protocol graphically. The *control area* consists of a menu-bar along the top of the application window and toolbars above and to the right of the design canvas. The menu-bar contains all the functions that are available to SPEAR I users, while the side toolbar contains all the protocol elements that are required to design a protocol. The *information canvas* on the right of the design canvas is used to display BAN beliefs and meta-execution data. Both the design canvas and the information canvas are scrollable and the amount of space that each canvas occupies can be adjusted by horizontally by moving the centre bar separating these two areas. Lastly, the *status bar* at the bottom of the application window is used to indicate which mode is currently selected when designing protocols. Most aspects of the SPEAR I interface can be customized by using the configuration option within the *Options* menu.

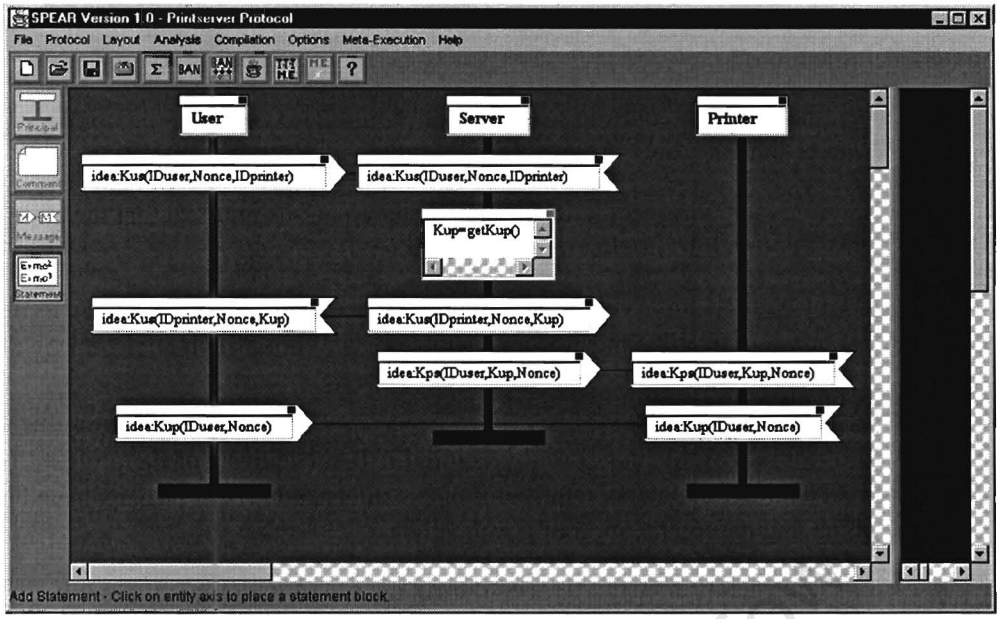


Figure 2.4: The SPEAR I user interface.

Items embedded within the pull-down menu-bar include the File, Protocol, Layout, Analysis, Compilation and Meta-Execution menus. The *File* menu allows users to start a new protocol and to load and save a protocol's possession declarations, function declarations and macro declarations. The *Protocol* menu allows users to switch between the modes that allow for the creation of principals, comments, messages and statements on the design canvas. This menu also allows a user to set the protocol name and declare all the possessions, functions, macros and initial BAN beliefs related to the protocol. The *Layout* menu is used to arrange the protocol elements on the design canvas in an aesthetic and easy-to-read format. The *Analysis* menu contains functions for security and performance analysis. Initial performance evaluation is performed by generating synchronous round information, while security analysis is performed by executing a BAN compilation of the entire protocol. The *Compilation* menu is used to generate Java source code, and lastly, the *Meta-Execution* menu provides facilities for simulating protocol runs from within the SPEAR I environment.

2.3.2 Specifying a Security Protocol

There are a number of steps which are necessary to implement a protocol design. SPEAR I provides guidelines to govern the order in which these steps should be carried out when working in the design environment. For example, an operation such as declaring a message sent between two principals cannot be performed until the principals have been declared. For this reason, it is often easier to generate messages, macros and perform other tasks if the associated possessions, functions and principals have already been defined. Operations which are not dependent on others can be performed at any time.

The SPEAR I interface makes extensive use of dialogs which contain lists of possible protocol elements to use at a given point during the construction of the protocol specification. For example, lists of previously declared possessions and functions are provided when defining macros and messages. This approach reduces confusion between function and possession names, and limits typographical errors and function usage problems. The optimum order for specifying a protocol using the SPEAR I interface is as follows:

1. Set the protocol's name. The name of a protocol will appear in the title of the SPEAR I window for quick identification.
2. The possessions that are to be used in the protocol should be declared at this point. Any amendments can be made at a later stage.
3. Function declarations that are specific to Java are declared at this point so that the code generation module will have enough information to call user code.
4. Macros that are to be used in the protocol can now be declared. These macros can then be used from within list boxes to quickly and easily generate declarations which are syntactically correct.
5. The principals involved in the protocol are defined.
6. The definition of initial BAN beliefs is only required if BAN analysis of the protocol is to be performed. The BAN builder dialog contains helpful constructs and lists of principals and possessions to aid in the quick and easy generation of BAN beliefs.
7. The possessions that each principal will need during the run of the protocol are initialized.
8. The actual communication between the principals is modelled by adding messages to the protocol and statement blocks to execute functions at certain stages of the protocol.

At this point the user will have fully defined a security protocol. Other engineering and analysis functions which SPEAR I provides can now be applied. This would include modifying the layout of the protocol on the canvas, carrying out performance evaluation using synchronous rounds and meta-execution, and conducting a BAN analysis. Once the user is satisfied with the protocol, he can generate fully functioning Java source code.

2.3.2.1 Declaring Protocol Components

The *Possession Declaration* dialog illustrated in Figure 2.5 is used to specify possessions that will be used during the protocol run. This dialog contains a list box on its right-hand side which enumerates all of the currently defined possessions. The left-hand side of the dialog lists information about the possession being edited. A given possession can either be an asymmetric key, a symmetric key, entity information, fixed length data, variable length data or delimited data. An entity information possession is an abstract notion used to represent a communicating principal and contains a name, an IP address and a port number. Delimited data is provided for use with existing protocols such as SMTP [67], which sends data delimited by particular characters. The other possession types are straight-forward.

Once a user is satisfied with the information specified for a possession, he can add it to the possession list. If any possession has the same name, then the user is given the option of replacing the existing possession's information with the newly specified information. Functions, possessions and macros may not have the same name. The dialog also presents the user with the option of removing possessions that have already been specified. To edit an existing possession, the user can double-click on a possession in the list and its information will be displayed on the left-hand area of the Possession Declaration dialog. The ability to load and save lists of possessions is very useful at this stage since a large number of protocols have similar types of possessions and the time needed to define these possessions can be saved.

Functions are declared by using the *Function Declaration* dialog which is shown in Figure 2.5. Since SPEAR I generates Java source code, the function declarations are specific to Java and are defined so that

the generated source code has access to user information. For example, the source code needs to look up a principal's IP address before transmitting it. The Function Declaration dialog operates in a similar way to the Possession Declaration dialog. The currently defined functions are enumerated on the left-hand side of the dialog and these are added and removed in the same manner as possessions.

Each function is given a name that SPEAR I will use to identify it in an expression. The notion of 'crypto-operands' and 'inputs' are used to distinguish keys and data. The normal method of writing *data* encrypted with a key K is $\{data\}_K$. However, this formatting is not possible in textual representations. Instead, SPEAR I formats this example as `DES:K(data)`, with the crypto-operand being `K` and the input being `data`. In practice the Java compiler requires function calls to contain the crypto-operands followed by the inputs. In this case, the actual function call for the above example would be something similar to `DES_encrypt(K, data)`. The user-provided Java class and method names are used to generate source code to call the actual methods which implement this abstract notion of a function. The return type of the function must also be specified by the user. The user must also specify the type of function and whether or not it has an inverse. If the function has an inverse, then the user is required to enter the inverse function name and the corresponding Java method details. For generality this is required, even if the same function is used to encrypt and decrypt data. As with possessions, the ability to load and save functions saves time normally taken by declaring functions for every protocol.

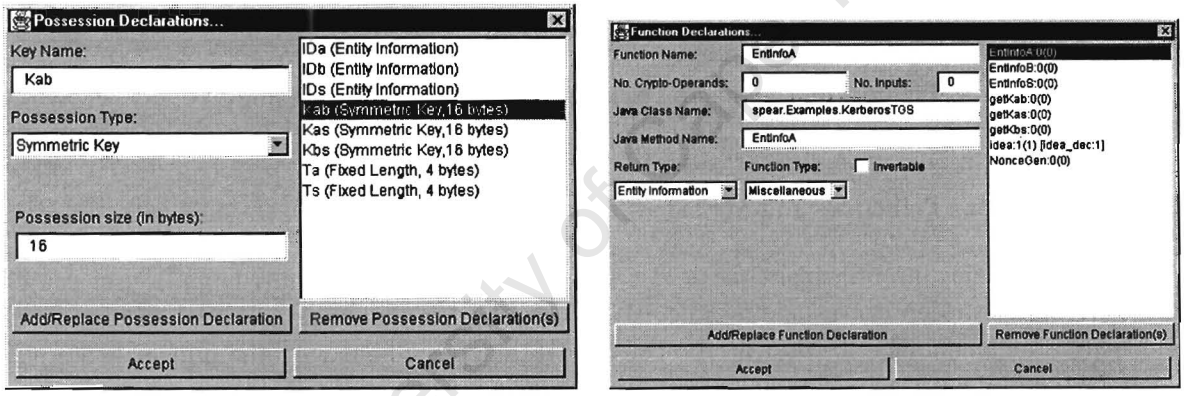


Figure 2.5: The Possession and Function Declaration dialogs.

SPEAR I macros are essentially the same as `#define` statements in C/C++. They are used to simplify the protocol by performing straight substitution, the macro name for the macro value when the protocol is compiled. This approach makes the protocol specification more readable and helps to avoid common errors. For example, it can be used in the Kerberos Ticket Granting Service protocol to simplify messages containing tickets. As with other protocol declarations, a special *Macro Declaration* dialog, shown in Figure 2.6, is used to facilitate macro generation. Adding and removing macros is done in a similar fashion to possessions and functions. The list of macros on the right-hand side of the dialog are those that are already defined, and the information on the left-hand side is for defining new macros. A user is required to enter a name for the macro and its real value. The macro name can then be used interchangeably throughout the protocol for the macro value. As with possessions and functions, double-clicking on a macro-definition in the right-hand list will display that macro's information in the left-hand information area. To further aid the specification of macros, the Macro Declaration dialog contains two list boxes which list functions and possessions which have already been defined. As in the case of functions and possessions, the macro declaration can be saved for use in other protocol specifications.

Initial BAN beliefs which are held prior to the commencement of a protocol run are entered using the *BAN*

Builder dialog which is illustrated in Figure 2.6. This dialog works in a similar manner to the function, possession and macro declaration dialogs — the area on the right contains the list of currently defined BAN beliefs, and the area on the left contains information about the belief currently being constructed. Double-clicking on a belief already defined in the right-hand list allows one to edit the belief. Along the top of the Belief Builder dialog are three lists containing already defined principals, BAN constructs and existing possessions. Double-clicking on these lists adds the selected element to the BAN belief currently being constructed. When entering the BAN belief, SPEAR I does not ensure that the belief is syntactically correct. Only once the BAN beliefs have been compiled can syntactic correctness be examined. The semantic meaning of beliefs is left to the designer to check.

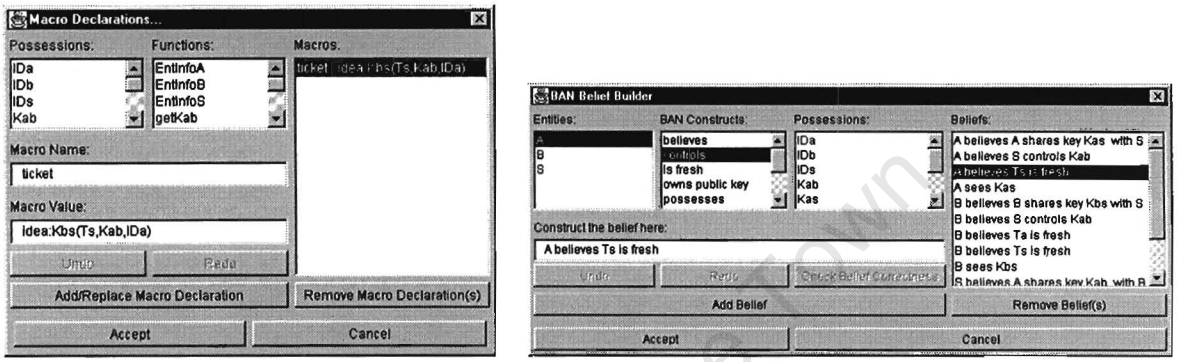


Figure 2.6: The Macro Declaration and BAN Builder dialogs.

2.3.2.2 Specifying Principals and Messages

To place communicating principals, the *Add Entity* mode is selected. At this point, a user can click anywhere on the design canvas and a principal will be added to the model. The initial placement of principals is automatic and the user has no control over the positioning. All the principals are placed along the top of the design canvas, with their axes extending downwards. These axes are automatically extended as more elements are added to each principal. By right-clicking on the black square at the top right-hand corner of the principal, the user can reposition the principal on the canvas by selecting the *Move* option from the displayed pop-up menu. The principal box can also be resized by clicking on the same black square and selecting the *Resize* option. A size restriction is placed on the principal box, ensuring that the user does not make it too small.

Messages that are created or edited have to be entered in a textual format using the *Expression Builder* dialog. SPEAR I uses the following grammar for specifying messages:

```
item ⟶ IDENTIFIER | function_call | list
list ⟶ { item more_items }
function_call ⟶ function_name optional_crypto_operand ( function_items )
function_items ⟶ item more_items | ∈
more_items ⟶ , item more_items | ∈
optional_crypto_operand ⟶ : crypto_operand | ∈
```

As can be seen from the above summary, a list can be specified by placing one or more items, delimited by commas, between curly brackets. A list may contain any number and type of items and essentially operates as a way of combining these items into a set. A function call is indicated by the function name,

and if there is a cryptographic operator associated with that function call, it is stipulated with a colon after the function name. This is followed by the data for the function call placed in brackets. Thus, examples of valid expressions are:

1. `{Kab}`
2. `DES:Kab(Message, SHA(Message))`
3. `{Kab, MD5(Message, Key)}`

Possessions need to be initialized with values before they can be transmitted. This initialization step is normally done at the beginning of the protocol, but it can also be done in any statement block by using the assignment operator. If the possessions will be initialized at the start of the protocol run, then the designer can use the principal popup menu to select the *Initialize Possessions* option. The user is then presented with the *Statement Builder* dialog, shown in Figure 2.7, which he can use to specify the assignment statements that will initialize the possessions. Typical examples of possession initialization statements include:

1. `IDa = getEntityInformationOfA()`
2. `IDa = getServerInformation()`
3. `Kas = getSharedKey(IDa, IDs)`

If possessions are not initialized before being used, then a compilation error will result when performing meta-execution or code generation of the protocol.

Messages are sent from a sender to a receiver. When the *Add Message* mode is engaged, the user must click on the originating principal axis to add a new message. At this point the *Expression Builder* dialog box, illustrated in Figure 2.7, will be shown and the user can type the entire message or use the supplied list boxes which enumerate all the possessions and functions that have already been defined. By double-clicking these functions and possessions, a user can add them to the message being defined. If the protocol has been compiled, then the user is able to check the syntactic correctness of the message specification. However, if the protocol has not yet been compiled, then this option is not available. Since compilation may be a lengthy and taxing process, the user must decide when to compile the protocol after making sufficient changes. For instance, it is best, but not essential, to compile after altering the declarations of functions, possessions and macros.

Once the user is satisfied with the message definition, he can press the *OK* button and indicate the receiver of the message by clicking the relevant principal axis. Senders are denoted by convex arrows on the right-hand side of the message text, while receivers have a concave section on the right-hand side. When either the sender or receiver is deleted then its corresponding messages are also removed from the protocol. This ensures the correctness of a protocol, by ensuring that messages are only sent to defined principals.

Statement blocks are primarily used for two purposes. One of these is to initialize possessions at specific points during the protocol run. For instance, a server may have to look up the information for a print server based on the information the client sends it. This could be written as follows:

```
IDpserver = getInfo(ClientInfo)
```

The second use of statement block is to perform arbitrary function calls to the user-defined code. This could, for example, be used for logging information the user wants to keep regarding service requests. The format of such a call would be as follows:

```
Log(IDclient, ServiceType)
```

Statements are placed on principal axes by clicking on the axis at the point where the block of code to be defined should be executed. The user is then prompted by the *Statement Builder* dialog to enter the desired statements. The user may enter as many statements as he want to in a statement box. As with other dialogs of this nature, the statements are listed in a list box on the right-hand side of the protocol and information regarding components that can be added to the statement appear on the left-hand side. If the protocol has been compiled and a valid Java compiler exists, then the user can also check the syntax of the statement. The order in which the statements are enumerated in the right-hand list box is the order in which they are executed. Once the user accepts the statement, it is placed on the canvas at the point where the user clicked the axis.

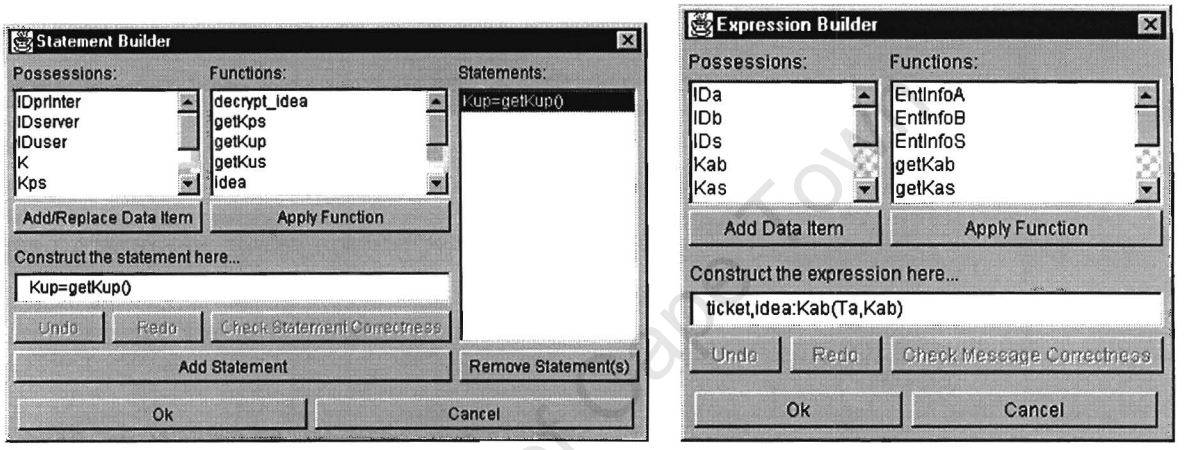


Figure 2.7: The Expression and Statement Builder dialogs.

2.4 The CAPSL Specification Language

CAPSL, the Common Authentication Protocol Specification Language, is a high-level language intended to support the analysis of cryptographic protocols using formal methods. The development of CAPSL was started in 1996 and is being managed by Jonathan Millen. Its goal is to permit a protocol to be specified once in a form that is usable as an interface to any type of analysis tool or technique, given appropriate translation software. The CAPSL Intermediate Language (CIL) acts as an interface to analysis tools, allowing protocols specified in CAPSL to be examined by these tools. CIL is designed to make the translation to tool-specific representations as easy as possible. A CAPSL specification is parsed and translated into CIL, and at that point a different translator can convert from CIL to whatever form is required for each tool. The translator from CAPSL to CIL can deal with the universal aspects of input language processing, such as parsing, type checking, and unraveling a message-list protocol description into the underlying separate processes.

A CAPSL specification is made up of three kinds of subspecifications: *type*, *protocol* and *environment* specifications, usually in that order. Type specifications define cryptographic operators and other functions axiomatically, and are also used to define different types of principals. A protocol specification contains declarations, messages and goals that the protocol should achieve. An environment specification is optional and is used to set up particular network scenarios for the benefit of model checkers, search

tools or other applications that need them. The following code sample is a small example of a protocol specification, illustrating several of the language's features:

```
PROTOCOL One_Message;
IMPORTS USER;
VARIABLES
    A, B: User;
    K: SKey, FRESH, CRYPTO;
ASSUMPTIONS
    HOLDS A: B;
MESSAGES
    A -> B: {A, K}Pub(B);
GOALS
    SECRET K;
    BELIEVES B: HOLDS A: K;
END;
```

This protocol has only one message in which user A sends a newly generated key K to another user B. The key K is concatenated with the identity of user A and encrypted with B's public key. Protocol variables are typed, and they may also have properties. For example, the property FRESH means that the value of the variable is newly generated for each session, and has not been used before by the principal generating it. The CRYPTO property means that the variable is unguessable. A specification may import another specification. Importing is understood as syntactically copying all definitions of the imported specification into the importing one. In the above case the USER type specification is imported into the current protocol. The HOLDS declaration implies that when a protocol session begins, A will possess the identity of user B. There are two security goals specified in the GOALS section, namely that the new key K is kept secret from any attacker, and that if the protocol session completes, B shares the key K with A.

2.4.1 Operators and Types

Messages in cryptographic authentication protocols are constructed using cryptographic operators and functions. In principle, all functions used in CAPSL, and the data types they operate on, must be specified axiomatically with abstract data type specifications, called *typespecs*. Several commonly used data types and operators are defined in a standard prelude. Type specifications in this prelude are considered built-in, and do not need to be supplied by a designer or imported explicitly. Data types defined in the prelude include `Principal`, `Skey` (symmetric key), `Pkey` (public key), `Nonce`, `Field`, `List` and `Timestamp`. Operators defined in the prelude include encryption operators such as `se` and `sd` for symmetric encryption and decryption; `ped` for public-key encryption and decryption; `kap` and `kas` for key agreement; `sha` for hashing; `cat` for combining items of type Field; `con` for creating values of type `List`; `seal` to create a checksum and `verify` to check it; and lastly `later` and `near` for timestamp comparisons. CAPSL also implicitly allows for finite-field arithmetic and boolean expressions to be evaluated and specified. An example of a type specification found in the prelude appears below:

```
TYPESPEC PKEY;
TYPES Pkey;
FUNCTIONS
    keypair(Pkey, Pkey) : Boolean;
AXIOMS
```

```
    keypair(J, K) = keypair(K, J);
END;
```

As has already been mentioned, protocol variables, besides having a type, may also have properties representing assumptions about the way in which they are generated or used. These properties generally do not affect the abstract model or representation of a protocol as a process, but rather affect the feasibility of certain types of attacks. The `CRYPTO` property indicates that a variable is unguessable; `FRESH` means that it is not reusable in other sessions by the same principal; `RANDOM` refers to the fact that the variable is unrecognizable; `EXPOSED` identifies a constant held by an attacker; and lastly `PRIVATE` is used to refer to a function computable only by a specific principal — essentially a local table lookup. Properties for a variable are stated after its type specification.

CAPSL provides for two types of concatenation, namely associative concatenation of fields, with the operator `cat`, and nonassociative concatenation to produce values of type `List`, using the operator `con`. Concatenation with `cat` is supposed to represent simple bit-string concatenation. All types in the prelude, except for `Field` and `List`, are subtypes of a type called `Atom`, which has a well-defined length. This allows `cat(X, Y)` to be uniquely decomposable into `X` and `Y`, if and only if either `X` or `Y` is atomic. Accessor functions named `first` and `last` exist for fields $\{u, v, \ldots\}$. These functions are well-defined when u is atomic; the value of `first` is u and the value of `rest` is $\{v, \ldots\}$. Concatenation with `con` constructs values of type `List`, where component delimiters are explicit and nestable in the concatenated field. Accessor functions named `head` and `tail` exist for `con`, which ensure that `head(con(A, B)) = A` and `tail(con(A, B)) = B`. A non-binary list is interpreted right-associatively. Thus `con(A, B, ...)` is interpreted as `con(A, con(B, ... ))`.

The expression `{A, K}Pub(B)` is interpreted as `ped(Pub(B), cat(A, K))`. This is a combination of two notational conventions, namely the use of `{A, K}` to abbreviate `cat(A, K)`, and the postfix of a `Pkey` term to a concatenation to encrypt it using `ped`. Decryption is indicated by placing a prime in front of the applicable key. For example, conventional decryption of `X` with the symmetric key `K` is indicated by `{X}'K`. The other concatenation operator, `con`, is indicated by using square brackets. Thus, `[A, B]` is interpreted as `con(A, B)`, and any encryption may be applied with postfix keys.

2.4.2 Messages

The `MESSAGES` section in a CAPSL specification contains messages and actions. A message is expressed in the form:

source $\longrightarrow$ *destination: field*$_1$, ..., *field*$_n$

In this notation the source and destination are variables of type `Principal`. Messages may also be numbered and preceded and followed by equational actions representing assignments or tests. A *phrase* consists of a message and the actions associated with it. An action prior to a message is performed by the sender of the message, and an action after the message is performed by the receiver. If the receiver of a message is the sender of the next message, any action between the two messages is performed by that principal, and it does not matter to which phrase the message belongs. If not, the association of actions with messages can be indicated with a '/' character, as in the following example:

```
A -> B: X;
X = Y;/
A -> C: Z;
```

Here the '/' shows that the action `X = Y` is performed by `B` rather than `A`, and it belongs in the phrase containing the first message. An equational action like `X = Y` may be either an assignment to `X` or a comparison test, depending on whether or not the acting principal already holds a value for `X`. The right side of the equation may be any term that is computable by the acting principal, while the left side of the equation is normally a variable, but it might also be any computable term, if a comparison test is intended.

Assignment actions do not have to be placed in the message list if they have already been placed in a `DENOTES` declaration. For example, the following `DENOTES` declaration defines a structure `PMK` that can be used in message declarations:

```
PMK = kas(kap,(Xs(S)), Xc(C));
```

A `DENOTES` declaration is a convenient way to prominently exhibit the structure of conceptually important items such as tickets or certificates. Through the use of these declarations a message specification can be simplified and made more abstract.

2.4.3 Assertions

CAPSL permits a variety of assertions to be stated about operators and protocol variables. Axioms in a type specification are universal assertions about the operators defined there. Assumptions and goals in a protocol are statements about protocol variables. The `ASSUMPTIONS` section in a protocol states initial conditions. An important use for this section is to indicate which variables are held initially by each principal. The `GOALS` section states the security objectives for the protocol. A goal referring to the state of a principal should be true for the final state of that principal. Statements intended as assumptions or intermediate proof obligations may be placed among the actions preceding or following a message. These assumptions are preceded with the keyword `ASSUME`, while proof obligations are preceded by the keyword `PROVE`. Examples of assumptions include:

```
HOLDS A : Ka
SECRET Ka : A, S
SESSION_SECRET R: A, B
BELIEVES A: HOLDS B: K
KNOWS B: SECRET Kab
```

The `HOLDS` keyword identifies the variables which a given principal possesses; `SECRET` indicates which principals hold the value of a given variable; `SESSION_SECRET` identifies the principals who may hold a variable's value during an entire session; `BELIEVES` is used to indicate the initial beliefs of a given principal; and lastly a principal `KNOWS` an assertion if the principal believes the assertion and the assertion is true. Principals are implicitly assumed to hold their own identities, so `HOLDS A: A` is unnecessary. Nonces may either be assumed to be held initially or not, while timestamps are implicitly assumed to be held by all principals.

2.4.4 Subprotocol Invocation

CAPSL allows a designer to identify subprotocols (functionally related groups of messages) in a protocol specification. Different concepts are involved, depending on whether a subprotocol is designed to be

subordinate to a larger protocol or whether one is in fact combining two independent protocols with a simple data flow relationship in the context of a larger system.

Subprotocols are usually offered as options to be chosen or negotiated, or applied in sequence. In CAPSL, one can replace a message with a conditional selection or a subprotocol invocation. The intent of the current CAPSL grammar is that subprotocols be declared in separate PROTOCOL specifications, which are invoked by name in the parent specification by using INCLUDE statements. This approach is illustrated in the following fragment of an SSL specification:

```
C -> S: C, CH, CS
IF CS = DH THEN INCLUDE SSH_DH;
ELSEIF CS = RSA THEN INCLUDE SSH_RSA;
ELSE ...
ENDIF
```

The child protocols import the parent protocol so that they can use the protocol variables. The parent may only refer to the names of the subprotocols.

2.4.5 Environments

When a protocol is being analyzed or simulated, the analyst may have to specify which principals are to be run. Other run-specific information, such as the initial knowledge of the attacker, may also have to be supplied. A CAPSL environment contains specifications detailing this kind of information. Environment specifications, like type specifications, are separate from the definition of a protocol. The content and interpretation of an environment specification depends on the analysis tool. However, CAPSL does provide syntax, keywords and organization so that different tools can take advantage of the CAPSL parser. Declarations to name principals and other constants can be placed in an environment, and sessions can be defined. More than one session may be declared and these sessions will run concurrently by default. Execution dependencies between sessions can be expressed, allowing some search tools to save time in the analysis process.

2.5 Closing Remarks

The modelling environments which have been discussed in this chapter are representative of a range of diverse approaches to security protocol specification. Table 2.2 presents a comparative summary of some of the attributes of the modelling environments presented. Different arguments can be made for the appropriateness of each interface and the selection of affordances used. Both CAPSL and SPEAR I attempt to create specifications which can be used as input to a large number of analysis tools and techniques. On the other hand, the Interrogator and Convince are more focused on creating protocol specifications that will be used within their intended scope of use.

Convince makes use of the OMT approach for modelling security protocols. This is one of the most generic approaches that can be applied. However, applying too many of the associated object-orientated techniques might lend itself to a system that is unnecessarily complex for cryptographic protocol engineering. The Event Trace-type approach employed by SPEAR I clearly represents the message flows between principals and is intuitive to use when creating security protocol specifications as all the relevant information and functionality is accessible on the design canvas. Both CAPSL and the Interrogator

Category	Convince	Interrogator	SPEAR I	CAPSL
Input Environment	Graphical	Textual	Graphical	Textual
Integrated Results Viewer	■	■	■	□
Logic Analysis	BGNY	N/A	BAN	N/A
Subprotocol Support	■	□	□	■
Message Rounds Calculation	N/A	N/A	Synchronous	N/A
Component Typing	■	■	■	■
Type Formatting	□	□	■	□
Source Code Generation	N/A	N/A	Java	N/A
Attack Analysis	□	■	□	□
Protocol Session Simulation	□	□	■	□
Export Formats	ISL	N/A	N/A	CIL
Embedded Statements or Conditions	■	□	■	■

■ = Yes □ = No

Table 2.2: Comparative evaluation of the modelling tools discussed in this chapter.

use textual specification techniques for creating the initial protocol specification. The GUI used by the Interrogator for creating attack sessions succinctly represents the message flows and also presents the user with a clear view of the possible components which can be used as search targets.

The ability of the CAPSL language to represent subprotocols is exceptionally useful for specifying large commercial protocols or protocols which have a choice of execution at some point. Convince allows for a similar representation by using dynamic models and finite state machines to indicate the messages which are sent when certain events take place. Although the Interrogator represents protocols internally using a state machine approach, protocols are defined by specifying only one possible scenario. Each scenario is input separately from the others and does not form part of a unified protocol specification. SPEAR I uses the same approach as the Interrogator with regard to subprotocols. Thus, analysis or code generation systems which require information about all the possible message flows would not benefit from the input schemes used by the Interrogator or SPEAR I.

All of the specification environments in this chapter provide some notion of component typing. SPEAR I ensures that every component has an associated type and format so that Java source code can be produced. CAPSL provides typing facilities which include types that are specifically suited to security protocols, such as nonces and timestamps, as well as user-defined types. Convince only considers typing keys, shared secrets and functions, as these must be explicit in the context of BGNY logic. The type specifications in the Interrogator are merely predicates, so virtually any named type can be supported by the tool. However, source code generation tools must know how to encode and decode types for transmission over a communications network. The SPEAR I code generation engine produces source code that encodes and decodes protocol components based on the type that they have been assigned. A difficulty with this encoding mechanism is that it is not consistent with known standards and thus SPEAR I protocol implementations can only communicate among themselves or with those that implement the same message format. The Interrogator, CAPSL and Convince do not even attempt to provide the ability to specify the format of a type.

Specifying statements to be executed between messages is a useful feature to have in a modelling environment, especially if source code is to be generated. The ability to specify conditions that should be satisfied before messages are sent or after they are received will also aid in later code generation or anal-

ysis. CAPSL allows designers to specify a set of statements or conditions to be executed or examined before a message is sent or after it is received. These statements are merely placed between messages in the specification and then associated with the sender or receiver. SPEAR I has facilities which enable designers to define a statement block that is executed between message flows. The positioning of these statement blocks indicates the principal who executes them and the point at which they are to be run or checked. Convince allows a designer to specify conditions that should be true after each message has been sent and received, however it does not provide the ability to specify statements to be executed. The Interrogator provides no way of defining intermediate code or conditions that should exist between messages as this is not required for the attack analysis that it conducts.

The environments presented in this chapter use differing degrees of graphical and textual specification techniques. On the one end of the spectrum, CAPSL is totally text-based. As a result of this fact, a designer creating a specification in CAPSL must have a good working knowledge of the associated syntax and semantic issues. The Interrogator also uses a textual interface to specify the protocol, however it makes use of a graphical environment to set up protocol attack searches. Between writing the specification and setting up penetration objectives, the designer has to compile the protocol specification into a Prolog-compatible form. In this respect, the Interrogator specification environment is similar to CAPSL since the protocol specification has to be typed out and then compiled. The penetration search engine can be thought of as a third-party tool for which output is being produced by the compiler, much like a CAPSL compiler will produce CIL output for other analysis tools. Convince allows a designer to create a specification in a graphical design environment and then export this specification to the textual ISL format which is used as input to the BGNY analyzer. Output produced by the analyzer is read back into Convince and displayed to the user through the graphical environment. SPEAR I can be considered as being almost fully graphical in nature, however entering message components and BAN beliefs is still done in a textual fashion, albeit with some guidance.

Chapter 3

Security Protocol Modelling with GYPSIE

"The wire protocol guys don't worry about security because that's really a network protocol problem. The network protocol guys don't worry about it because, really, it's an application problem. The application guys don't worry about it because, after all, they can just use the IP address and trust the network."

— Marcus J. Ranum

Cryptographic protocol design and modelling is a complex task that is not always straight-forward to carry out, since subtleties within a protocol specification can make it vulnerable to a number of attacks which can be used to subvert its original goals and intent [2, 22]. To facilitate the development of more advanced and reliable protocols, we need to ensure that protocol engineers have access to an environment from within which they are able to easily create a high-quality protocol specification. Such a specification can then form a basis for further security and performance analysis, as well as code generation. The main function of a modelling environment is to serve as a core around which other protocol engineering operations can revolve. For this reason, it should be simple to use, straight-forward and expressive. The protocol model which is created must also be exportable to a wide variety of formats.

The use of graphical interfaces substantially reduces the amount of knowledge people have to remember about an interface. [68]. In many situations, the intuitive and direct feel of the interface means that users do not need to think about what they are doing or remember sequences of commands. Instead, they primarily need to learn how to interact with a simulated world of objects. With a graphical interface we also find that much of the information about a system's structure and functionality is available within the interface, meaning that users have to remember less details to use the system. A key feature of well-designed graphical user interfaces is 'direct manipulation'. Direct manipulation is an interaction style in which objects are represented on a computer screen and then manipulated by the user in ways analogous to how the user would manipulate the real object.

The primary goal of the SPEAR II protocol modelling framework is to create an environment which allows for the specification of cryptographic protocols in such a way so as to distil the critical issues and present the user with varying levels of abstraction, each level presenting an appropriate view of the protocol design. The most suitable environment for this type of system is a graphical user interface that employs direct manipulation to allow for easy interaction with the objects in each of these abstracted views. By making use of a graphical user interface for protocol design we also ensure that the system is easy to use since users do not need to memorize a specific syntax to be productive. They also have instant visual feedback on how the protocol appears through each of the abstracted views.

The aim of this chapter is to describe the GYPSIE graphical protocol specification environment. GYPSIE is a pivotal component of the SPEAR II Framework and is used to design and model security protocols. Two core advantages of GYPSIE are its simplicity and generalizability. The primary focus of the GYPSIE environment is rapid, effective and accurate construction of a protocol model, with the other components of the SPEAR II Framework being used to carry out analysis and code generation activities based on the information specified in this model. We will begin this chapter by developing a set of requirements which we believe a flexible and generic security protocol modelling environment such as GYPSIE should implement. Thereafter we will describe the core features of GYPSIE, followed by a discussion of the message rounds calculator which has been created to work in conjunction with GYPSIE. User experiments which we conducted are then described. Implementation details will also be elaborated on briefly. The chapter concludes with closing remarks that draw together all of the issues discussed in these sections.

3.1 Requirements for a Security Protocol Design Environment

Any tool that is concerned with design ing and engineering cryptographic protocols should provide a design module that facilitates the rapid and accurate specification of a protocol, but at the same time is flexible enough to accommodate new types of protocols and security methods. The design environment should also be tailored to provide a degree of guidance as a protocol engineer constructs a protocol model. Guidance can be provided by enforcing the order in which protocol construction takes place or by providing an interface that clearly displays the current state of the protocol specification. For example, we do not want an individual to be able to specify messages if no principals have been defined. Also, if one component is used in multiple messages, any changes made to one of these should be reflected in the others in order to maintain consistency. To present a clean interface and to avoid unnecessary complexity, different views of a protocol specification can be used. These views can represent completely distinct parts of a protocol specification, or they can overlap to some extent.

When creating a protocol specification, the first components that need to be defined are the principals who partake in each protocol session and exchange messages with each other. Each principal defined in the protocol essentially represents a role that an entity will fulfil while the protocol is in effect. A given role can be enacted by different entities each time a protocol session is run. After the principals have been defined, the messages that are to be transferred must be constructed. Each message requires a sender and at least one receiver. Besides specifying the sender and receivers, the components used within each message must also be defined. Some standard components used in cryptographic protocols include nonces, timestamps, encryptions and hashes. Since components such as encryptions and hashes contain further components, we essentially end up with a component hierarchy that can be represented as a tree-like structure to aid in the visualization of a message.

During the construction of a protocol specification, a designer may want to isolate functionality into subprotocols to simplify a design or to improve the modularity of the implementation. For this reason, the ability to add, view and modify subprotocols should be provided by a design environment. *Automatic* subprotocols are always executed, while *conditional* subprotocols are only executed if certain criteria are fulfilled during a session. The protocol specification environment should differentiate between these two subprotocol classes when a specification is displayed. Besides being able to view subprotocols, it is also important to be able to visualize their hierarchical relationship relative to each other. It should also be possible to copy components, messages and principals between subprotocols. However, many formal analysis techniques are not compatible with specifications that employ subprotocols. For this reason, the ability to 'flatten' a subprotocol hierarchy should be included in the design environment.

Once the essential features of a protocol specification have been defined, additional information that can be used for security and performance analysis, meta-execution or code generation can be supplied. The interfaces to supply this information would be provided by the design environment itself or by the engineering modules that plug into the system. These secondary modelling steps could include carrying out the following tasks:

- The underlying structure of components within each message can be clearly defined to enable source code generation, simulation or formal analysis.
- External functions that are to be applied to the message components may be defined. The linkage between these external functions and generated source code must also be made explicit.
- Communication settings, such as transport protocols and instance timeouts, can also be specified.
- Information specific to security or performance analysis may be declared. This could include details such as initial beliefs and possessions of principals and pre-recorded protocol timings.
- Source code or checks that are to be executed while the protocol messages are being transmitted can be embedded in the appropriate locations.

There are obviously a large number of other details that can be specified. But for now, this list gives a general idea of how a design environment can be used in the context of multi-dimensional protocol engineering. Essentially, the design environment can be viewed as the central interface through which a protocol specification is created. After construction, this specification is used when conducting the actual engineering functions which examine the protocol and generate source code. In effect, all of the protocol engineering modules rely on the specification produced by the design module. Thus, the goal of a design module should be to provide an easy-to-use, flexible and powerful interface within which security protocols can be accurately modelled and exported for later use.

3.2 Overview of the GYPSIE Environment

The GYPSIE environment is an integral component of the SPEAR II Framework and is used to construct a protocol specification before any protocol engineering or analysis operations are conducted. GYPSIE itself is divided into three views, each view revealing information about a specific portion of a protocol specification. The *High-Level Protocol View* describes the overall flow of messages, graphically indicating the principals which send and receive messages and the sequence in which these messages are transmitted. Any subprotocols embedded within the protocol under inspection are also displayed. The more detailed *Component View* displays the contents of each message as a hierarchical tree and provides mechanisms whereby each component within this tree can be edited, ordered, deleted or viewed. The *Navigator View* presents a user with a concise summary of the structure and contents of a protocol by using a tree-view with expandable and collapsible nodes. Through the use of this tree-view, the navigator also facilitates the exchange of messages and principals between subprotocols. Besides allowing a protocol to be designed in these three views, GYPSIE also ensures that a protocol can be saved to disk, loaded from disk and exported to either text or LaTeX. An included component tracker can be used to highlight a given item embedded within a message so that its use can be tracked through the protocol model. To determine optimal or synchronous message rounds for use in simple performance measurements [34], a rounds calculator which is part of the SPEAR II Framework can be used.

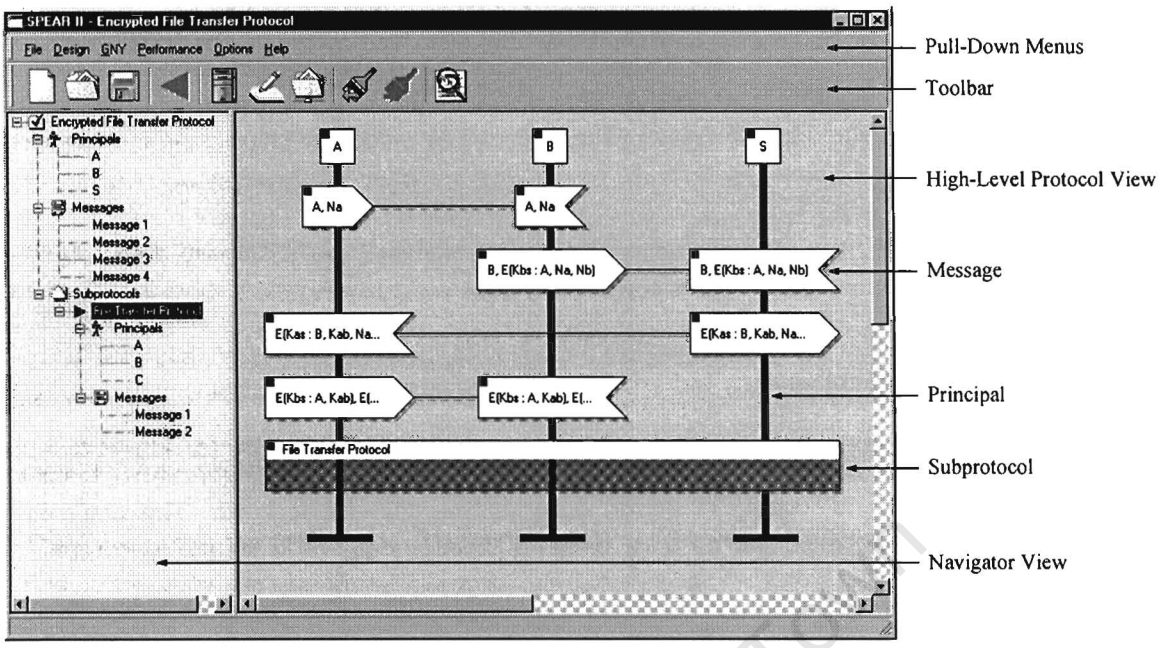


Figure 3.1: The GYPSIE protocol design environment shown in a SPEAR II screenshot.

3.2.1 High-Level Protocol View

The High-Level Protocol View, shown in Figure 3.1 can be considered as a front-end to the GYPSIE protocol design module and essentially consists of a dynamically-sized canvas on which a protocol engineer places objects representing principals, messages and subprotocols. Its main purpose is to provide a suitable abstraction and encapsulation mechanism so that designers can retain a high-level perspective of the operation of the protocol. Manipulation of the High-Level View canvas components takes place through dragging-and-dropping operations, pop-up menus and the SPEAR II pull-down menus. To aid in protocol construction, the High-Level View incorporates an undo and redo feature which ensures that designers can recover from accidental message, subprotocol and principal moves, deletions and edits. The Component Tracker is also part of the High-Level View and attempts to reduce the complexity that is found in specifications containing a large number of components.

3.2.1.1 Selection of a Formalism

The Message Sequence Chart (MSC) [44] syntax forms the basis for the hybrid representation that is used within the High-Level View to describe the flow of messages. In essence, the High-Level View formalism can be described as a simplified MSC syntax which makes use of symbols from the Specification and Description Language (SDL) [43] to denote the sender and receiver of a message. The use of these SDL symbols in conjunction with the sequencing and tracing ability of MSCs leads to a clean, accurate and concise representation. An advantage of using the SDL symbols to represent messages is that we are able to include a flat list-like representation of a message's contents within each one, since each of these symbols is a box-like structure that can act as a text container. The rectangular symbol used to represent subprotocols does not stem from any existing formal technique. It was derived well after the message and principal representations and its development was primarily influenced by the need to integrate with these formalisms and to maintain compatibility with the underlying GYPSIE architecture.

When deciding on a formalism for representing a security protocol specification, the use of SDL was initially considered. SDL is a specification language used to describe communicating systems such as telecommunications protocols. It is an International Telecommunication Union standard, and as such is widely recognized and supported. The SDL language includes both a textual syntax (SDL/PR) and a graphical syntax (SDL/GR). In this regard, SDL is superior to other formal description techniques such as Estelle [18] and LOTOS [10], which have only textual representations. There are also existing tools such as editors, correctness analyzers, simulators and code-generators that exist for specifications written in SDL. Members in our research group have also developed a system called SPECS II [19] which performs correctness and performance analysis of an SDL protocol specification and generates code from this input. Other projects undertaken in our research group have also made extensive use of SDL, and thus it was an obvious consideration.

However, after due deliberation we decided against using SDL as our input formalism. Our reason for not selecting SDL as our specification formalism was largely due to the fact that SDL does not allow for a high-level cryptographic specification, but rather provides a low-level functional specification. While recognizing that it is at the functional level that subtle flaws can be identified, we perceive a lot of common low level functionality amongst cryptographic protocols. By this we mean that although message recipients and contents vary greatly in security protocols, much of the actual implementation details such as connections and time-outs can be viewed as common to many protocol implementations. Accepting this, one can consider the High-Level View as being a level above SDL — where designers can concentrate chiefly on logical security protocol design and actual message contents.

With a view to allowing prototyping and experimentation, it was decided to use a protocol specification technique closer to that of MSCs, since MSCs capture the exchange of messages at a higher level which is more appropriate to cryptographic protocol design. More intricate details that are necessary for source code generation, or different forms of analysis, can be included in further interfaces which work in tandem with the GYPSIE environment. In fact, we want to enable a protocol engineer to model a protocol as quickly as possible by supplying only those details which are mandatory for understanding the protocol's operation. Our simplified MSC syntax achieves this goal as only the protocol messages, senders, receivers and subprotocols are specified, resulting in a simple irreducible specification. An implication of our decision to use a syntax closer to MSCs is that SDL could still be generated from the specification for use by another modelling tool.

3.2.1.2 Fundamental Building Blocks

Components used in the High-Level View formalism which we have derived are illustrated in Figure 3.2. Communicating *principals* are specified as MSC-style axes, with the head containing the principal's name. Each *message* in the protocol is represented by two linked SDL-style boxes. The sending and receiving principals are designated through the placement of these message boxes — a convex box indicating the sender of a message, and a concave box indicating the recipient. At this point in time, each message has only one recipient, but this representation can easily be extended. Messages are ordered sequentially in time, with the earlier messages at the top of a principal axis, and the later messages near the bottom of the axis. Each message displays a textual representation of the components which it contains. Functionally related groups of messages, known as *subprotocols*, can also be placed on the canvas. A subprotocol is rendered as a rectangular box with the name of the protocol in the title bar. Subprotocols are ordered sequentially, with the subprotocols called first being placed towards the top of the design canvas. Subprotocol components lie on top of the principal axes and those principals which are involved in a given subprotocol are visible through the subprotocol body, as seen in Figure 3.1.

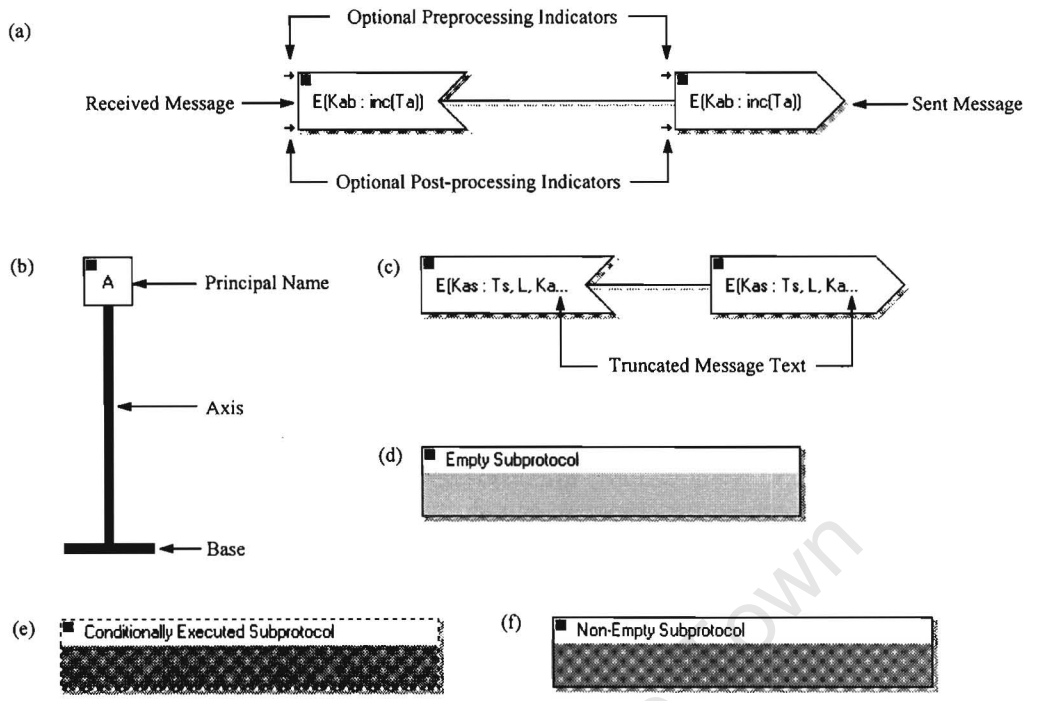


Figure 3.2: Components used to represent a protocol in the High-Level View.

To assist code generation and analysis modules that will plug into the SPEAR II Framework, we have augmented the message representation so that it is able to indicate whether additional processing code has been inserted into the protocol specification. These optional processing indicators are shown in Figure 3.2 (a). An arrow near the top left of a message indicates that preprocessing code has been inserted by the protocol engineer, while an arrow near the bottom left indicates that post-processing code has been inserted. Preprocessing code is executed before a message is sent or received, while post-processing code is executed after a message has been sent or received. If a subprotocol does not contain any principals or messages, then its fill colour is the same as the canvas background, as shown in Figure 3.2 (d). On the other hand, if any principals or messages are present, the subprotocol contains a mesh pattern, illustrated in Figure 3.2 (e) – (f). A subprotocol that is always executed has a solid outline, while one that is executed conditionally has a dashed outline.

The High-Level View can be fully manipulated through drag-and-drop operations. A message or subprotocol can be reordered in time by dragging it up or down along an axis. The sender and receiver of a message can also be changed by dragging either the convex or the concave message box onto a new principal axis to signify the new sender or receiver respectively. Principal axes can also be repositioned to 'neaten' or simplify the appearance of the specification. Note that movement of the principal axes does not change the functioning of the protocol in any respect. When dragging a component, an XORed representation is displayed. This representation appears green when a drop is allowed, and red when it is not. The XORed representations also contain guidelines to help users to accurately position the components. When dragging a concave or convex message object, a drop will only be valid if the guide, which is shaped as a cross, is positioned on top of a principal axis between messages and subprotocols. A subprotocol drop is considered as valid if the guide, which is basically a horizontal line, is not on top of a message or subprotocol. Finally, a principal drop is allowed if the principal axis does not overlap the horizontal space of any message or principal.

If a message contains a large number of embedded components, then its textual representation could be rather long with the result that its corresponding representation in the High-Level View could span quite far across the canvas or even scroll out of the visible window area. Such a situation could make viewing the protocol tedious and would also affect dragging and dropping operations by making it difficult to position the message over another principal axis. For this reason, a protocol engineer is allowed to set the maximum allowed message box width in the SPEAR II Preferences Dialog. Any message box exceeding this size is truncated as shown in Figure 3.2(c). However, to allow a designer to identify a message or view its contents, tooltips have been added to the message boxes so that the message contents are displayed when hovering the mouse pointer over each message. A subprotocol automatically occupies the width of the visible canvas area if no principals have been defined. If principals are present, then the subprotocol's width is roughly equivalent to the width of the protocol specification. A truncated version of the subprotocol name is displayed if it is too long for the rectangular box in which it appears. Tooltips are also used to reveal the full name of a subprotocol to cater for situations in which the name has been truncated.

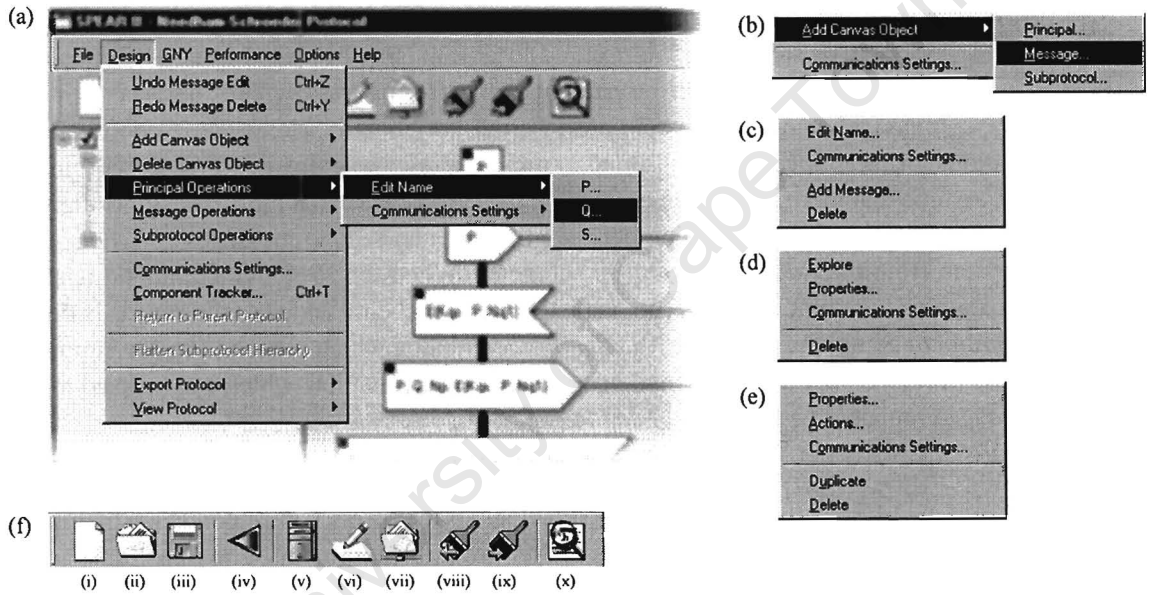


Figure 3.3: Graphical components used to work in the High-Level View.

3.2.1.3 Accessing High-Level View Features

The features provided by the High-Level View are all accessible through the use of pull-down and pop-up menus. Shortcuts and toolbar buttons are provided for accessing frequently used operations, while direct-manipulation is used to modify the sequencing and transmission details related to protocol messages and subprotocols. We have tried to be as logical and as flexible as possible in the construction of the interface so as to cater for a wide range of users and preferences. This goal has lead to us developing multiple ways to accomplish frequently used High-Level View operations. In Figure 3.3 the four pop-up menus used to interact with a protocol model, the *Design* pull-down menu and the toolbar buttons used for accessing critical features of the SPEAR II Framework are all illustrated. In the list that follows we describe each of these graphical components and the role which they fulfil in providing functionality, usability and flexibility to the High-Level View:

(a) The *Design* pull-down menu provides access to all of the High-Level View features. Whereas the pop-up menus are all limited to dealing only with principal, message or subprotocol operations, the *Design* pull-down menu facilitates all of these operations and more. In Figure 3.3 (a) we see the *Principal Operations* submenu being used. Notice that when selecting the *Edit Name* submenu, a list of principals is constructed so that the user can indicate which principal he wishes to rename. The principals inserted into this submenu are those who take part in the protocol currently being edited. An advantage of this dynamically-constructed menu system is that it provides an alternative to right-clicking on a principal's canvas representation and then using the principal pop-up menu illustrated in Figure 3.3 (c) to select an operation. The principle of dynamically updating the submenus found in a pull-down menu with the names of the appropriate target objects is also used in the *Delete Canvas Object*, *Message Operations* and *Subprotocol Operations* submenus. Undo and redo functionality is provided by the first two pull-down menu items, as well as the toolbar buttons labelled as (viii) and (ix) respectively.

(b) This pop-up menu is used to add a principal, message or subprotocol object to the High-Level View canvas. It appears when right-clicking on an open portion of the canvas not currently occupied by a principal, message or subprotocol object. The advantage of using this pop-up menu to add items to the protocol model is that the resultant object is inserted where the user right-clicked the canvas. Messages and subprotocols are inserted horizontally along the Y-coordinate of the mouse pointer, while principals are inserted vertically along the corresponding X-coordinate. For example, assume that there are three messages on the canvas. To add a fourth message after the first one, a user merely needs to right-click between the first and second messages and then select the *Message* item from the *Add Canvas Object* submenu. When the *Principal* command is chosen, a dialog box which the user can employ to name the principal being added to the canvas is displayed. The *Subprotocol* command opens the *Add Subprotocol* dialog which is used to set the subprotocol's properties, while the *Message* command opens the Component View which can be used to edit the new message components.

(c) This pop-up menu is invoked when right-clicking on a principal object. Once the pop-up menu appears, a dashed box is drawn around the principal so that users can remember which principal they selected to be the subject of commands chosen from the pop-menu. The *Edit Name* command activates a dialog containing a text box which the user can employ to rename the principal. The *Add Message* item invokes the Component View and then sets the selected principal to be both the sender and the receiver of the newly created message. The user must obviously change the sender or the receiver of the message to reflect another principal on the canvas. To delete a principal, the *Delete* command is used. This command erases the selected principal, as well as all of the messages that it sends and receives, from the specification.

(d) This pop-up menu appears when right-clicking on a subprotocol object. A dashed line is drawn around the subprotocol once the pop-up menu becomes visible. As discussed in (c), this high-lighting effect helps a user to remember which subprotocol he is currently editing. The *Explore* command is the most important item on this pop-up menu and will probably be the most frequently used in protocol specifications involving subprotocol hierarchies. This command brings the selected subprotocol to the foreground, causing the current High-Level View canvas to be redrawn and replaced with the subprotocol's own principals, messages and subprotocols. To return to the parent protocol, the back button located on the toolbar, and labelled as (iv) in Figure 3.3, is used. The *Properties* command invokes a dialog which can be used to modify a subprotocol's properties. Properties that can be changed at present include the name, execution style and se-

quencing details. The *Delete* command is used to remove a subprotocol from the specification. If a subprotocol is removed, then all of the components defined therein are also deleted, unless they appear in other non-related subprotocols.

(e) This pop-up menu is brought to the foreground when right-clicking on a message object. To help users remember which message was right-clicked, a dashed line is drawn around the message while the pop-up menu and any commands spawned therefrom are active. The *Properties* item activates the Component View which a user can then use to edit the components embedded within the message. The *Actions* command is used to specify processing operations that take place before or after sent and received messages. At present it merely displays a dialog box with two tabbed panes for specifying the processing actions for the sender and receiver of the message respectively. To duplicate a message a designer can make use of the *Duplicate* command. This command produces a copy of the selected message which can then be altered as needed. This feature is useful for modelling backward and forward replay attacks. The *Delete* command is used to remove a message from the specification.

(f) The toolbar is used to provide fast access to key features of the SPEAR II Framework. Buttons (i) to (iii) are used to create a new specification, open a specification, and save a specification respectively. When creating a new specification or opening one that has been saved, the user is first asked whether she wishes to save the current protocol if it has been modified since the last save. Returning to a parent protocol after entering a child is accomplished by pressing button (iv). Of course, this function will only be applicable if the specification in the High-Level View is a subprotocol with an existing parent. To create a new principal, message or subprotocol, the buttons labelled (v) through to (vii) are used. These buttons open the dialog that is invoked when selecting the *Properties* item from the message and subprotocol pop-up menus, and the *Edit Name* item from the principal pop-up menu. Undo and redo functionality exists for all of the High-Level View commands and is accessible by pressing buttons (viii) and (ix) for undo and redo respectively. The component Tracker is invoked by pressing button (x). Tooltips that appear when hovering the mouse cursor over a toolbar button describe what each button does.

The *Communications Settings* command, found on the *Design* pull-down menu and each of the High-Level View pop-up menus, is used to invoke a dialog which protocol designers can employ to supply technical information necessary for code generation or performance analysis. This information could include details such as IP addresses, transport protocols, timing information and message sizes. The *Communications Settings* commands found in the *Design* pull-down menu and the canvas pop-up menu are both used to modify the default settings of the root protocol and the current subprotocol. To modify the communications settings of a principal, subprotocol or message, the *Communications Settings* commands found on the principal, subprotocol and message pop-up menus are used.

If a given command is not available, then the graphical components used to access it are dimmed and rendered inactive. For example, if two or more principals have not been defined, then all of the pull-down and pop-up menu items, as well as toolbar buttons, used to add messages are disabled. The graphical components become usable once the commands become available. This approach helps to avoid tedious, intrusive and unnecessary error message dialogs. We have chosen to dim disabled components as opposed to hiding them so that users always maintain an idea of the tasks that are possible to carry out with the interface. However, the disabling of redundant commands does not indicate to users how to use the interface. Instead, it is still the responsibility of individual protocol engineers to ensure the presence of appropriate objects on the canvas and to supply information necessary for any engineering operations.

The direct manipulation built into the High-Level View helps protocol engineers to easily set up the sequencing of messages and subprotocols, and also to specify the sender and the receiver of a given message. However, the current protocol view may not always fit into the visible window. This may hinder dragging-and-dropping operations which change a message's sender and receiver if the target principal does not reside in the visible window area, as automatic window scrolling has not been implemented. In such a situation, a designer can still use the Component View to change the sender and receiver. The same problem could apply when re-sequencing messages and subprotocols if there are already a large number in the specification. In this case, the Component View and Subprotocol Properties Dialogs both allow one to set the position of a message or subprotocol by using a vertically-orientated slider control.

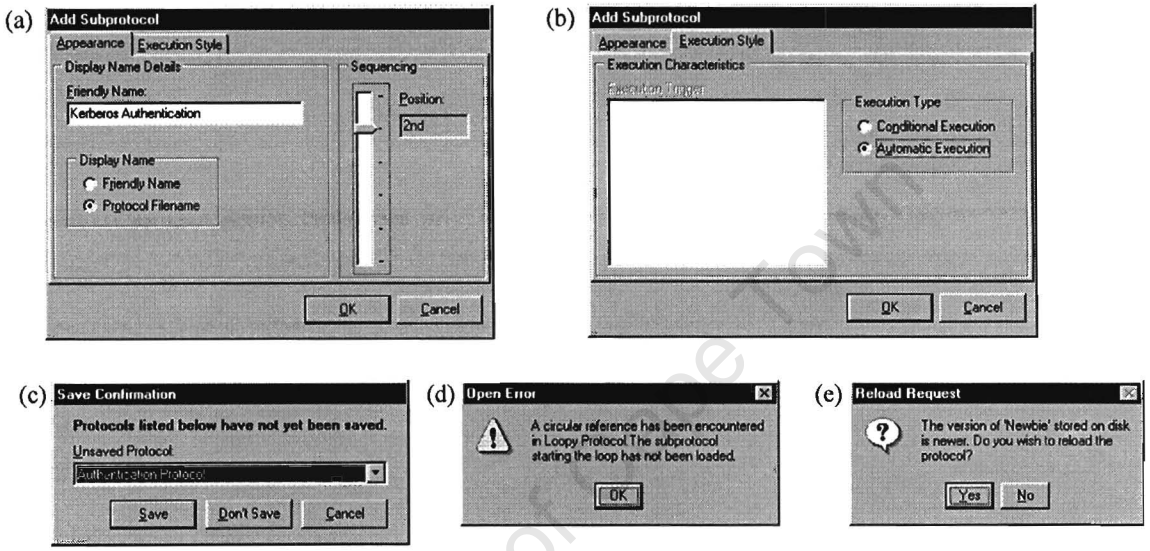


Figure 3.4: Dialog boxes used in conjunction with subprotocols.

3.2.1.4 Working with Specifications in the High-Level View

When creating a protocol specification, a user first begins by specifying at least two principals who send or receive messages. Once these two principals have been defined, messages can be added and more principals can be introduced as needed. The addition of subprotocols to a specification does not require the prior existence of any other canvas objects. Thus, a given protocol can be made up entirely of subprotocols, just like some functions in a procedural language might only consist of calls to other functions. When adding a principal, the user is first requested to supply a name that will be used to identify the principal during modelling and engineering operations. Once this name is supplied, a principal axis is created and placed on the canvas. The same dialog used for a principal add operation is used in principal edit operations. When adding messages, the Component View is invoked so that the sender, receiver and embedded components within the message can all be defined. A message object is added to the canvas when the Component View is closed. The Component View is also used to edit message properties. The vast majority of protocols will not contain any subprotocols. However, in the case of specifications that do contain subprotocols, each subprotocol can essentially be viewed as a High-Level View model that can be brought into focus as necessary to edit the messages and principals contained therein. At present, the only details that are required when specifying a subprotocol are its name, execution style and position in the specification. Principals and messages are added when exploring the subprotocol.

To add subprotocols to a specification, the *Add Subprotocol* dialog shown in Figure 3.4 (a) – (b) is used. This dialog consists of two tabbed panes. In the *Appearance* pane a designer can set the vertical position of a subprotocol as well as the name that is displayed in its rectangular representation on the design canvas. There are two choices for this display name — a *Friendly Name* or the protocol's file-name. We have distinguished between these two names to allow individuals to use a simple name when saving the protocol, and a more descriptive name in the protocol specification. The *Execution Style* pane is used to set the execution type used for later code generation or analysis. If a subprotocol is always executed, then the *Automatic Execution* radio-button is checked. However, if the execution depends on the parent protocol's state, then the *Conditional Execution* radio-button must be checked. The text-box on the left of the pane is used to specify the execution trigger for a conditionally executed subprotocol. At this stage, no code generation routines have been added to the SPEAR II Framework, so the trigger is not functional but merely present to give an indication of how the system would operate. When editing a subprotocol, the same dialog box is used, however the title is changed to reflect the editing operation.

Whenever a protocol specification is modified, its dirty bit is set and the save function is enabled. When a save command is issued, only the protocol contained within the High-Level View is written to disk and the information contained in any embedded subprotocols is ignored. Each protocol is saved in a separate file. Protocols containing further subprotocols store the properties associated with each subprotocol as well as the relative path to the subprotocol file on disk. To save an entire protocol specification the *Save All* command, which is accessible from the *File* menu, is used. If a protocol is being saved for the first time, the user will be asked to supply a filename. When closing the High-Level View, a list which contains a reference to every unsaved protocol specification is automatically created. This list is then used to initialize the *Save Confirmation* dialog, shown in Figure 3.4 (c). The drop-down list in this dialog contains the names of all the unsaved protocols. Using this dialog a user can decide whether to save or discard the changes made to a protocol in the list. A protocol is removed from the drop-down list when it is saved or its modifications discarded. Once a decision has been made regarding the status of every protocol in the drop-down list, the *Save Confirmation* dialog is closed.

Two or more subprotocols in a specification cannot have the same name unless they are all tied to the same source file. However, if such a scenario is the case, the source file can be modified and saved while one of these duplicate subprotocols is being edited in the High-Level View. Changes made to the source file in this way would not be reflected in any of the other subprotocols unless a check was explicitly carried out to determine whether the file on disk had changed. Thus, in order to avoid synchronization issues of this nature between multiply included subprotocols, the High-Level View subsystem conducts a test before exploring a subprotocol to ensure that the source file on disk has not been modified since the subprotocol was last edited. If the subprotocol source has been modified, the warning message shown in Figure 3.4 (e) is displayed and the user is given the option of reloading the subprotocol specification. This feature is useful when dealing with subprotocols that carry out error handling, since a given set of error handling messages could be packaged in a subprotocol, included in multiple locations and then edited and saved from wherever the container subprotocol appears in the specification.

Loading a protocol specification containing subprotocols is essentially a recursive process, since the same load operation is executed on the files which contain the subprotocols embedded within a parent. A subprotocol is only loaded if the file containing the subprotocol exists in the location defined within a saved specification. An inherent danger when loading protocol specifications containing subprotocols is that an infinite loop could result from a circular reference. To avoid infinite loops, all of the protocol filenames in a subprotocol's ancestor list are examined to determine whether the protocol file has already been loaded. If it has been loaded, then the error message shown in Figure 3.4 (d) will be displayed and the loading operation will ignore the subprotocol in question.

3.2.1.5 Dealing with Duplicate Components in a Specification

The memory model used by GYPSIE ensures that only one copy of a component ever exists in memory at any given time. For example, in a protocol specification containing the components $\{X\}_K$, X and $H(X)$, there would only be one copy of the data structure defining the characteristics of X. All of the instances of X throughout the protocol specification would reference the memory location containing this data structure. In this way changes made to a component remain consistent and are immediately reflected throughout an entire protocol specification. Guards built into the GYPSIE environment prevent an individual from creating two terminal components with the same properties. To duplicate a terminal component, only the functions provided by the Component and High-Level Views can be used, thus ensuring that all duplicates remain instances of the same component.

A given component might be duplicated in a protocol or even exist in two separate subprotocols that are both part of the same protocol specification. When reloading such a specification after it has been saved, we would like all of these duplicates to again reference a single memory location that contains the data structures pertaining to the component of which they are all instances. To achieve this goal we assign a unique numerical identifier to each component when writing it to disk. Duplicates of a given component will all be assigned the same identifier. The identifier essentially tags each component in the protocol source file and in this way maintains the linkage between components that are duplicated across subprotocols and saved in separate source files. With this component tagging in place, reloading a given component merely involves restoring the component's data structures once and then ensuring that all of the remaining components with the same numerical identifier reference this information when they are loaded from the same or other source files.

Two protocols which are not part of the same specification may occasionally contain terminal components with identical data definitions. Since these terminal components were never duplicates of the same component instance, they would have received different numerical identifiers while being saved to disk and thus will not be forced to reference the same memory location when they are included in the same protocol specification as subprotocols. In such a situation, confusion and naming conflicts would most probably occur. To eliminate the possibility of such a scenario, we examine every terminal component during a subprotocol load operation and then determine whether a similar component already exists. If it does, we modify the name of the terminal component that is currently being loaded. For example, if protocol X containing the nonce N_a is loaded into a specification that already contains a nonce by that name, then the nonce in protocol X is renamed to N'_a. Generating an error message for each duplication would be tedious from an end-user perspective, and besides this fact, we cannot assume that if two components have the same data definition they are meant to be one and the same when merging two protocols.

3.2.1.6 Flattening a Protocol Specification

In certain situations, an analysis system tied to the SPEAR II Framework might not be able to accommodate or understand subprotocol hierarchies. For this reason, the ability to transform a collection of subprotocols into a single set of messages embedded within one root protocol is provided by the GYPSIE environment. This 'flattening' process essentially extracts all of the messages from a given subprotocol and then embeds them in its parent. The flattening procedure can work out to be a recursive process, since every subprotocol can potentially contain further nested subprotocols that will need to be flattened before the subprotocol itself is incorporated into its parent. Once all of the messages have been extracted from a subprotocol, the subprotocol is deleted and the messages take its place on the design canvas. In

the case of conditionally executed subprotocols, a protocol engineer should ensure that only those messages applicable to the protocol scenario under examination are used. This essentially means that he would have to examine all of the extracted messages and then delete those which are not required for the engineering routines being performed. Messages belonging to automatically executed subprotocols can be embedded in a parent protocol immediately and don't require any tweaking by a protocol engineer.

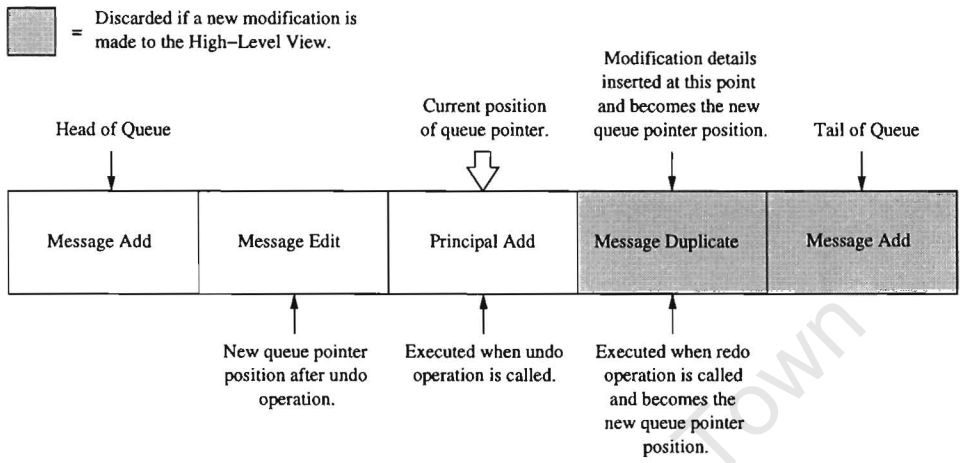


Figure 3.5: An explanation of how the history queue is used for undo and redo operations.

3.2.1.7 Undo and Redo Operations

A dynamically-sized queue containing the undo and redo details pertaining to every High-Level View operation carried out during an engineering session is used to facilitate undo and redo commands. Associated with this queue is a pointer which references the queue element containing the details of the next High-Level View operation to be undone. The number of High-Level View operations stored in the history queue can be set using the SPEAR II Preferences Dialog. When the queue is full, the oldest operations at the head are discarded first. An advantage of the undo and redo feature is that it encourages individuals to experiment with a protocol specification, as they can easily recover from any changes or errors that have been introduced. The ability to undo and redo specification changes is also useful in protocol analysis and code generation tasks, as components can be modified before carrying out analysis or code generation, and then restored to their original state thereafter, thus allowing a user to experiment with a variety of protocol configurations without having to save each one.

The operation of the history queue is relatively straight-forward and is illustrated in Figure 3.5. Every time a modification which affects a protocol's High-Level View is made, the details necessary to undo or redo this modification are recorded and inserted behind the element referenced by the queue pointer. The queue pointer is then reset to reference the newly inserted element and any elements that previously existed behind the pointer are discarded. Thus, older operations will be found at the head of the queue, while newer operations will be found at the tail. When performing an undo operation, the information stored at the current queue pointer position is first used to restore the protocol specification, after which the pointer is moved one element forward, towards the head of the queue. When performing a redo operation, the queue pointer is first moved backwards by one position, towards the tail of the queue, and the information at this new position is then used to remodify the protocol specification.

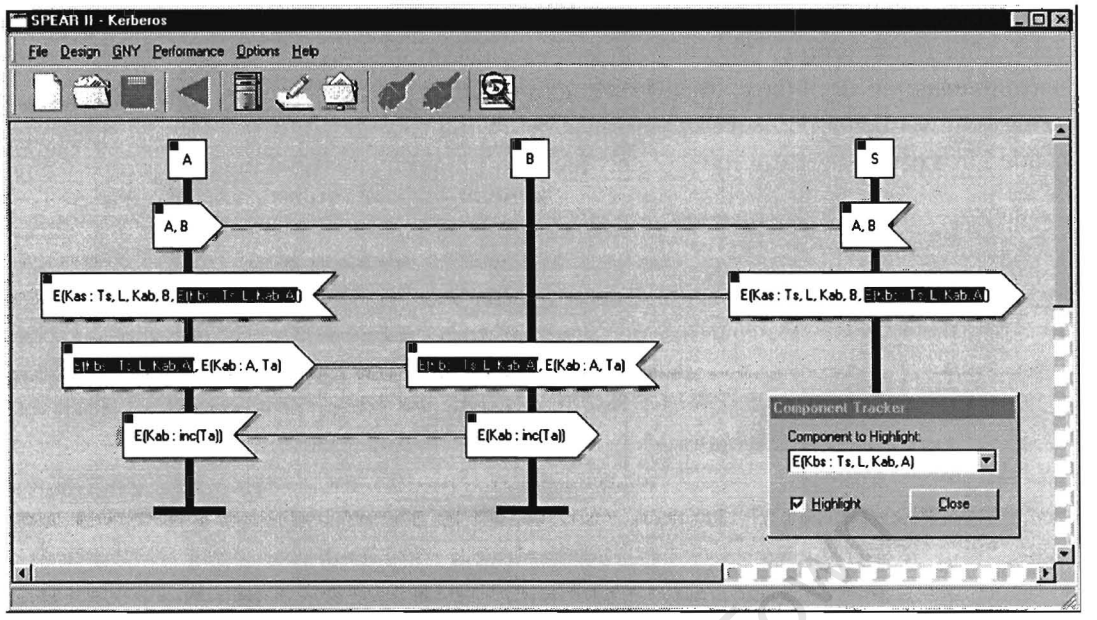


Figure 3.6: The Component Tracker being used to highlight a Kerberos ticket.

3.2.1.8 Highlighting the Position of Components

The Component Tracker helps designers to deal with the complexity that results from having a large number of components defined in a message passing specification by allowing them to highlight all of the locations where a given component appears on the design canvas. As a result, they can easily locate a component in the protocol model and also track its movement through the protocol specification. The ability to track a component is useful for educational and informative demonstrations of a protocol's underlying operation and also helps one to focus more on the critical issues associated with a design, such as where and how a specific component is being used. In Figure 3.6 we see the Component Tracker being used alongside a simplified version of the Kerberos protocol. In this screenshot the Component Tracker dialog in the lower right-hand corner has been used to highlight the ticket $\{T_s, L, K_{ab}, A\}_{K_{bs}}$ in the second and third messages. Because of the highlighting it is significantly easier to follow the progress of the ticket from the authentication server S to the client A and then to the server B.

As shown in Figure 3.6, the Component Tracker dialog is a modeless window that contains a drop-down list of all the components specified in a protocol definition. Only components that are part of the currently visible protocol specification appear in this drop-down list. Those belonging exclusively to child or parent subprotocols are excluded. To highlight a given component, the user merely needs to select it from the drop-down list and then place a check mark in the *Highlight* check-box. The foreground and background colours used to highlight a component are user-definable and are set using the SPEAR II Preference Dialog. While the Component Tracker is active, a user can still scroll through the High-Level View canvas and explore any existing subprotocols. A component will only be highlighted if it is contained within the current High-Level View message passing specification. If the component is not visible because it is part of a truncated message, then it will not be highlighted. However, if a portion of the text representing the component is visible, then that specific portion will be highlighted.

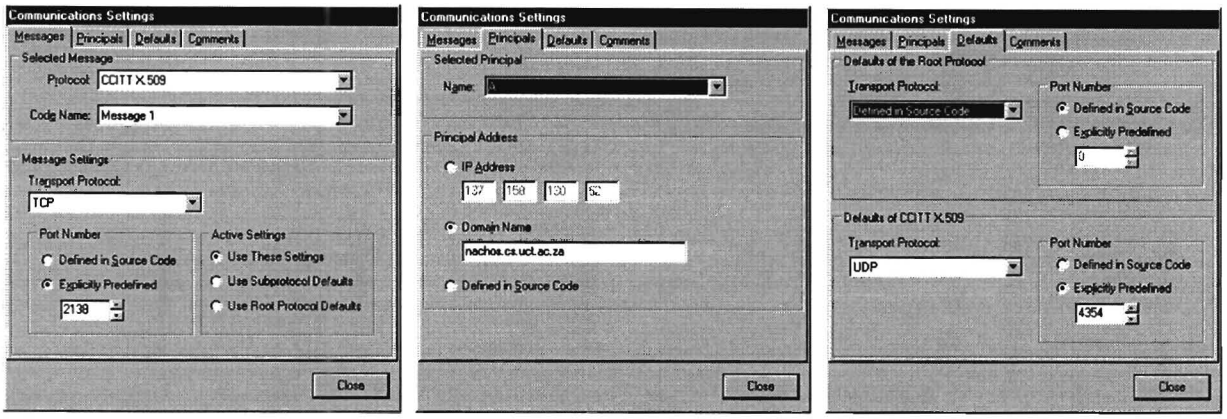


Figure 3.7: The tabbed-pane dialog used to configure communications settings.

3.2.1.9 Setting Protocol Properties

The ability to configure the communications settings of a protocol specification is essential for code generation and selected performance analysis routines. For this reason we felt that it would be useful to prototype an approach for incorporating the collection of these details within the SPEAR II Framework. The *Communications Settings* dialog, shown in Figure 3.7, consists of four tabbed panes which are used to specify communications settings for messages and principals, as well as default communications settings for protocols. Because of the prototypical nature of this dialog, the input fields do not cover all of the details that would be required for most code generation or performance analysis routines.

A list of all the protocols used in a specification is contained in the *Protocol* drop-down list of the *Messages* pane. After selecting a protocol from this list, the *Code Name* drop-down list is initialized with the names of all the messages in the selected protocol. The properties of a message that can currently be modified are the transport protocol and port number. These two settings can either be defined explicitly or specified manually in any generated source code after it has been created. A protocol engineer can apply the settings specified in this pane to the currently selected message. Alternatively, he can associate the subprotocol defaults or the root protocol defaults with this message.

The *Principal* pane allows a protocol engineer to define the communications settings associated with a given principal role. A list of all the principal names defined in the specification is contained in the *Principal* drop-down list. A principal's address can be specified as an four-byte IP address, a domain name or it can be relegated to being specified in any generated source code. If a principal changes its name from A to B, it obtains the settings that the B role defines. Furthermore, if principal A is duplicated in the protocol specification, then changing the settings of one of these duplicates will also update A and all of its duplicates to maintain consistency within the protocol specification.

The *Defaults* pane is used to specify the default message settings of the root protocol in a hierarchical specification, as well as the default settings of each of the descendant subprotocols. These settings can then be associated with individual messages in the *Message* pane. Using this approach allows us to keep the settings for a group of messages in a protocol or subprotocol consistent, eliminating the need to modify each message every time a global change needs to be made. Fields included in the *Defaults* pane must be of such a nature that they can be common to a collection of messages. If the *Message* pane were to include performance measurement fields for transmission timings and message sizes, then these settings would not be included in the *Defaults* pane as they apply to individual messages.

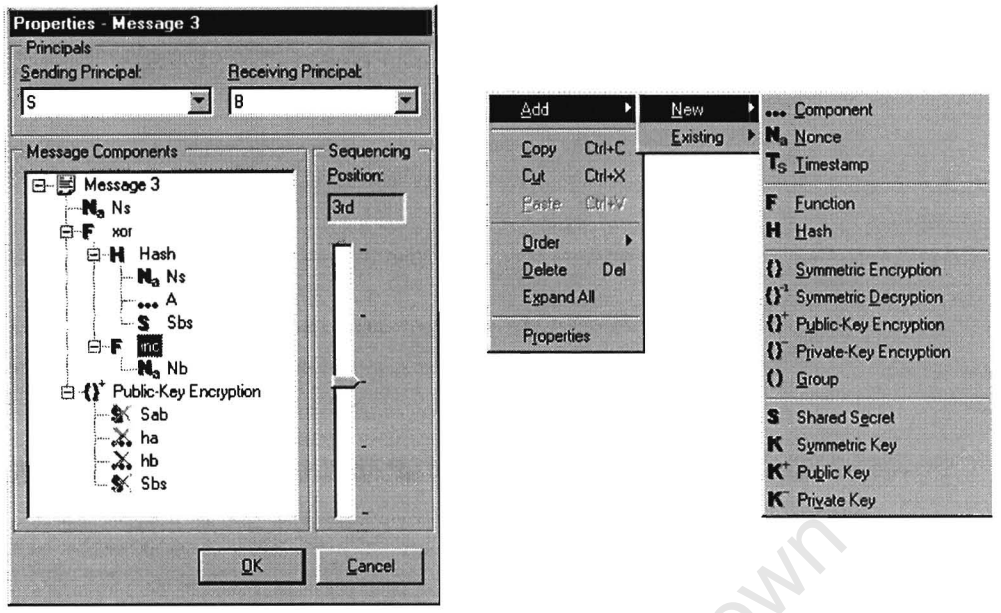


Figure 3.8: The Component View and its associated pop-up menu.

3.2.2 Component View

While the High-Level View focuses on the external environment within which message exchanges take place, the Component View has a much narrower scope, focusing primarily on the composition and internal structure of messages in a protocol. Within the Component View dialog the internal structure of a message is represented through the use of a hierarchical tree structure. This hierarchical tree can be manipulated by using drag-and-drop operations or a pop-up menu associated with the Component View. The sender and the receiver of a message, as well as the position of the message in its container protocol, can also be modified by using the appropriate graphical controls provided within the Component View. The Component View as implemented in the SPEAR II application is shown along with its associated pop-up menu in Figure 3.8. The message displayed as a hierarchical tree is represented in conventional cryptographic notation as $S \longrightarrow B$: N_s, $xor(H(N_s,\ \mathrm{A},\ S_{bs}), inc(N_b))$, $\{S_{ab},\ \mathrm{ha},\ \mathrm{hb},\ S_{bs}\}_{+K_b}$. The public key $+K_b$ is defined as a property of the public-key encryption node and is visible through the use of tooltips which appear when hovering the mouse pointer over the encryption node. A flattened representation of the entire message is displayed via a tooltip when hovering the mouse pointer over the root node.

3.2.2.1 Using the Component View

Each node in the hierarchical tree represents an embedded message component. Now, to be able to represent a message as a hierarchical structure, we must categorize these components into terminal and non-terminal node types. Logically, we can think of cryptographic types such as hashes, encryptions and functions as non-terminal nodes, since they all have to contain child nodes and cannot exist independently of these. Components such as nonces, timestamps and keys would obviously be classified as terminal nodes, since they cannot encapsulate other components but instead exist as independent entities. We have defined thirteen primary component types and categorized them as follows:

- *Non-terminal* components include functions, hashes, symmetric encryptions and decryptions, public-key encryptions, private-key encryptions and groups.
- *Terminal* components include nonces, timestamps, shared secrets, symmetric keys, public keys, private keys and user-defined components.

Components can only be added to a non-terminal node. To add a component, the Component View pop-up menu, shown in Figure 3.8, is used. When right-clicking on a non-terminal node, the *Add* submenu item is enabled. This submenu contains the *New* and *Existing* submenus. When selecting the *New* submenu a list of all the component types which can be inserted as child nodes are displayed. Selecting one of these component types results in a node of the selected type being created and inserted into the message tree as a child of the currently selected node. After insertion, the properties of the newly inserted node can be edited by using the *Properties* command of Component View pop-up menu. There are two ways to add a component that already exists in the specification to a non-terminal node:

- The first approach makes use of the *Existing* submenu nested within the *Add* submenu. The *Existing* submenu contains a list of all the terminal and non-terminal components that have been defined in the current specification. These components are sorted by type and are extracted from all of the protocols within the specification, thus facilitating the sharing of components between subprotocols. Once a component is selected from the pop-up menu, it is inserted as a child of the non-terminal component which was right-clicked. When inserting a non-terminal component, all of the components anchored off it are copied and inserted as well.
- The other technique for copying components involves using the copy, cut and paste facilities provided by the Component View pop-up menu. To select a component, a user merely needs to right-click on it and select *Copy* or *Cut*. Once a component is stored in the clipboard, the *Paste* command is enabled and its execution will result in the stored component being added as a child to the currently highlighted non-terminal node. A single clipboard is used to store all of the messages and subprotocols in a specification, so items copied in one message, can be pasted into another message in any subprotocol. When pasting a non-terminal node, all of the child nodes that were anchored off it are also pasted in the process.

An important point to note is that when a component is copied, we merely insert a reference to the location in memory that holds the data structures associated with the copied component. This approach ensures that all duplicates of a component remain consistent, since all these duplicates point to the same storage location in memory. Whenever a new component is added to a message, it is given a unique identifier. For example, if a nonce is added to a message, it will initially be named N_1. If this name is retained, the next nonce that is added will be called N_2, and so on. When editing the properties of a component, it cannot be given the same name as any other component that exists in the specification, and if an attempt is made to do so, an error message is generated.

The ability to change the order of components within a message is provided by the Component View pop-up menu. The *Order* submenu has both *Move Up* and *Move Down* commands which can be used to accomplish this task. The position of a component can also be modified by using the up and down keyboard arrow keys in combination with a *Shift* key. To move components from one container to another, drag-and-drop operations are used. For example, consider the hierarchical tree shown in Figure 3.8. To move the component *hb* from the *Public-Key Encryption* Node into the *Hash* node, we merely drag the *hb* node and then drop it on top of the *Hash* node. Linearly moving or dragging and dropping a non-terminal node results in all of the components anchored off it being moved as well.

If a component is subordinate to an encrypted node, then its icon in the tree-view has a light-red cross drawn through it. This marking helps individuals to quickly see whether a given component has been enciphered, although it does not give any details regarding the strength or security of the cipher. All non-terminal nodes can be expanded or collapsed. If a node is expanded, then all of its immediate children are revealed. Collapsing hides all of a node's children as well as their descendents from view. The *Expand All* command on the Component View pop-up menu expands all of the descendent nodes of a non-terminal so that all of their children are visible, not just the first level.

To modify the sender and the receiver of a message, the two drop-down lists in the Component View are used. Each list contains the names of all of the principals involved in the current protocol displayed in the High-Level View. The position of a message among the other messages and subprotocols in a specification is modified by using the vertical slider control. The current position of the message is displayed above the slider in a text box. No undo and redo facilities are available in the Component View. If a user makes a mistake or later wishes to undo or redo an operation, he must make use of the undo or redo feature provided by the High-Level View to undo or redo the message add or edit operation facilitated by the Component View.

GYPSIE Type	Icon	GYPSIE Type	Icon
Component	•••	Public-Key Encryption	$\{\}^{+}$
Nonce	N_a	Private-Key Encryption	$\{\}^{-}$
Timestamp	T_S	Group	$\{\}$
Function	F	Shared Secret	S
Hash	H	Symmetric Key	K
Symmetric Encryption	$\{\}$	Public Key	K^{+}
Symmetric Decryption	$\{\}^{-1}$	Private Key	K^{-}

Table 3.1: Icons used for representing cryptographic types in the GYPSIE environment.

3.2.2.2 Fundamental Component View Types

Each of the thirteen components used in the GYPSIE environment has its own iconic depiction, modifiable properties and exportable textual, LaTeX and Prolog representations. The icons used in the hierarchical tree to represent components are shown in Table 3.1. If any of these components is inside an encryption, its icon has a red cross drawn through it. The thirteen Component View types have been partitioned into six classes.

Class A: *Nonces and Timestamps*

Nonces and timestamps are used to determine the freshness of a message. Timestamps record when a message was sent or constructed, while nonces are used in challenge-response exchanges to ensure that a message is not a replay from a previous session. All nonces and timestamps have a mandatory alphanumeric identifier which is used to distinguish them from each other. For example, the nonce N_{ab} has identifier ab, while the timestamp T_1 has identifier 1. A unique integer identifier is automatically associated with each new timestamp or nonce when it is added to a specification. Within a protocol specification, two nonces or two timestamps cannot have the same identifier unless they are duplicates of each other and reference the same data structures stored in memory.

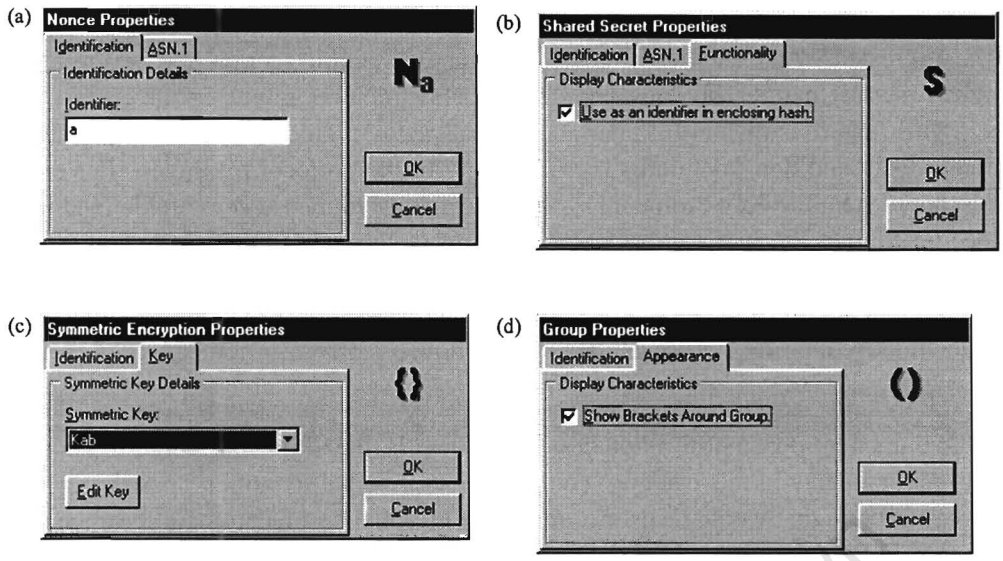


Figure 3.9: Some dialogs used in the Component View.

The dialog used to edit a nonce's properties is shown in Figure 3.9 (a). The *Identification* tab is used to modify the identifier, while, the *ASN.1* tag is used to define the structure of the nonce in Abstract Syntax Notation One [77]. The *ASN.1* tab is merely present for cosmetic purposes, as code generation is not yet supported by the SPEAR II Framework. Any changes made to a nonce in this dialog, will affect all of its duplicates, as the changes are applied to the region in memory storing the nonce's data structures. The timestamp properties dialog appears similar to the nonce properties dialog, and also includes the *Identification* and *ASN.1* tabbed panes.

Class B: *Cryptographic Keys and Secrets*

Shared secrets, symmetric keys, public keys and private keys all fall into this category. A shared secret is kept private between a group of principals and, when necessary, used to identify a message or collection of components as originating from a member of this closed group. For example, when encrypting components with a public key, a shared secret can be included among the encrypted components to identify the principal who enciphered the data. When a shared secret S is used as an identifier, as opposed to being used as plain data, it is written as $< S >$. Cryptographic keys are used to perform transformations on data which encipher or decipher it. Symmetric and private keys must be kept secret, while public keys can be revealed to the world. These three key types can be included as stand-alone components in a message, or they can be aggregated in non-terminal encryption nodes.

As in the case of nonces and timestamps, each of the components in this category is assigned a unique identifier. A unique integer identifier is created when the component is first inserted into a message. This identifier can be modified at a later stage by using the associated properties dialog. The properties dialog for a shared secret is displayed in Figure 3.9 (b). This dialog is the same as that used for symmetric, public and private keys except for one significant difference — it includes an extra tabbed pane called the *Functionality* pane. The *Functionality* tabbed pane is used to indicate whether a shared secret is being used as an identifier. The action of checking the embedded check-box only associates the shared secret as an identifier within the encapsulating container. The check-box in this tabbed pane is active when the shared secret is included in a hash, symmetric encryption or public key encryption and the corresponding textual label reflects the type of this container node.

When a public or private key is inserted into a specification, the corresponding private or public key partner is also created. Any data structures in memory pertaining to a public key reference the corresponding private key partner, and any private key data structures reference the corresponding public key partner. This feature is pivotal for later analyses, as many protocol specifications make use of only one asymmetric key and don't define its partner in any of the protocol messages. For example, in the Needham-Schroeder Public-Key Protocol [1], the following asymmetric keys are defined in the message passing specification: $+K_b$, $+K_a$ and $-K_s$. The corresponding public or private key is not mentioned for any of these keys. Thus, it is imperative that we automatically create $-K_b$, $-K_a$ and $+K_s$ and make sure that each public and private key references its partner, since protocol engineers may have to specify properties about these missing keys in analyses that follow.

Class C: *Other Terminal Components*

If a terminal component does not fit into Class A or Class B, then it is considered to be a generic component type. Components in this category could include principal identifiers, passwords, usernames or any other data structure that would be used to represent information in a protocol message. The identifier of a component in this class is its name. For example, in the encryption $\{A, N_a, Data\}_{K_{ab}}$ there are two plain components, and their identifiers are A and $Data$ respectively. Any two components in this class cannot share the same name, unless they are both duplicates of the same component. The dialog used to edit a plain component is structured in the same way as the one used to edit nonces, shown in Figure 3.9 (a) and includes all of the same graphical interface components.

Class D: *Functions and Hashes*

Due to the lack of code generation facilities with which to interact, a function or hash node merely operates as an anchor point for a collection of terminal nodes. With the current level of functionality, function statements such as $increment(N_a, 1)$ and hash statements such as $\{H(Document, T_a)\}_{-K_a}$ can all be defined. This level of functionality is sufficient for the aims set out in this dissertation, as we merely require that all functions and hashes be representable. The properties dialog for a function node allows one to set the name of the function. This name is known as the descriptor. Hashes do not have an associated properties dialog, as every hash function has the same name. To distinguish between functions with the same name and hashes used within the specification, a unique identifier is associated with each function or hash. When copying a function or hash node, all of the child nodes are copied and inserted into the hierarchical tree as well and the identifier of the resulting node stays the same. After inserting a function or hash node into the hierarchical tree, a protocol engineer has to add components to it, otherwise an error message will be generated if the node is left empty when closing the Component View.

Class E: *Cryptographic Transformations*

A cryptographic transformation uses a symmetric, public or private key to encipher or decipher information to be transmitted in a protocol message. There are four types of cryptographic transformations that can be represented within the Component View. These are symmetric encryptions and decryptions, public key encryptions and private key encryptions. When inserting an encryption or decryption node into the hierarchical tree, a key is automatically associated with the node. For example, if a public key encryption node is created, then a key named $+K_1$ would automatically be created to serve as the key used to carry out the encryption operation. An encryption or decryption node cannot be inserted into the tree and left empty and an error message will be generated when closing the Component View if there are any empty nodes which should have child nodes anchored off them. At present, protocol engineers can only specify the key associated with a cryptographic transformation and an explanatory text string called the descriptor.

The key used to carry out an encryption or decryption is aggregated within the cryptographic transformation node. This key can be renamed or even replaced with another key of the same type that already exists in the protocol by using the properties dialog. Figure 3.9 (c) shows a symmetric encryption properties dialog. The *Keys* tabbed pane has a drop-down list which contains a list of all the symmetric keys defined within the protocol specification. A protocol engineer can use this drop-down list to select a new key or she can use the *Edit Key* button to edit the properties of the currently selected key using the symmetric key properties dialog. The identification tabbed pane contains a text box which can be used to edit the descriptor of the node. This descriptor is displayed next to the node icon in the hierarchical tree. The default descriptor text is the type of the encryption or decryption represented by the node. For example, a public key encryption will have a default descriptor of 'Public-Key Encryption'. Also associated with each encryption and decryption node is a unique integer identifier used to distinguish the node from others of the same type.

Class F: *Component Collections*

In some protocols a set of components may be grouped together and sent and received among principals in unison. To cater for situations such as this, we have created a grouping non-terminal node which can be used as a container for these components. To copy or move all of the components in a group, a user merely needs to copy or move the grouping node. A descriptor can be associated with a group to describe it in greater detail in the hierarchical tree. The default descriptor is the text, 'Group'. Besides the descriptor, a protocol engineer can also specify whether a group should have brackets surrounding it. This property is useful when exporting the group to text or LaTeX formats. The properties dialog for a grouping node is shown in Figure 3.9 (d). As with all the other components, a unique integer identifier is associated with each grouping node.

Class G: *Message Tree Root*

The message tree root is not a member of any of the six official component classes, and thus we have included it within this extra seventh class. The main purpose of the message tree root is to serve as an anchor point for all of the first-level components. The only modifiable property of a message tree root is the code name of the message it represents. As we've already seen in the *Communications Settings* dialog, shown in Figure 3.7, each message in the protocol has an associated code name that is used to identify the message. The default code name for a message is 'Message n', where n is an integer describing the vertical position at which the message was first inserted into a protocol. This code name can be changed by using the message properties dialog which is accessible from the *Properties* item on the Component View pop-up menu which appears when right-clicking on the message tree root.

3.2.2.3 Exporting Messages and Components

The GYPSIE environment has facilities that allow one to export a protocol specification to text, LaTeX or Prolog-compatible output. Exporting a protocol entails converting each individual component into the correct format, concatenating these results into a message, appending the sender and the receiver, and then incorporating the resulting output in the context of the overall protocol specification. Terminal nodes are normally straight-forward to convert as they mostly require a token to be concatenated with the component's identifier. However, in the case of non-terminal nodes, there is usually a front and rear portion of output to create, all the embedded components being inserted between these two pieces. In the list that follows, we show the text and LaTeX representations for each of the Component View types. The Prolog representation has been excluded from this list and is described in Chapter 5.

GYPSIE Component	*Text*	LaTeX
Generic component named X	`X`	X
Nonce with identifier i	`Ni`	N_i
Timestamp with identifier j	`Tj`	T_j
Function with arguments X and Y named F	`F(X, Y)`	$F(X,Y)$
Hash of X concatenated with Y	`H(X, Y)`	$H(X,Y)$
Symmetric encryption of X with key K	`E(K : X)`	$\{X\}_K$
Symmetric decryption of X with key K	`D(K : X)`	$\{X\}_K^{-1}$
Public key encryption of X with key $+K$	`E(+K : X)`	$\{X\}_{+K}$
Private key encryption of X with key $-K$	`E(-K : X)`	$\{X\}_{-K}$
Shared secret S being used as an identifier	`<S>`	$< S >$
Symmetric key with identifier ij	`Kij`	K_{ij}
Public key with identifier i	`+Ki`	$+K_i$
Private key with identifier i	`-Ki`	$-K_i$

In this list we have shown how the LaTeX output will appear once it has been parsed. The actual LaTeX output consists of the textual mark-up codes. The plain textual representation of components is used in tooltips since it is easy to display and does not contain any subscripts. The textual notation used to represent cryptographic transformations has been taken from [23]. A protocol specification can be exported and saved to a file on disk, or alternatively it can be viewed in a dialog containing a text edit box. When exporting a protocol specification, messages are indented appropriately to indicate their membership of subprotocols. Any messages belonging to a subprotocol are clearly marked.

3.2.3 Navigator View

The aim of the Navigator View is to provide a protocol engineer with a concise overview of the structure and contents of a protocol specification. A tree-view with expandable and collapsible nodes is used to represent a protocol and to give an indication of the principals, messages and subprotocols that have been defined. Consider the protocol specification shown in Figure 3.10. The Navigator View is the window pane on the left of the screen-shot. From this Navigator View we can discern the following facts:

- *Meaningless Protocol* contains three principals (*Lana*, *Billy* and *Alice*), three messages (*Initialization*, *Exchange* and *Goodbye*) and two subprotocols (*John's Subprotocol* and *Mike's Subprotocol*).
- *John's Subprotocol* contains three principals (*Billy*, *Alice* and *Fred*) and two messages (*Authenticate* and *Drop*).
- *Mike's Subprotocol* contains four principals (*Lana*, *Billy*, *Derek* and *Fred*) and three messages (*Welcome*, *Transmit* and *Close*).

Besides providing an overview of a protocol's composition, the Navigator View also functions as an alternative means through which High-Level View functionality can be accessed. When clicking on nodes in the tree-view, pop-up menus in accordance with the selected node's type are displayed, allowing a subset of operations provided by the High-Level View to be conducted. The Navigator View can also be used to import principals and messages into the protocol currently being edited and to explore the subprotocols defined in a specification. Tooltips displaying the contents of a message appear when hovering over the message's code name in the Navigator tree-view.

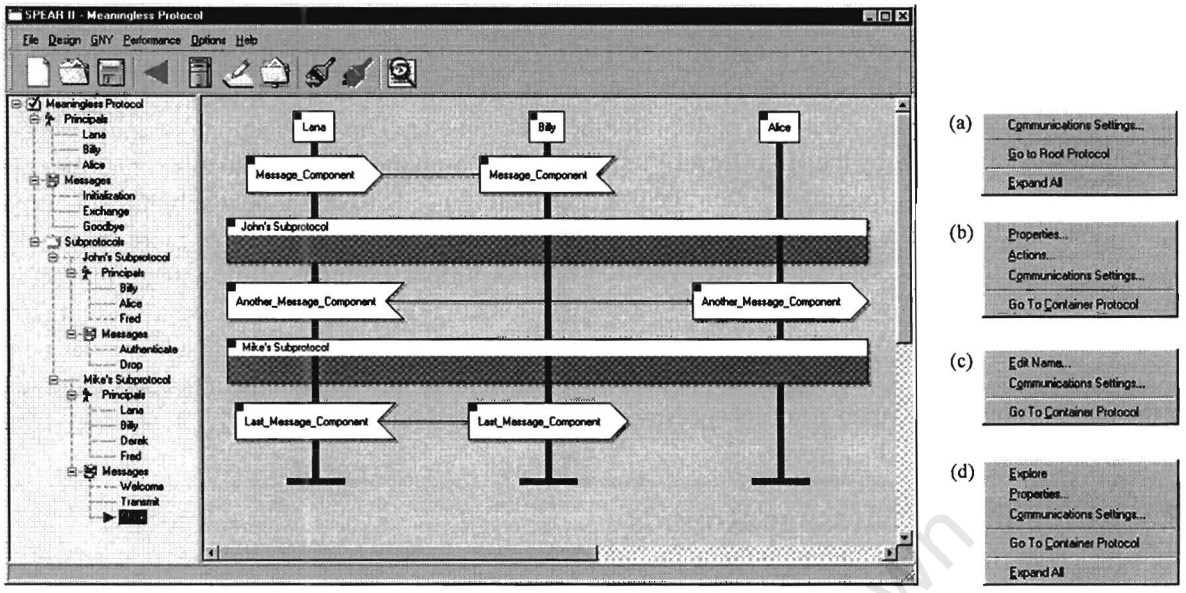


Figure 3.10: The Navigator View and associated pop-up menus.

The pop-up menus used in conjunction with the Navigator View are very similar to those used within the High-Level View. Figure 3.10 contains all four of the pop-up menus used in the Navigator interface. Pop-up menus (b) – (d) are invoked when clicking on a message node, principal node or subprotocol node respectively, while pop-up menu (a) appears when right-clicking on the root node of the tree-view. In pop-up menu (b) the first three commands are shared with the High-Level View message pop-up menu, in (c) the first two commands are shared with the High-Level View principal pop-up menu, while (d) shares the first three commands with the High-Level View subprotocol pop-up menu. The remaining commands are used to navigate through a subprotocol hierarchy and expand tree-view nodes.

In each of the pop-up menus, the *Go to Container Protocol* command makes the subprotocol of which the selected message, principal or subprotocol is a member the active one in the High-Level View. This means that a protocol engineer can have direct access to a subprotocol and he does not have to navigate to a given subprotocol by repeatedly using the *Explore* command from a subprotocol pop-up menu. The *Expand All* command recursively expands each of the tree-view nodes anchored off a selected node until all of the descendant nodes have revealed their children. Individually expanding or collapsing a node is accomplished by clicking the box-like token to the left of a node. Pop-up menu items are dimmed and rendered inactive if a command is unavailable or the subprotocol which is the target of a navigation command is already being edited in the High-Level View.

Tight integration between the Navigator and High-Level Views allows a user to copy principals and messages between subprotocols. To copy a principal into a subprotocol being edited in the High-Level View, one merely needs to drag the principal node from the Navigator View tree onto the High-Level View canvas. A principal axis will then be inserted at the point where the dragging operation ended. Similarly, copying a message merely involves dragging the message node from the Navigator tree-view onto the High-Level View canvas. The message will be inserted at the point where the dragging operation ended. When copying a message, the sender and the receiver are added to the target protocol if they are not already members thereof. The components inside the message are all duplicated so that they still reference the same data structures in memory as the source components.

If a principal already exists in a protocol, then another principal with the same name cannot be copied into the protocol and the copy feature of the Navigator View will not allow the drop operation to proceed. However, if a message that is being copied already exists in the target protocol, then it is merely duplicated and its code name has a dash appended after the last character. This duplication operation is exactly the same as that provided by the High-Level View message pop-up menu. Subprotocols cannot be copied into a protocol. Instead, to copy a subprotocol a user merely needs to add a new subprotocol, explore it, and then load the subprotocol file. The GYPSIE environment ensures that two or more subprotocols that stem from the same source file stay synchronized. A further example of the integration between the Navigator and High-Level Views is seen when right-clicking on a principal, message or subprotocol object on the High-Level View canvas. This act of selecting a canvas object results in a triangular pointer icon being displayed next to the message, principal or subprotocol tree-view node in the Navigator View. If the node is not visible, then all of its ancestors are expanded to reveal it.

3.3 Calculating Message Rounds

When constructing security protocols, most designers would like to have an indication of how optimal their protocols are with respect to standard, well-known performance metrics. In [35] Li Gong describes research that he conducted regarding the minimum number of rounds, messages and encryptions required for selected classes of network authentication protocols. To extend the SPEAR II Framework and make it more useful for designers wanting to obtain performance measurements, we decided to construct a protocol rounds analyzer to work in tandem with the GYPSIE environment. We chose to calculate the message rounds metric as we wanted to build on and extend research gathered during the construction of the original SPEAR application [6]. By using the rounds analyzer in conjunction with the principles presented in Gong's paper, a protocol engineer can determine how close a given protocol is to being optimal with respect to the number of rounds it employs.

As in [35], we make the assumption that the network on which protocol messages are exchanged is uniformly connected so that a message always travels from a source node to destination node in one unit of time, irrespective of the location of the sender and the receiver. The computation time at each node is neglected. One protocol round consists of all of the messages which can be sent and received in parallel within one time unit. Thus, it is possible for a principal to send different messages to more than one principal during a round. The number of rounds in a protocol is the number of time units that elapse from the instant that an originator sends the first message in the protocol to the instant that the last message is received. When calculating protocol rounds, we make a distinction between synchronous rounds and optimal rounds. The original SPEAR tool could only calculate synchronous message rounds.

When carrying out optimal rounds calculations, our only restriction is that a principal needs to possess all of the information contained in a message before it is transmitted. This information must have been received in a previous round. Any messages transmitted can be sent asynchronously. However, in the case of synchronous rounds calculations, we assume that only synchronous communication can take place between principals. Thus, a particular entity cannot send and receive messages asynchronously and the transmission of these messages has to progress in a sequential fashion. A reason for using synchronous rounds calculations is that there are situations where subtle side-effects of previous message receptions may affect the following messages or one may not want to progress further without receiving some kind of acknowledgment of reception. Synchronous rounds calculations always yield a value greater than or equal to that obtained when using the optimal rounds calculation technique on the same protocol.

```
Message message;
StringList receiversInRound;
Map messagesInRounds;
Integer round = 1;

for i = 1 to getMessageCount() {
    message = getMessage(i);

    if (receiversInRound.contains(message.getSender())) {
        round = round + 1;
        receiversInRound.clear();
    }

    receiversInRound.add(message.getReceiver());
    messagesInRound[round].add(message);
}
```

Figure 3.11: C++-style pseudocode for determining the synchronous rounds in a protocol.

In Figure 3.11 we have listed C++-style pseudocode to describe how the synchronous rounds for a set of protocol messages can be determined. This algorithm groups together all of the messages that belong to the same synchronous round. The messages in round n are placed into the list denoted by the map structure `messagesInRound[`n`]`. The basic principle underlying synchronous round calculations is that no principal who has received a message in a given synchronous round can send another message in the same round. In Figure 3.12 similar C++-style code for generating the optimal message rounds for a protocol is described. The basis behind this algorithm is that a principal cannot send a component in a message round if it has not received that component in a previous round. If a principal sends a component which it has received in the current round, then the rounds counter, named `rounds`, is incremented and the message and all the ones following thereafter become part of one of the subsequent rounds. As can be seen from the pseudocode, the implementation of optimal rounds checking requires a lot more effort and, most importantly, the ability to examine the components transmitted within messages. Also, notice that the optimal rounds algorithm determines the initial possessions of every principal involved in the protocol using the `getInitialPossessions()` function call. To illustrate the difference between synchronous and optimal rounds calculations, consider the protocol that follows [35]:

(1) $A \longrightarrow S : A, B$
(2) $S \longrightarrow A : \{S, A, A, K_{ab}, B, T_s\}_{K_{as}}$
(3) $S \longrightarrow B : \{S, B, A, K_{ab}, B, T_s\}_{K_{bs}}$
(4) $A \longrightarrow B : \{A, B, T_a\}_{K_{ab}}$
(5) $B \longrightarrow A : \{B, A, T_b\}_{K_{ab}}$

Optimal Rounds: $\{1\}, \{2, 3\}, \{4, 5\}$
Synchronous Rounds: $\{1\}, \{2, 3\}, \{4\}, \{5\}$

This protocol consists of three optimal rounds and four synchronous rounds. It is easy to see that message (1) must proceed before any of the others can, thus constituting the first round. The second round consists of messages (2) and (3) since the server S doesn't receive any new information between these two messages. However, messages (4) and (5) cannot proceed because the shared key K_{ab} hasn't been received yet. Now, the third optimal round consists of messages (4) and (5). This is not as obvious as the

```
Message message;
StringList receiversInRound;
Map messagesInRounds, receivedInProtocol, receivedInRound;
List componentList;
String principal, sender, receiver;
Integer round = 1;

for i = 1 to getPrincipalCount() {
    principal = getPrincipal(i);
    getInitialPossessions(principal, receivedInProtocol[principal]);
}

for i = 1 to getMessageCount() {
    message = getMessage(i);
    receiver = message.getReceiver();
    sender = message.getSender();

    if (receiversInRound.contains(sender) {
        componentList = receivedInRound[sender];
        for j = 1 to componentList.count {
            if (message.contains(componentList[j])) {
                if not(receivedInProtocol[sender].contains(componentList[j])) {
                    round = round + 1;
                    for k = 1 to receiversInRound.count {
                        principal = receiversInRound[k];
                        receivedInProtocol[principal].add(receivedInRound[principal]);
                    }
                    receiversInRound.clear();
                    receivedInRound.clear();
                    break;
                }
            }
        }
    }

    receiversInRound.add(receiver);
    messagesInRounds[round].add(message);
    componentList = message.getContents();
    for j = 1 to componentList.count {
        receivedInRound[receiver].add(componentList[j]);
    }
}
```

Figure 3.12: Pseudocode for determining the optimal rounds in a protocol.

above rounds since it appears that B first receives a message and *then* sends a message. The subtle thing to notice is that message (5) does not rely on any information received from message (4) and therefore it can be sent simultaneously with message (4). However, there are four synchronous message rounds in the protocol. This is because it is assumed that some side-effects can take place between message (4) and (5), therefore disallowing their concatenation into a single round. Now, consider the following protocol:

(1) $A \longrightarrow B : N_1, N_2$
(2) $B \longrightarrow A : \{N_1\}_{K_{ab}}$
(3) $A \longrightarrow B : N_3$
(4) $B \longrightarrow A : H(N_2, N_4)$

Optimal Rounds: $\{1\}, \{2, 3, 4\}$
Synchronous Rounds: $\{1\}, \{2\}, \{3\}, \{4\}$

This protocol is not useful to any extent and serves more to illustrate the differences between optimal and synchronous rounds calculations. When performing the relevant calculations, we end up with two optimal rounds and four synchronous rounds. The reason why there are so many synchronous rounds is because the protocol alternates between two principals and each principal can only send a message after it has received the previous message. On the other hand, the optimal rounds calculation takes into consideration the fact that when transmitting the last three messages, the senders both already possess the items to be transmitted, and thus they do not need to wait for these components to be received in a prior message and can instead just broadcast all three of these messages simultaneously.

To perform rounds calculations in SPEAR II, the *Performance* pull-down menu is used to reveal the *Calculate Synchronous Rounds* and *Calculate Optimal Rounds* commands. Upon initiating one of these commands, the relevant calculations are conducted and a dialog box is displayed containing the details of the rounds computation. The dialog contains the number of rounds and also lists the messages involved in each of these. Besides rounds calculations, we also foresee more performance measurement functions being added to this menu. Such functions could work in tandem with code-generation routines to carry out simulations and gather timing information, or they could compute more stand-alone metrics like the rounds calculations which we have implemented.

3.4 Experiments with the GYPSIE Environment

In order to obtain an indication of how protocol engineers will interact with the GYPSIE design environment, we decided to conduct a number of user experiments. These experiments were essentially structured as a series of case studies and allowed us to examine how individuals made use of the system, the time they took to create protocol specifications, and the accuracy with which they completed these specifications. All twenty participants who took part in the experiments were Computer Science majors who had recently completed a fourth-year network and internetwork security course which included cryptographic protocol design principles as part of the curriculum. As such, this sample of individuals represented a collection of users who are reasonably likely to make use of a system similar to the SPEAR II Framework at some point within their chosen profession.

The experiments which we developed required users to construct three cryptographic protocols, answer questions about a specification and manipulate an existing protocol in the High-Level View. Users were asked to input a voting protocol [33], an authentication protocol which we had devised, and a modified version of the Needham-Schroeder protocol [33] using GYPSIE. A screen-shot of the Navigator and High-Level Views was used to test how much information about a protocol's structure and composition users could gather from these two environments. Finally, the participants were asked to modify and manipulate a model of the Kerberos protocol in the High-Level View. Besides helping us to determine how users interacted with the design environment, the experiments also gave each of the participants an opportunity to exercise the GYPSIE environment and test the system for any latent bugs. The experiments test sheet can be found in Appendix C.

GYPSIE Type	Voting Protocol	Authentication Protocol	Needham-Schroeder
Generic Components	5	2	8
Nonces	4	5	10
Functions	0	0	1
Hashes	2	0	0
Public Key Encryptions	0	1	0
Private Key Encryptions	0	1	0
Symmetric Encryptions	0	2	7
Asymmetric Keys	0	2	0
Symmetric Keys	0	3	8
Shared Secrets	2	0	0
Principals	2	2	3
Messages	3	3	7

Table 3.2: The distribution of cryptographic types in the protocols specified in the experiments.

Table 3.2 describes how the component types were distributed among the voting, authentication and Needham-Schroeder protocols. When examining this table we notice that the voting protocol does not contain any encryption nodes, but instead uses only hashes to group components. An important point to keep in mind is that the shared secrets used in the voting protocol are both used as identifiers and not as data. Both the authentication and Needham-Schroeder protocols contain encryption nodes. However, the Needham-Schroeder protocol only uses conventional encryption, while the authentication protocol uses both conventional and public-key encryption. None of the protocols which the participants had to specify contained subprotocols. However, the model comprehension portion of the experiment, which employed the use of a screen-shot containing the Navigator and High-Level Views, included a subprotocol hierarchy. Participants were asked to examine this hierarchy and then answer questions related to the principals, messages and subprotocols contained therein.

Protocol	Construction Time				Number of Errors	
	Sample Mean	95% Confidence Interval for Population Mean	Min	Max	Sample Mean	95% Confidence Interval for Population Mean
Voting Protocol	300s	(258s, 348s)	173s	450s	0.65	(0.10, 1.26)
Authentication Protocol	378s	(306s, 459s)	214s	673s	0.20	(-0.23, 0.68)
Needham-Schroeder	589s	(518s, 669s)	400s	958s	0.60	(0.15, 1.11)

Table 3.3: Results of tests pertaining to GYPSIE protocol construction.

The time that each of the participants took to construct the voting, authentication and Needham-Schroeder protocols using GYPSIE was recorded. Each of their specifications was also saved to disk and the number of errors that had been made while modelling the protocol was recorded. Errors that we checked for included incorrect subscripts, encryption types and component types and improper ordering of components. The model comprehension questions were answered on the question sheet and marked afterwards, while the manipulation questions were answered by the participants physically demonstrating the required tasks. None of the participants had ever used SPEAR II before and as a result each of them was given a brief introduction to the system before proceeding with any of the questions. The results obtained from the protocol construction experiments are listed in Table 3.3 and a set of graphs detailing the construction times and errors for each individual user is shown in Figure 3.13.

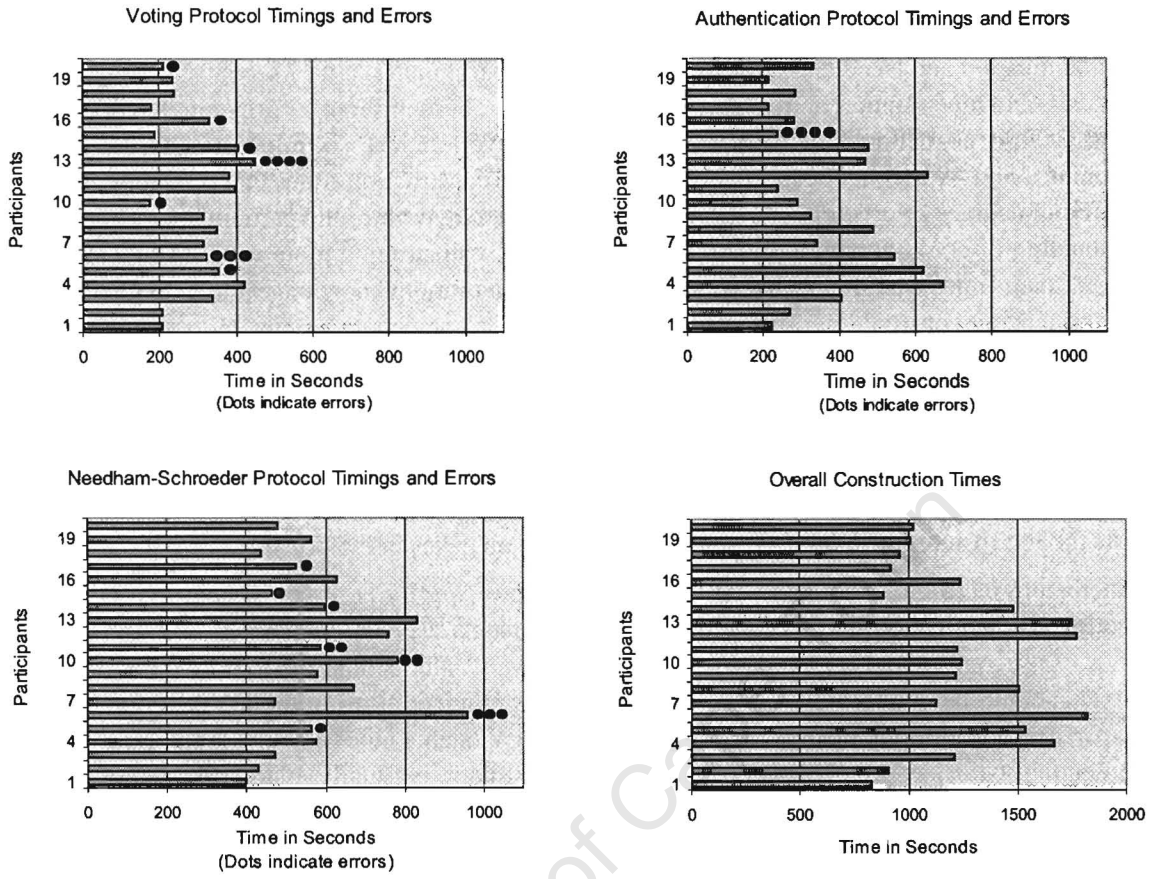


Figure 3.13: Time taken and number of mistakes made when participants specified protocols.

The timing and error results that we recorded essentially function as a benchmark at this point. It was difficult to obtain access to other protocol design environments, and for those we did have access to, training people to use them and then testing them in this area would not have been a simple task and would have consumed more time and finances than we had available. Also, subtle differences in how these design environments work would have made any comparisons a non-trivial task. For example, different design environments define a complete protocol specification in different ways and thus require differing amounts of information to be supplied before a specification can be concluded. Thus, a simplistic design environment might allow a protocol engineer to quickly specify a protocol, but the resulting specification might not be very comprehensive. On the other hand, a more advanced environment might take longer to use, but the resulting specification could well be more extensive and useful. In effect, what we have done is to set a standard that other design environments similar to GYPSIE can be compared against. In Table 3.3 the mean, minimum and maximum construction times for each protocol are listed. From the data we calculated the 95% confidence interval indicating where the population mean for the construction time should lie. We also calculated the mean number of errors committed per protocol and the 95% confidence interval for this population mean. In Figure 3.13 the number of dots to the right of each bar indicate the number of errors made during the time that the protocol in question was being constructed.

When examining the graphs in Figure 3.13, we notice that the construction times are widely distributed. Glancing at Table 3.3 reveals that the difference between the minimum and maximum times for the voting, authentication and Needham-Schroeder protocols is four minutes and 37 seconds, seven minutes and 39 seconds, and nine minutes and 18 seconds respectively. These differences are quite large, considering that the mean construction time for these protocols are five minutes, six minutes and 18 seconds, and nine minutes and 49 seconds respectively. This spread of construction times can probably attributed to the differing degrees of cryptographic knowledge possessed by each of the participants. Those with less understanding of cryptographic protocol design principles will struggle more when using GYPSIE, and as a result their times would most likely be lower than those of more knowledgeable individuals. Another factor influencing construction times could be the fact that some people are just more proficient at using graphical interfaces and able to use them more naturally. In fact, during the course of the experiments we noticed that certain individuals did not seem to familiarize themselves with the environment that easily, while others immediately mastered the three views and were able to construct protocols accurately and efficiently. Another factor to consider is that individuals sometimes decide to favour accuracy at the expense of time and thus take longer to complete a specification. Others consider time to be more significant and rush to finish quickly, sometimes making mistakes in the process.

An interesting metric to examine is the mean time taken to specify each component in a protocol. For the voting, authentication and Needham-Schroeder protocols a given component takes on average 27 seconds, 23 seconds and 17 seconds to specify respectively. Notice that the trend is one of decreasing time. This could be due to individuals becoming more proficient at using the GYPSIE environment as time increases, resulting in them being able to specify components more quickly. Another explanation is that in certain cases, as protocols become more complex, there is usually a larger degree of component repetition and as a result, components are added using the copy and paste or *Add Existing* feature of the Component View, resulting in reduced insertion times. The population mean interval computed from the construction times help to view the results we obtained in context. When looking at the 95% confidence intervals we notice that on average individuals will specify a given security protocol within a very reasonable amount of time. In fact, an average individual within our target population will take in the order of six minutes, eight minutes and eleven minutes to specify the voting, authentication and Needham-Schroeder protocols respectively.

For each protocol specification the mean number of errors that an average individual within our target population will commit is at most one for the voting and Needham-Schroeder protocols, and a fractional value below one for the authentication protocol. Out of all sixty protocol specifications that were constructed, forty-five of these were completely error free. Ten of the participants committed errors when specifying the protocols. Five of these individuals made an error in two protocol specifications, while the remaining five only made an error in one protocol. The ten participants who did not create flawless specifications committed an average of 2.7 errors each while constructing the three protocols. Three individuals specified incomplete or incorrect subscripts, resulting in a total of seven incorrect subscripts. In the case of the voting protocol, two individuals forgot or did not know how to specify the fact that a shared secret was being used as an identifier and not as mere data. Another error committed by three participants was the use of incorrect component types — for example, using a generic component instead of a nonce and then naming it 'Na' or something similar. Seven of the participants struggled to determine which message components should be generic component types or more specialized components, however, four of them ended up making the right decision. With regard to transcription errors, six individuals placed components in the incorrect location or left components out of a message.

Four of the participants specified incorrect message senders or receivers. A contributing factor to these errors could have been that these individuals were not totally comfortable with the sender and receiver notation of the High-Level View. We noticed that to ensure that the senders and receivers were correct some individuals used the Component View or viewed a protocol specification as text. However, when manipulating the Kerberos protocol, all of the participants managed to correctly change the sender and receiver of a message using High-Level View drag-and-drop operations. To an extent, the stress factor associated with constructing a protocol specification sometimes forces individuals to overlook correctly setting a message's sender and receiver as they tend to get caught up in creating the actual structure and embedded components of a message. This stress also affects their ability to accurately validate a protocol specification. On at least one occasion the errors which an individual had made were fairly blatant and clearly visible, yet he did not even notice or attempt to correct them.

Approximately four of the participants struggled to create encryption nodes. Instead of inserting an encryption node into the specification, they inserted the corresponding cryptographic key and then tried to add the encrypted components to it. Of course, this did not work as key nodes are all terminal. However, these users eventually figured out how to create encryptions by experimenting with the interface, although it did incur a time penalty. The fact that these individuals had difficultly creating encryptions is difficult to understand, as everyone was shown how to create encryptions during the SPEAR II introduction prior to the experiment. Also, the commands to add encryption nodes are clearly visible in the Component View *Add New* pop-up menu. However, because the encryption node text in the tree-view does not plainly display the key associated with an encryption, users might feel that an encryption is not actually being represented as they cannot immediately associate a public, private or symmetric key with the node. On the other hand, we did notice that most of the participants made use of the Component View tooltips to determine the key associated with an encryption node.

During the construction of the voting, authentication and Needham-Schroeder protocols we were able to observe how individuals made use of the Component View. In general we found that once users had become acquainted with the various terminal and non-terminal node types, they were able to modify message contents with ease. As we have already mentioned, the types that posed the most difficulties were the encryptions. No significant problems were encountered with hash and function nodes, although one individual created a function instead of a hash and then renamed this function to 'H' to make it appear as a hash. To reorder components some individuals first tried to use drag-and-drop operations and only when this did not work did they realize that there was a *Move* command on the Component View pop-up menu. One individual went so far as to delete all of the message components when he discovered that they were in the incorrect order. He then re-inserted all of these components in the correct order.

The model comprehension portion of the question sheet presented users with a screen-shot of the Navigator and High-Level Views which they had to answer questions about. Essentially, what we were testing in this section was how easily individuals would make use of the information contained in these two views and how well they could understand and apply it. Of the twenty participants, only three answered questions incorrectly in this section. In fact, most of the participants were able to easily make use of the Navigator tree-view in conjunction with the High-Level View and deduce facts about the sample protocol. During the SPEAR II introduction prior to each experiment, we explained to all of the participants the meaning of the translucent subprotocol bodies which appear above the axes of principals involved in a subprotocol. When asked to determine which principals the currently displayed protocol had in common with a direct child subprotocol, eleven out of the nineteen participants that answered correctly indicated that they had determined this information by searching for translucent axes in the High-Level View. The remaining eight used the Navigator View and compared the principal names in the *Principals* category of each protocol.

All of the participants were able to correctly manipulate the Kerberos protocol as required by the High-Level View manipulation questions. However, approximately six individuals took some time before realizing that the cross-hair in the center of a dragged message must be placed on a principal axis before a drop is considered valid. Instead, these individuals tried to reorder messages and change senders and receivers by dragging and dropping message objects in the vicinity of their new position. Of the twenty participants, three hesitated when having to decide which message boxes represented the sender and the receiver, but they all eventually arrived at the correct solution, changing senders and receivers appropriately. We had explained to the participants how to move messages and change senders and receivers prior to the experiment. However, it seems as though these individuals had forgotten this information or were not concentrating during the explanation. Another factor could be that we just explained too much information too quickly during the SPEAR II introduction, not giving them enough time to absorb it all. All of the participants were able to edit message contents and principal names. One individual even used the Navigator message pop-up menu to bring the Component View to the foreground so that he could edit a message.

The GYPSIE environment was created to facilitate the modelling of security protocols. However, as with most tools, a user should have an understanding of the underlying theory that the tool implements in order to make effective use thereof. The GYPSIE environment is not of much use to someone who has little or no understanding of security protocol design concepts. The individuals who took part in this evaluation had all attended a one semester security course that included protocol engineering as part of the curriculum. However, because of the differing levels of skill and knowledge possessed by each of these participants, the time taken to construct protocols in GYPSIE varied, quite remarkably in some cases. A positive point to note was that everyone managed to complete three diverse protocol specifications in a reasonable amount of time, each specification being completed in under ten minutes on average. The error level was also relatively low, with 75% of the specifications produced being totally error-free. As with any tool, effectiveness and productivity increases with familiarity. However, in this situation none of the participants had even used SPEAR II before, yet they were still able to complete all of the tasks set before them. With these facts in mind, and judging from the results obtained, we feel that the GYPSIE environment will be able to simplify, enable and facilitate the construction of accurate protocol specifications when used by cryptographic protocol engineers.

3.5 Implementation Details

An object-orientated approach was employed when creating the GYPSIE environment in order to facilitate expansion and understanding of the source code. GYPSIE itself consists of approximately 18000 lines of source code and was written using the Borland C++ Builder package which significantly aids in the creation of event-driven Windows applications. In essence, the GYPSIE framework provides a mechanism through which a protocol and its associated messages can be defined. API calls allow built-in SPEAR II modules, such as the rounds calculator and Visual GNY environment, to access this design information and use it for further protocol engineering tasks. These API calls can also be tailored to package and export the information stored in GYPSIE so that it can serve as input to applications which form part of the SPEAR II Framework and are executed using system calls. An example of such an application is the Prolog-based GNY analyzer, known as GYNGER. Thus, the GYPSIE API facilitates the retrieval and manipulation of specifications created in the High-Level and Component Views by external and built-in protocol engineering modules. Besides these functions, the API also handles tasks such as drawing the canvas, saving and loading specifications, and controlling interaction with the user.

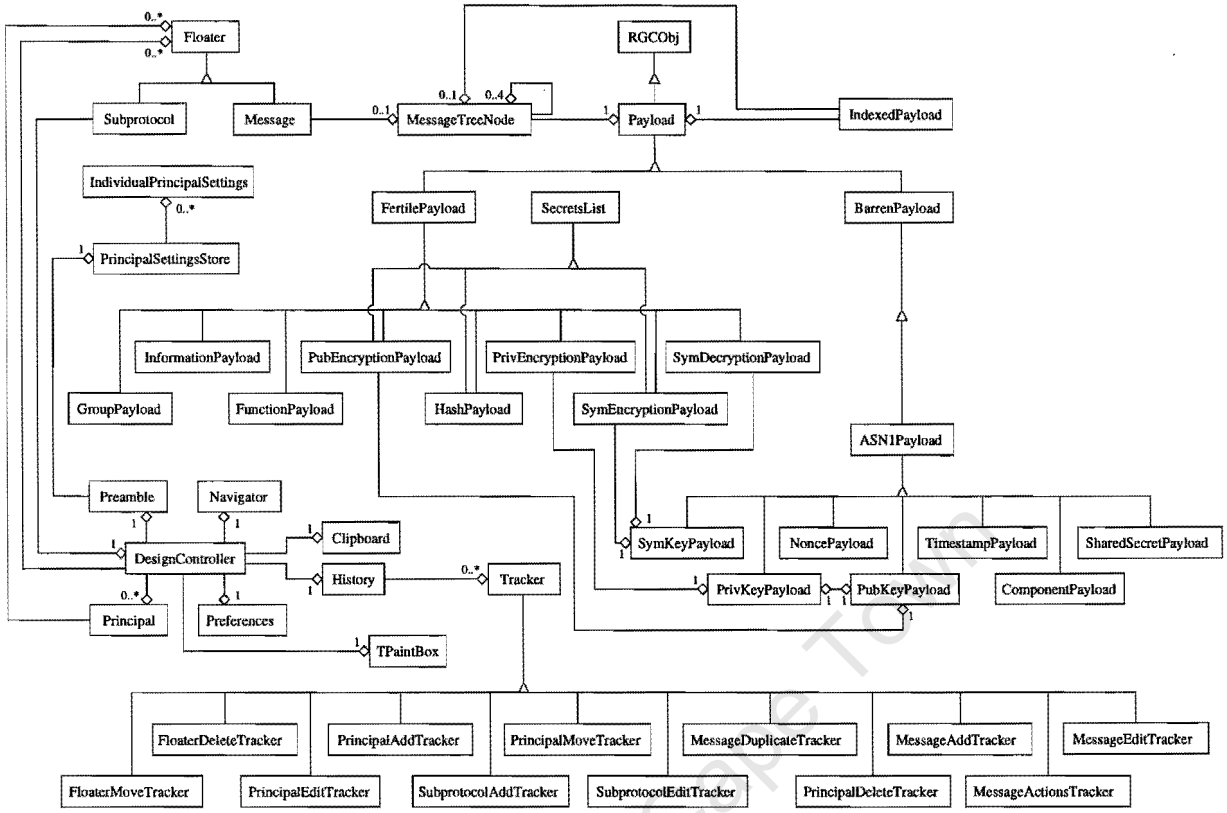


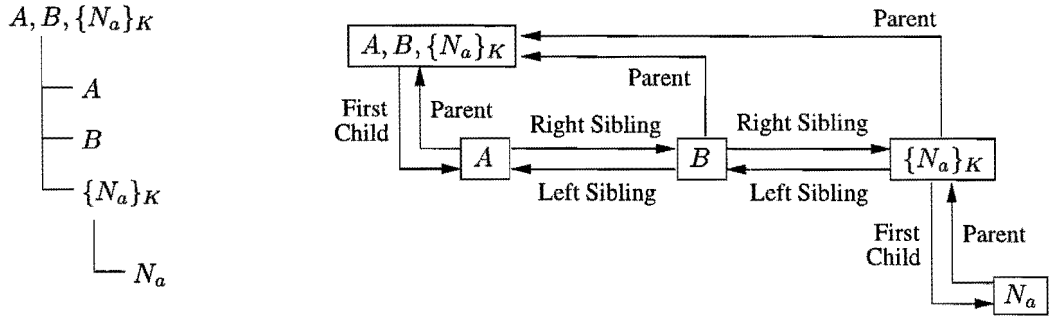
Figure 3.14: Diagram of selected classes used in the GYPSIE implementation.

3.5.1 Class Hierarchy

The GYPSIE class hierarchy consists of approximately 59 classes, 49 of which are illustrated in Figure 3.14. These classes can be divided into four broad categories. There are 21 classes which are used for storing and modifying message components, each component type ultimately being represented by a separate class. Another two classes are used for representing the structure of a message by means of a sibling-child tree approach and loading stored messages from file. Four classes are used to store, render and manipulate message, principal and subprotocol High-Level View components which appear on the design canvas. Finally, 22 classes are used to store and manage protocol designs, interact with the user and control GYPSIE sessions.

3.5.1.1 Storing Message Components

All of the classes used to represent message components inherit attributes and behaviour from the top-level `Payload` class. There are two primary types of message components. The `FertilePayload` class represents those classes which can contain child components. Payloads in this class include hashes, encryptions and functions. On the other hand, message components which are terminal inherit from the `BarrenPayload` class. At present, all of the `BarrenPayload` classes also inherit from the `ASN1Payload` class, which is a direct descendant of `BarrenPayload`. The `Payload` class is a descendant of the `RGCObj` class. This class is used to implement reference count-based garbage collection in C++ and is critical for memory management within the SPEAR II environment. In order to enable the



Standard Tree Representation

Sibling–Child Representation

Figure 3.15: Two ways of representing message contents.

use of identifying secrets within public key encryptions, hashes and symmetric encryptions, all of these components inherit functionality from the `SecretsList` class. This class contains a list of shared secrets that are used as identifiers within the encapsulating containers. Thus, to determine whether a shared secret is being used as an identifier within a suitable container, we merely need to check if it appears in this list. All of the encryption components aggregate a cryptographic key of the appropriate type. The public and private key types both contain a reference that points to the corresponding partner key. This allows the identifiers of both keys to stay in synchronization and it also ensures that a public or private key does not appear in isolation in a specification. The `InformationPayload` class is used to store a message's code name and is associated with the root node of a message.

3.5.1.2 High-Level View Objects

In the class hierarchy which we have developed, messages and subprotocols are represented by the `Message` and `Subprotocol` classes respectively. Both of these classes are derived from the `Floater` class. This generalization allows one to manipulate message and subprotocol objects displayed on the design canvas more easily, as they can be treated identically for certain operations, such as deletion and reordering. Deriving messages and subprotocols from a single base class also makes the storage and manipulation of these components within a linked-list much easier. To aid in identifying message and subprotocol objects, the `Floater` base class contains virtual methods that return a derived object's type. Every `Message` object contains a `MessageTreeNode` object that contains all of the components stored in the message. Each subprotocol contains a `DesignController` object, which contains all of the information related to the subprotocol being included in the protocol.

Each principal is represented by a `Principal` object. Objects in this class contain a text string representing the principal's name as well as a reference to the list of messages and subprotocols which is stored in the `DesignController` associated with the protocol. Thus, `Principal` objects can directly manipulate and access messages in a protocol specification. A situation in which this is useful is when a `Principal` object changes its name and then modifies the sender and receiver names of messages which it originates or receives. This aggregation also allows `Principal` objects to be given the responsibility of rendering their own messages. The `Message`, `Subprotocol` and `Principal` classes all have methods that allow them to be rendered to a given canvas. The rendering style can either be normal, in which case the colours specified in the SPEAR II Preferences Dialog will be used. Otherwise, an XORed representation can be generated at a predefined location on the canvas.

3.5.1.3 Representing the Structure of a Message

While the actual contents of a given component is saved within an individual `Payload` class, the structure of a message is stored by a collection of `MessageTreeNode` classes. The `MessageTreeNode` class effectively provides a mechanism for representing the structure of a message through through the use of a sibling-child tree approach (see Figure 3.15). Pointers within a given `MessageTreeNode` are used to reference the parent, left sibling, right sibling and first child of a given node. Because of this representation technique, the contents of a message can be treated as a binary tree and manipulated as such. Within the `MessageTreeNode` class, methods are provided to extract information, manipulate nodes and link the tree to the tree-view contained within the Component View. Every `MessageTreeNode` contains a class derived from the `Payload` class. This class contains all of the details pertaining to the component represented by the tree node. Thus, when querying or manipulating a `MessageTreeNode`, the underlying `Payload` class is actually being viewed or modified. The `IndexedPayload` class is used when loading a saved specification from disk and contains an integer index, a `Payload` and a possible `MessageTreeNode` that references a component's parent node.

3.5.1.4 Managing Protocol Design

The `DesignController` class is the central controller and storage repository within a protocol specification. For every protocol present in a specification, there exists one `DesignController` object wherein the protocol messages, principals and further subprotocols are embedded. Most of the API calls in GYPSIE are in fact methods of the `DesignController` class. The `DesignController` class encapsulates a number of important classes. The `Preamble` class is used to store protocol properties and principal settings. This information includes details such as comments, default communications ports, transport protocols and the IP addresses of principals. Settings pertaining to individual messages are stored in `Message` objects. Principal settings are stored in `IndividualPrincipalSettings` objects which are contained in a `PrincipalSettingsStore` embedded in a `Preamble` object. This approach is used to ensure that principal settings remain consistent across subprotocols included in the same specification. When modifying the settings of any principal, the root protocol `DesignController` has its `Preamble` object updated, since all of the principal settings are stored in the `PrincipalSettingsStore` embedded in the root `Preamble` object.

The `Preferences` class stores user settings specified in the SPEAR II Preferences Dialog. Settings within this class specify details such as the colours to be used when drawing canvas components, the style in which a dragged component is drawn, save paths and tooltip activation settings. The `Preferences` object is loaded from an initialization file stored on disk. If this file is not present, then default settings are used. All `DesignController` objects reference the same `Preferences` object. A `Clipboard` object is shared between all of the `DesignController` objects and is used to facilitate the transfer of `MessageTreeNode` objects between messages. Because this clipboard is shared, components can be copied between subprotocols. If a non-terminal component is present in the clipboard, then all of the child nodes are pasted as well when it is inserted into a message. The `History` class contains a list of `Trackers` and is used to facilitate the undo and redo feature described in Section 3.2.1.7. A class derived from `Tracker` exists for every operation defined in the High-Level View. These classes are stored as a queue within a `History` object, with a queue pointer being used to obtain the next operation to be undone or redone. Information contained within each `Tracker` can be used to undo or redo High-Level View operations. For example, the `PrincipalDeleteTracker` class stores the information pertaining to a deleted principal and all of the messages which it sends and receives.

The `Navigator` class is used to manipulate the Navigator View. Embedded within each `Navigator` object is a reference to the tree-view used to display the protocol structure. A `Navigator` object is encapsulated within each `DesignController` and the API calls provided by this `Navigator` object can be used to add, remove and edit nodes in the associated Navigator tree-view. When modifying a specification, the coupled `DesignController` automatically modifies the embedded `Navigator` object by invoking the appropriate API calls so that the Navigator View is updated as required. Most of the `Tracker` objects manipulated by the `History` and `DesignController` objects include routines which modify the Navigator View appropriately when conducting undo and redo operations. These `Tracker` objects are passed a reference to the embedded `Navigator` object through method parameters so that they can save or restore information as necessary. Also stored within each `DesignController` is a list of `Principals` and a list of `Floaters`. These two lists are used to manage principals, messages and subprotocols — messages and subprotocols being stored in the same list. The order in which components are stored in these lists parallels their order in the protocol specification and as a result, no other ordering information is required. Each `DesignController` also contains a pointer to a `TPaintBox` object so that all rendering operations can be directed to the correct canvas. This pointer is instantiated when the `DesignController` is created.

3.5.2 Saving and Loading Specifications

All of the information contained in a SPEAR II protocol specification is saved to disk as ASCII text using a simple block-like format. The GYPSIE portion of a standard SPEAR II source file is divided into four sections which are used to store the preamble, component information, principal records, message details and subprotocol properties. Other SPEAR II modules are free to add their own sections to the file format, since a given module only reads sections that apply to it, discarding the rest. The protocol shown below is referred to as *Sample* throughout this section and has been created for illustrative purposes. It contains ten components: two nonces, two hashes, two private key encryptions, two public keys and two private keys. Three messages, a subprotocol and two principals are also present in the specification:

(1) $A \longrightarrow B : N_a$
(2) $B \longrightarrow A : \{H(N_a), N_b\}_{-K_b}$
Foobar Subprotocol Executes
(3) $A \longrightarrow B : \{H(N_b)\}_{-K_a}$

When storing the details applicable to a given principal, we only save its name and position relative to the other principal components. In the case of protocol messages, the constituent components, structure, communications settings and sequencing associated with a given message are all written to disk. The file format which we have developed ensures that duplicate message components can be correctly restored, with each duplicate referencing the same data in memory after a file load operation. When saving subprotocols we store the location of the subprotocol source file, the subprotocol's friendly name and its execution style. The text extract which follows shows the principal section of the *Sample* protocol's source file:

```
PRINCIPAL {
    NAME = "A"
}
PRINCIPAL {
    NAME = "B"
}
```

Each PRINCIPAL block has a solitary field containing the name of the principal which it represents. One block is allocated per principal and the order in which these blocks are written parallels the order in which the principals are rendered on the canvas. In this case, A will be rendered to the left of B when the protocol is drawn. The preamble section of a SPEAR II source file stores the communications settings for every principal, as well as the default message settings. Only the source file of the root protocol contains a preamble with principal communications settings. The remaining subprotocols obtain these settings from the DesignController object associated with the root protocol after a file load operation. However, a given preamble block always contains information describing the default message settings for a protocol. The preamble section of the *Sample* protocol's source file appears as follows:

```
PREAMBLE {
   COMMENT = ""A sample protocol for","the SPEAR II thesis.""
   MSG_TRANSPORT = "udp"
   MSG_PORT = "4443"
   MSG_PORT_SELECT = "explicit"
   PRINCIPAL_SETTINGS {
       A {
           IP = "137.158.130.52"
           DOMAIN = ""
           ACTIVE_SETTING = "ip"
       }
       B {
           IP = "137.158.130.51"
           DOMAIN = ""
           ACTIVE_SETTING = "ip"
       }
   }
}
```

In this text fragment the default transport protocol is UDP and the default port on which messages will be sent is 4443. Other options for the MSG_TRANSPORT field are tcp and src. A setting of src means that the designated field is defined in source code. The MSG_PORT_SELECT field can either be set to explicit or src. If it is set to explicit, the MSG_PORT field defines the default port number. The COMMENT field is used to save a multi-line comment attached to the protocol. The PRINCIPAL_SETTINGS block contains the name of every principal taking part in the protocol and a list of communications settings for each of them. At this point, a user can only specify the principal's IP address and domain name. The ACTIVE_SETTING field indicates which of these is currently in force. The definition blocks for five components defined in the *Sample* protocol appears below:

```
PAYLOAD (mcNonce, 14347972) {
   ID = "a"
   ASN.1 = ""
}
PAYLOAD (mcPrivKey, 14348112) {
   ID = "b"
   ASN.1 = ""
   PUBKEY = 14348348
}
PAYLOAD (mcPubKey, 14348348) {
   ID = "b"
   ASN.1 = ""
   PRIVKEY = 14348112
}
```

```
PAYLOAD (mcHash, 14340540) {
   ID = "1"
   DESCRIPTOR = "Hash"
}
PAYLOAD (mcPrivEncryption, 14348740) {
   ID = "1"
   DESCRIPTOR = "Private-Key Encryption"
   PRIVKEY = 14348112
}
```

This text fragment is written in two columns to save space. In the actual source file the mcHash block would appear below the mcPubKey payload. When writing a component to disk, it is assigned a unique integer identifier which is used to reference it in later sections of the SPEAR II source file. This identifier is specified in parenthesis as the second parameter of the PAYLOAD keyword. The first parameter is a text string indicating the component's type. Within each component block are the fields which are required to restore the component correctly. The encryption payloads all store the integer identifier of the cryptographic key embedded within them. This implies that cryptographic keys must be specified before the encryption in which they are included so that undefined references do not result. In fact, all terminal components are always saved to disk before the non-terminal components. In the case of asymmetric keys, each key contains a reference to its partner. The file load operation caters for this special case of undefined references while reading the source file and restores the references correctly after both keys have been loaded. The list of shared secrets being used as identifiers is recorded through the use of a SECRETS block placed within a hash, symmetric encryption, or public key encryption block. This block contains a list of the integer identifiers of the shared secrets being used for identification purposes within the given encryption or hash container. The definition block for the second message which is sent from B to A and contains the components $\{H(N_a), N_b\}_{-K_b}$ appears below:

```
MESSAGE {
    SENDER = "B"
    RECEIVER = "A"
    ACTIVE_SETTINGS = "subprotocol"
    SETTINGS {
        TRANSPORT = "udp"
        PORT = "4443"
        PORT_SELECT = "explicit"
    }
    ACTION {
        SENT_PREACTION = ""
        SENT_POSTACTION = ""
        RECEIVED_PREACTION = ""
        RECEIVED_POSTACTION = ""
    }
    ROOT_PAYLOAD {
        ID = "Message 2"
        DESCRIPTOR = ""
    }
    COMPONENTS {
        TOP_LEVEL: 14348740
        14348740:  14340540 13870552
        14340540:  14347972
    }
}
```

The most important portion within a MESSAGE block is the COMPONENTS section. The text therein conveys the structure of a message by indicating which components are anchored off each other. For instance, in the case above we can see that the public key encryption with key $-K_b$ (14348740) is anchored off the root and contains a hash (14340540) and the nonce N_b (13870552). The hash in turn has one child component anchored off it, namely the nonce N_a (14347972). The SENDER and RECIPIENT fields signify the originator and recipient of the message respectively. The communications settings are stored in the MESSAGE block. The ACTIVE_SETTINGS field can be set to self, root or subprotocol. A value of self indicates that the SETTINGS block is used to specify the communications setup, while the root and subprotocol values indicate that the root or subprotocol defaults are used instead. The

message code name is stored in the ROOT_PAYLOAD block and any pre-processing and post-processing associated with the message is stored in the ACTION block. The following text extract reveals the saved definition for the *Foobar* subprotocol included within the *Sample* protocol specification:

```
SUBPROTOCOL {
    EXECUTION_TYPE = "automatic"
    CONDITIONAL_TEXT = ""
    FRIENDLY_NAME = "Foobar Subprotocol"
    USE_FRIENDLY_NAME = "yes"
    FILE_NAME = "Foobar Subprotocol.spr"
}
```

This fragment of text is found between the MESSAGE blocks for the second and third message, as the order of the MESSAGE and SUBPROTOCOL blocks indicates their sequencing. In the above SUBPROTOCOL block the name of the source file associated with the subprotocol is *Foobar Subprotocol.spr*. This file is located in the same directory as the Sample source file, since the path-names of included subprotocol files are always specified relative to the location of the parent's source file. The remainder of the subprotocol fields are self-explanatory and reflect the options presented in the Subprotocol Properties Dialog.

3.5.3 Memory Management

The GYPSIE environment makes use of a reference-count based garbage collector to ensure that memory occupied by Payload objects is correctly referenced and deallocated. The garbage collector that we have developed was based on the description given in [60]. In the context of the GYPSIE environment, a garbage collection system is essential, since there are multiple pointers scattered throughout the system's memory at any given time, many of which reference the same locations in memory. In fact, when using the SPEAR II tool, multiple Payload objects easily arise due to explicit duplication or as a consequence of undo and redo information that is created when High-Level View operations are conducted.

The reference-counting garbage collector that we have developed does not need to search through memory to find objects to delete. Instead, for every object a running count is kept of the number of pointers that reference it. This running count is known as the object's "reference count". Every time a pointer is assigned to reference an object, that object's reference count is incremented. When a pointer is reassigned to reference a different object, the original object's reference count is decremented and the count of the object to which the pointer has been newly assigned is incremented. Also, every time a pointer goes out of scope, the reference count of the object to which it pointed is decremented. An object can only be deleted if there are no pointers referencing it.

To implement the reference-count based garbage collector we created two classes named RGCObj and RGCRef. The RGCObj class contains an integer to store a reference count and three methods named incRefCount(), decRefCount() and canDelete(). To increment or decrement the reference count the incRefCount() and decRefCount() methods are used. To check if the reference count is zero, the canDelete() method, which returns a boolean value, is used. The RGCObj is declared as a virtual public superclass of any class that requires reference-count based garbage collection. However, to perform the actual incrementing, decrementing and deallocation of memory, the RGCRef class is used. The RGCRef class is templated and takes a class descended from RGCObj as a parameter.

To use the garbage collection engine in conjunction with a given class, say class `X`, a programmer first needs to make sure that `RGCObj` is a superclass of `X`. She then needs to create a templated type by passing `X` as a parameter to `RGCRef`. The `RGCRef` class contains overloaded operators to facilitate operations such as dereferencing, assignment, comparison and pointer address calculation. In the code sample that follows we give a practical example of how the garbage collector is used by creating an `Integer` class and then using the `RGCRef` type to create a garbage collected type based on the `Integer` class. We then instantiate two variables of this type and show how they can be used by initializing one of them with an integer value, and then making the other point to and modify the same location in memory. The important point to note in this example is that the memory pointed to by both references is correctly deallocated at the end of the `main()` function, with no memory faults occurring:

```
// Integer is a subclass of RGCObj.
class Integer : public virtual RGCObj {
public:
    Integer(int newValue) { value = newValue; }
    int getValue() { return value; }
    void setValue(int newValue) { value = newValue; }
private:
    int value;
};

// Make declaration with the wrapper class easier.
typedef RGCRef<Integer> PInteger;

void main() {
    PInteger pointer, duplicatePointer;

    // Initialize pointer with 5593879.
    pointer = new Integer(5593879);
    // Make duplicatePointer reference the same memory.
    duplicatePointer = pointer;
    // Change the value of the integer in memory.
    duplicatePointer->setValue(5583565);

    // See the effects of the change.
    cout << pointer->getValue() << endl;
}
```

The technique outlined in this code fragment is employed in the GYPSIE source code. The `Payload` class is used to create a type named `PayloadWrapper` through the use of the `RGCRef` template. An instance of this wrapper class is then embedded inside each `MessageTreeNode`. When creating a `MessageTreeNode`, a `Payload` pointer is passed as a parameter to the constructor and this pointer is then used to initialize the embedded wrapper. Because every `Payload` object contains a reference counter, the wrapper associated with the `MessageTreeNode` will be able to determine when a `Payload` object can be safely deleted. The incrementing and decrementing of the `Payload` object's inherited reference counter is performed by the wrapper. In this way, we can have multiple `MessageTreeNodes` scattered throughout the system's memory. When these nodes are deleted from memory, the associated `Payload` objects will either have their reference count decremented or they will be deleted. `RGCRef` wrappers are also used in the encryption and decryption `Payload`-based classes to store cryptographic keys. This ensures that a key used in multiple locations will not be inadvertently deleted from memory when the encryption or decryption node it is associated with is deallocated.

3.5.4 Functions Provided by GYPSIE

The GYPSIE API is too large to describe in detail due to space constraints. However, we will give some brief details pertaining to portions of the core feature set in order to present an understanding of how the API operates. The ability to add, edit, delete and retrieve canvas components is provided by `DesignController` methods. To add a message to the design canvas, the `addMessage()` function call is used. This method takes a `Message` object and vertical position as parameters. The `editMessage()` method allows one to modify a message by providing the old message, new message and new vertical position as parameters. The `deleteFloater()` function removes the message or subprotocol at the vertical position specified by its parameter. Similar add, edit and delete methods are provided for principal and subprotocol components. To retrieve a given canvas component, the `getFloater()` or `getPrincipal()` methods are used. Obtaining a count of the number of messages, principals or subprotocols can be accomplished with the `getMessageCount()`, `getPrincipalCount()` and `getSubprotocolCount()` methods. Finally, to render a protocol we use the `drawProtocol()` method which renders the current protocol to the canvas specified by the `TPaintBox` object embedded within the protocol's `DesignController` object.

To obtain a flat textual representation of a component, the `getText()` method belonging to the `MessageTreeNode` class is used. If the component represented by the node is non-terminal, then the `getText()` method is applied recursively to all of the child nodes and the resulting text components are concatenated. Any text that needs to be appended to the front or rear of the output is also concatenated to the result before the text is returned. To obtain the textual representation of a given message, the `getText()` method of the root node associated with the message is invoked. The `getLatex()` method of the `MessageTreeNode` class is used to obtain LaTeX output. Within the `Message` class, the `getText()` and `getLatex()` are used to invoke the embedded root node's `getText()` or `getLatex()` method. Text or LaTeX output of the entire specification is obtained using the `getProtocolText()` or `getProtocolLatex()` methods which belong to the `DesignController` class.

The retrieval of components can be carried out by using the `getPayloadsUsedInProtocol()` or `getPayloadsByTypeInProtocol()` methods. Both of these methods return a linked list of all the components used in the current protocol, however, the latter method returns only those components of a specified type indicated by its associated parameters. To initialize the Component View tree, the `initTreeView()` method belonging to the root `MessageTreeNode` object of a message is invoked with the tree-view pointer as a parameter.

Saving and loading operations are conducted by the `DesignController` object associated with a protocol. When saving components, the `saveProtocolFile()` method iterates through the principal, message and subprotocol linked-lists, invoking the `doWrite()` method, which is a member of the `Principal`, `Message` and `Subprotocol` classes. To save the preamble the `saveProtocolFile()` method examines the preamble and writes its settings to disk, while in the case of component data it invokes the `doWrite()` method of each component. The `doWrite()` methods all take the target file as a parameter and produce a specification block therein. A number of methods within the `DesignController` class have been created to read the different specification blocks that exist in a protocol source file. For example, to read a `PREAMBLE` block, the `readPreamble()` method is used. Within the `readMessage()` method, the `readTreeComponents()` method is used to construct the trees which represent a message's structure. When reading components from disk, the `IndexedPayload` method is used to store a `Payload` object along with its integer identifier. The `MessageTreeNode` object within the `IndexedPayload` class is used to temporarily store the node within which a non-terminal component is embedded so that further nodes can be anchored off it.

To determine whether certain SPEAR II operations are permissible, the GYPSIE API includes function calls that can be used to query whether specific conditions are true so that the appropriate interface to a set of engineering operations can be enabled or disabled appropriately. For example, a message rounds calculation should only be conducted when at least one message exists. So, to check whether a message does indeed exist, the `aMessageExists()` method which belongs to the `DesignController` class can be used. To determine whether an undo or redo operation is possible, the `canUndo()` and `canRedo()` methods, which both return a boolean value, can be used. The `getUndoDescription()` and `getRedoDescription()` methods are used to obtain textual descriptions of the undo or redo operation to be carried out next. This text is included in tooltips which appear when hovering over the undo or redo buttons. The `getChangesMade()` method determines whether a protocol specification has been modified by examining a dirty bit. To determine the width and height which a protocol specification takes up on the design canvas we use the `getProtocolWidth()` and `getProtocolHeight()` methods. This information is used to dynamically resizing the canvas when protocol specifications become large.

3.6 Closing Remarks

In this chapter we begun by presenting requirements for a generic security protocol design environment and then followed this up by presenting the GYPSIE environment, which is a realization of these requirements. GYPSIE is a core component of the SPEAR II Framework and is used to create protocol specifications which can be used as input to a wide-range of SPEAR II engineering modules. The GYPSIE environment is composed of three distinct views. The *High-Level Protocol View* describes the overall flow of messages and works in conjunction with the *Navigator View* which presents a concise summary of the structure and contents of a protocol by using a tree-view with expandable and collapsible nodes. The more detailed *Component View* is invoked from the High-Level View and allows one to view and edit the contents of protocol messages, each message being displayed as a hierarchical tree. In effect, the GYPSIE environment acts as a central hub and is the key enabling component of the SPEAR II Framework. Without GYPSIE, no protocol engineering operations could take place within SPEAR II as there would be no protocol specification to analyze or manipulate. Because of this fact, we have tried to make GYPSIE as generic as possible, so that it can produce output that can be easily augmented to cater for a wide array of protocol engineering and analysis operations.

The ultimate value of the SPEAR II Framework lies in the extent to which it can be expanded and integrated with existing and future protocol engineering techniques. Within the current SPEAR II application, three modules work in conjunction with the GYPSIE design environment. *Internal* modules are embedded within the SPEAR II source code and make use of the GYPSIE API to obtain details about a protocol specification. Examples of internal modules that currently exist are the rounds calculator and Visual GNY environment. Both of these modules use API calls to retrieve components, determine types, uncover message flows and obtain the principals involved in a specification. However, in the majority of situations, protocol analysis tools already exist and these tools may have been written in languages other than C++. One of the ways in which such *external* analysis modules can be integrated within the SPEAR II Framework is by piping input into the module in the required format. This approach requires that the GYPSIE API be used to collect, collate and format the protocol specification and any other necessary information. Once this task has been performed, the module can be executed using system calls, the extracted information being transferred through the use of command-line parameters. After execution, the output from the module is retrieved, parsed and then displayed appropriately. This approach has been used to integrate the Prolog-based GYNGER analyzer within the SPEAR II Framework.

There are currently no code generation modules associated with the SPEAR II Framework. However, selected features within the GYPSIE environment indicate how code generation routines can be incorporated within the system. For example, some component properties dialog boxes contain a tabbed pane which can be used to specify a textual ASN.1 specification. This specification is not used for any current engineering operations, but it is still saved with the component. To facilitate the construction of ASN.1 specifications, we envisage the creation of a graphically-based ASN.1 specification environment. This environment would help designers to specify a component's structure so that appropriate encoding and decoding routines can be constructed and included in any generated source code. The GYPSIE environment also includes a Communications Settings Dialog that can be used to specify the communications properties associated with given message, principal or protocol. These settings are minimal at present, but give an impression of how this type of information can be requested. SPEAR I included a simulation module that allowed a protocol engineer to step through a protocol execution and view the progression of BAN beliefs. However, we envisage the creation of a more powerful attack analysis module based on the Interrogator [24] that would work in conjunction with the GYPSIE environment and try to determine replay attacks that would reveal secret information during a protocol session.

The GYPSIE High-Level View is very similar to the SPEAR I design environment and uses the same formalisms and user-interaction style. However, a significant difference is that GYPSIE includes the ability to add subprotocols to a specification. To aid in visualizing a specification and its contents, GYPSIE includes the Navigator View and the Component Tracker. The SPEAR I design environment includes no such aids. The undo and redo feature is a powerful addition to the GYPSIE environment and frees users to experiment with a specification and recover from accidental errors. SPEAR I does not include any such error-recovery mechanism. The Component View presents a designer with a detailed overview of the structure and composition of a given message using a hierarchical tree-view, allowing one to conduct modifications through drag-and-drop operations, pop-up menus and specially tailored dialog boxes. However, in the case of SPEAR I, the message structure and composition is specified textually and the components themselves must all be defined using dialog boxes prior to being added to a given message. The SPEAR I system incorporates a code generation module which produces rudimentary Java code. As we have seen, the SPEAR II environment does not yet contain a code generation module. However, API calls are in place to allow one to be incorporated within the system. In fact, the GYPSIE API was used to incorporate a rounds calculator, an export subsystem which generates textual, LaTeX and Prolog output and a GNY analysis environment within the SPEAR II Framework.

The CAPSL [28] specification language is intended to support the creation of cryptographic protocol specifications which can be examined by formal analysis tools. Interoperability with CAPSL would be of tremendous benefit to SPEAR II, since it could well become the standard language in which protocol specifications are modelled. In fact, the appeal of using GYPSIE would improve significantly if it could be used to create syntactically correct CAPSL specifications, since the process of creating a textual CAPSL specification would be transformed into one of merely creating a specification through drag-and-drop operations and dialog box completion. Also, no graphically-based tools to aid in the construction of CAPSL specifications currently exist. At present, the SPEAR II file format uses our own specially-devised syntax. However, as a possible upgrade we are considering the incorporation of CAPSL import and export functionality within the GYPSIE environment. However, this upgrade would require some thought as a given CAPSL specification does not map directly to a GYPSIE specification. There are portions of a specification which both environments will have in common, however the difficultly lies in resolving the differences they might have. For example, a potential conflict could arise when importing or exporting a specification since CAPSL provides both associative and non-associative component concatenation, while GYPSIE only provides associative concatenation.

The Software Through Pictures (StP) [42] OMT environment serves as a front-end to the Higher Order Logic analyzer that is part of the Convince toolkit [50]. In a similar vein, GYPSIE serves as a front-end to the SPEAR II Framework and currently creates input for a rounds calculator and Prolog-based GNY analyzer. However, there are significant differences in how these two systems operate and interact with users. The StP environment is a full-featured OMT modelling tool and can be used to create specifications for a diverse range of applications. On the other hand, the GYPSIE environment is tailor-made for creating cryptographic protocol models. Thus, a user wishing to employ the StP environment to produce cryptographic protocol models must be familiar with *both* OMT and cryptographic protocol modelling concepts. However, a GYPSIE user need only understand the issues related to cryptographic protocol modelling since there is an almost direct mapping between the interface and protocol engineering concepts. In effect, the StP environment forces one to manually make an association between protocol modelling concepts and the OMT syntax and notation in order to represent the flow of messages. A short-coming of StP is that it does not feature built-in functionality for typing protocol components. On the other hand, every GYPSIE component has an associated type which is derived from cryptographic protocol theory. In effect, StP forces users to create annotations to specify a component's type, while GYPSIE considers typing as an integral part of a specification, which is necessary to correctly perform certain analyses and code generation. Both StP and GYPSIE can produce output for a variety of protocol engineering tools, StP producing ISL [15] for use with a HOL-based GNY analyzer, and GYPSIE producing text, LaTeX and Prolog output for use with the GYNGER GNY analyzer.

A challenge that confronts users of the GYPSIE environment, and perhaps all protocol design environments in general, is that as a protocol increases in size it becomes more difficult to model and manage, since mistakes can creep into the specification more easily while it is being created. To facilitate accurate protocol construction and verification, the Component View displays a hierarchical tree-like representation of a message, using icons to represent each component type and expandable and collapse nodes to represent containers such as encryptions. The High-Level View lists all of the principals involved in a specification and illustrates the flow of messages using a formalism similar to Message Sequence Charts [44]. Furthermore, to aid in validating a protocol model, a text or LaTeX representation of the protocol can be generated through the High-Level View so that a designer can print the the specification and examine it off-line. A limitation of the GYPSIE environment is that it does not cater for the modelling of multi-cast protocols, since each message can only have one recipient. This current lack of functionality is due to the fact that GYPSIE was created with the aim of providing specifications for GNY analysis and rounds calculations. Thus, when a code generator is integrated within the SPEAR II Framework, the GYPSIE environment will have to be upgraded to incorporate multi-cast mechanisms.

We conducted a number of experiments to examine how individuals who had been lectured in security protocol modelling techniques interacted with the GYPSIE environment. The primary goal of these experiments was to determine whether the GYPSIE environment facilitates accurate protocol modelling. We also wanted to examine how much time individuals took to specify security protocols. One of our primary observations was that the effectiveness of the GYPSIE environment depends to a large degree on the cryptographic expertise and fluency already possessed by a user. In order to create cryptographic protocols, an individual should have an understanding of the underlying theory of security protocol design. However, even with this requirement, we found that every one of the participants was able to complete three diverse protocol specifications in a reasonable amount of time, each specification being completed in under ten minutes on average. The error level was also reasonably low, with 75% of the resultant sixty specifications being completely error-free. The comments from participants were favourable and no major problems were encountered. Judging from the results that we obtained we feel that GYPSIE will be able to simplify, enable and facilitate the construction of high-quality protocol specifications.

Chapter 4

GNY-Based Protocol Analysis

"A formal method that tries to cover all the features of cryptographic protocol analysis is like a Swiss Army knife — not a terribly good instance of any of the tools it contains."

— Roger Needham, Quoted in "On Unifying Some Cryptographic Protocol Logics"

Analysis methods for cryptographic protocols have predominantly focused on detecting *information leakage*, rather than determining whether a protocol attains its stated goals. However, security protocols often fall short of achieving their intended objectives, usually for very subtle reasons [2]. As a result of this fact, cryptographic logics have been developed to aid in determining whether protocols actually fulfil their intended goals [32]. Using logics to analyze security protocols has a number of advantages:

- The use of logics forces protocol designers to explicitly state the security assumptions which they have made and will require after the protocol has executed.
- Reasoning with logics makes designers think about the use for which each component is intended, thus minimizing redundancy.
- Cryptographic logics can also be used to explicitly bind the evolution of beliefs in a protocol session to message contents, number of messages and message rounds, thus helping to determine the minimum number of messages required to achieve a given set of beliefs and possessions.

The BAN modal logic [1] popularized the notion of using logics to detect flaws and redundancies in protocols. It has been labelled as a success by many commentators [31, 56, 23] and has been used to find flaws in several protocols, including Needham-Schroeder [63] and CCITT X.509 [21]. BAN has also been used to uncover redundancies in well-known protocols such as Kerberos [59] and Otway-Rees [64] and many published papers use BAN to make claims about their protocol's security.

BAN spawned the creation of a number of related logics, each of which has tried to improve on or add to its underlying premises. Popular descendants of BAN include GNY [36, 33], AT [3], VO [81] and SVO [79]. Each of the logics descended from BAN has its own advantages, disadvantages and vested interests. The aim of this chapter is to introduce the GNY modal logic. We will sketch the fundamental assumptions underlying GNY and introduce definitions and concepts from [36, 33]. Thereafter we will describe the inference rules, give hints for determining protocol goals and discuss modifications which have been made to the GNY postulate set. Finally, we work through two example analyses, showing how useful results can be achieved. The chapter then concludes with some closing remarks.

4.1 Model of Computation

A distributed environment consists of independent entities called *principals*. Depending on the context, they can either be users, processes, computers, or any entity that wishes to communicate with another. Principals are connected by network links and these constitute the only means of communication. Communication occurs through the exchange of messages on these links. A principal can place a message on any link and may also observe and alter messages being passed along any link.

A protocol is a *distributed algorithm* which determines the messages a principal should communicate as a function of his internal state. Protocols are divided into *stages* by message transmissions. A *session* is an execution of a protocol. Principals participate in a session with certain initial beliefs and initial possessions. From this point, they can obtain new beliefs from current beliefs and incoming messages. Similarly, principals can increase their possessions by receiving messages or generating new message components. Inference rules presented later in this chapter govern the derivation of new beliefs and possessions.

A principal's state consists of two sets. A *belief* set includes beliefs of the principal which reflect his view of the state of the physical world and other principals. The state of the world records the occurrence of events such as the originator and time at which a certain message was sent. A *possession* set includes information available to the principal from which he can derive new beliefs and construct new messages. Beliefs and possessions are *monotonic* within a session. This means that if a belief or possession is a member of its respective set at any phase of the session, then it remains in that set at every subsequent phase of that session. However, no such claim is made across sessions. A principal's belief or possession in a session will not necessarily be a member of the corresponding set in past or future sessions in which the principal has or will participate.

The only universal assumption that we require is that secrets which are used for identification purposes are not discovered or used maliciously by untrusted principals. Since principals will be extremely limited in attaining useful beliefs without such a basic trust, it is a convenience to make this assumption default. However, if it was so desired, we could let principals choose, as part of their initial beliefs, whether to trust each other in this respect.

4.2 A Protocol Description Language

In this section the basic notions underlying the GNY reasoning process will be introduced. As has already been mentioned, principals exchange messages during the execution of a protocol. A protocol may be represented as a finite sequence of n messages:

$$
\begin{aligned}
&(1)\ P_1 \longrightarrow Q_1 : X_1 \\
&(2)\ P_2 \longrightarrow Q_2 : X_2 \\
&\qquad \vdots \\
&(3)\ P_n \longrightarrow Q_n : X_n
\end{aligned}
$$

In this notation, P_i is the sender, Q_i is the intended recipient, and the formula X_i is the message body. During the execution of a protocol, principals communicate messages in the order in which they are listed. However, in practice, principals may send some messages simultaneously and the transmission of these messages may overlap in time.

4.2.1 Formulae

A *formula* in a protocol description is a name referring to a bit string which would have a particular value in a session. This is analogous to a variable identifier in a programming language. Let X and Y range over formulae. Two kinds of special formulae, shared secrets and encryption keys, are denoted as S and K respectively. The following are also formulae:

- (X, Y): The conjunction of two formulae. Conjunctions represent sets and have properties such as associativity and commutativity.
- $\{X\}_K$ and $\{X\}_K^{-1}$: Conventional encryption and decryption of X, assuming that the cryptosystems are resistant to ciphertext-only and known-plaintext attacks. The encryption (or decryption) depends on the plaintext (or ciphertext) and key in such a way that any change to plaintext (or ciphertext) or the key causes an apparent random change in the ciphertext (or plaintext). It is assumed that $\{\{X\}_K\}_K^{-1} = X$ is always satisfied, but that $\{\{X\}_K^{-1}\}_K = X$ is not necessarily satisfied.
- $\{X\}_{+K}$ and $\{X\}_{-K}$: Public key encryption and decryption of X. A public-key cryptosystem satisfies the requirements stated for a conventional cryptosystem. Public key applications that are used for key exchange satisfy $\{\{X\}_{+K}\}_{-K} = X$, while public-key cryptosystems that are applicable to digital signatures satisfy $\{\{X\}_{-K}\}_{+K} = X$.
- $H(X)$: A one-way function of X. Given X it is computationally feasible to compute $H(X)$, however, given $H(X)$, it is infeasible to compute X. It is also infeasible to find an X and X' such that $H(X) = H(X')$. This defines a strong one-way hash function.
- $F(X_0, X_1, \ldots, X_{n-1})$: A many-to-one computationally feasible function, such that for any X_i, where $0 \leq i \leq n-1$, and constants $C_0, C_1, \ldots, C_{n-1}$, $F(C_0, \ldots, C_{i-1}, X_i, C_{i+1}, \ldots, C_{n-1})$ is a one-to-one computationally feasible function, and its inverse is also computationally feasible. As a special case $F(X)$ denotes a feasible one-to-one function whose inverse is also feasible.
- $< S >$: Denotes that the secret S is used for *identification* purposes. Thus, it can be distinguished when other secrets are included merely as data in the same message or computation.

Principals often exchange formulae to express their current beliefs. Beliefs are described by statements, introduced in the next section. Let C range over all statements. The following is also a formula:

- $X \rightsquigarrow C$: A formula with an *extension*. Statement C is the extension and is considered an integral part of the formula. By definition, $X \rightsquigarrow C_1 \rightsquigarrow C_2$ and $(X \rightsquigarrow C_1) \rightsquigarrow C_2$ are equivalent to $X \rightsquigarrow (C_1, C_2)$.

Essentially, an extension to a formula is a formal specification which dictates that a principal should proceed to send a formula only if certain conditions hold. A formal specification such as this helps to eliminate ambiguity as these conditions are often only expressed verbally in traditional protocol specifications. Having accepted that a formula is genuine, the recipient can choose to believe that the formula's extension holds, if he trusts the sender's competence and honesty. Without extensions, principals can draw conclusions about the physical world. However, with extensions principals may be able to draw conclusions about beliefs held by other principals.

4.2.2 Statements

A basic *statement* reflects some property of a formula, typically reflecting a relation between a principal and a formula. As a convention, "believes" means "believes or is entitled to believe" and "possesses" means "possesses or is capable of possessing". Let P and Q range over principals. The following are statements:

- $\emptyset$: This is an empty statement and does not denote anything except that there could have been a non-empty statement in its place. The formulae X and $X \rightsquigarrow \emptyset$ are equivalent.
- $P \lhd X$: P *is told* a formula, X, possibly after performing some computation, such as decryption. Thus, the formula being told is itself, or some computable content thereof.
- $P \lhd *X$: P is told a formula, X, which he is *not* the first to convey in the current session of the protocol, though he could have transmitted it in a previous session. *Also*, it is the first time that P receives X in the current session.
- $P \ni X$: P *possesses* formula X. P is able to repeat this formula in future messages of the current session. At a particular stage of a session, P possesses all the formulae that he has been told, all the formulae he started the session with, and all the ones that he generated during the current session. In addition, P possesses all the formulae that are computable from formulae he already possesses.
- $P \mid\sim X$: P *once conveyed* formula X. X can be a formula explicitly exchanged or some computable content of a formula. Thus, a formula can also be exchanged implicitly.
- $\sharp(X)$: Formula X is *fresh*. A principal should believe that a formula originated by another principal is fresh if it has been constructed after the occurrence of some fresh event. A principal believes anything he has originated to be fresh if he cannot have chosen the same formula for the same purpose before.
- $\phi(X)$: Formula X is *recognizable*. A principal would believe X to be recognizable if he has certain expectations about the value or structure thereof. He may recognize a particular value, a particular structure or other forms of redundancy. In either case, he may not possess part or all of the formula.
- $P \overset{S}{\longleftrightarrow} Q$: S is a suitable *secret* for P and Q. These entities may use S to prove their identities to each other. They may also use it as, or derive from it, an encryption key K to communicate, denoted as $P \overset{K}{\longleftrightarrow} Q$. This notation is symmetrical. Thus $P \overset{S}{\longleftrightarrow} Q$ and $Q \overset{S}{\longleftrightarrow} P$ can be used interchangeably.
- $\overset{+K}{\mapsto} Q$: $+K$ is a suitable *public key* for Q. The matching secret key is given by $-K$.

As we have mentioned, the only default assumption which we require is that S, K or $-K$ will never be discovered by any principal except the legitimate owners or principals which the owners trust. In the latter case, the trusted principals should never use S, K or $-K$ as a proof of identity or as an encryption key to communicate. Continuing, the following are also statements:

- $P \propto X$: P is eligible to convey formula X. P holds the relevant possessions and beliefs. This notation is used to detect inconsistencies in the protocol description.

- $P \dashv (X)$: P is *not the first* principal to originate formula X. This formula must first be generated and conveyed by another principal.

Statements are often associated with individual principals to specify their states. Let C range over statements. The following are also statements:

- (C_1, C_2): The conjunction of two statements. Conjunctions represent sets and have properties such as associativity and commutativity.
- $P \mid\equiv C$: P believes that statement C holds. $P \mid\equiv$ is considered an empty statement.
- $P \mid\equiv Q \mid\Longrightarrow C$: P believes that Q has *jurisdiction* over statement C. He believes that Q has authority on C and should be trusted in this respect.
- $P \mid\equiv Q \mid\Longrightarrow Q \mid\equiv *$: P believes that Q has *jurisdiction* over all his beliefs. P considers Q to be competent and honest.

Trust can be defined at multiple levels in order to derive weaker or stronger conclusions. For example, if honesty of a principal implies that he maintains the correspondence between the contents of a formula and the extension during conveyance, then the conclusion that $P \mid\equiv Q \mid\sim Q \mid\equiv C$ can be obtained. Since Q may not be competent, this represents a lower level of trust and the conclusion is weaker than $P \mid\equiv Q \mid\equiv C$. Moreover, we can specify different levels of trust for particular aspects of a principal's behaviour. This reflects the fact that often a principal is trusted differently with respect to the different tasks in which he engages.

4.2.3 Operational Semantics

The following operational semantics are similar to those in BAN [1]. Principals develop new beliefs and accumulate possessions by applying computationally tractable inference rules to their current beliefs, possessions and received messages.

The *local state* of a principal P consists of two sets, viz. a set of formulae $\mathcal{P}_P$ and a set of statements $\mathcal{B}_P$. Intuitively, $\mathcal{P}_P$ is the set of formulae the principal possesses and $\mathcal{B}_P$ is the set of beliefs the principal holds. These sets have some closure properties, as will be illustrated by the inference rules.

A *global state* is a tuple containing the local states of all principals. Suppose s is a global state. Then s_P is the local state of P in s and $\mathcal{P}_P(s)$ and $\mathcal{B}_B(s)$ are the corresponding sets of possessions and beliefs. The *satisfaction* relation between global states and statements is:

- $P \mid\equiv C$ holds in a state s if $C \in \mathcal{B}_P(s)$.
- $P \ni X$ holds in a state s if $X \in \mathcal{P}_P(s)$.
- A set of statements holds in a given state if each of its members holds.

A *protocol session* is defined as a finite sequence of states $s_0, \ldots, s_n$ where $\mathcal{B}_B(s_i) \subseteq \mathcal{B}_B(s_{i+1})$ and $\mathcal{P}_P(s_i) \subseteq \mathcal{P}_P(s_{i+1})$ for all $i \leq (n-1)$. That is, the belief and possession sets are *monotonic* and do not decrease in size during a session.

As described previously, a protocol is a finite sequence of n expressions of the form:

$$(P_1 \longrightarrow Q_1 : X_1),\ (P_2 \longrightarrow Q_2 : X_2),\ \ldots, (P_n \longrightarrow Q_n : X_n)$$

Now X_i, where $i \leq n$, is normally a conjunction of formulae and certain formulae embedded within X_i will contain extensions. Let C_{ij} be an extension attached to the j^{th} formula within X_i and assume that there are m_i formulae within X_i, where $m_i \geq 1$. During a session, all of the messages within the protocol are communicated so that $X_i \in (\mathcal{P}_{Q_i}(s_i) \cap \mathcal{P}_{P_i}(s_{i-1}))$ and $C_{ij} \in \mathcal{B}_{P_i}(s_{i-1})$ for all $i \leq n$ and $j \leq m_i$.

A protocol analysis consists of annotating the protocol with statements and manipulating these statements with the aid of the inference rules. An annotation for the protocol *holds* in a session if all the statements in the annotation hold in their corresponding states. An annotation is *valid* if it holds in all sessions of the protocol. The first set of the annotation is the initial assumptions and this must be the first set to hold.

4.3 Protocol Parsing

Security protocols are normally described by listing the messages sent between principals and symbolically showing the source, destination and the contents of each message. However, this conventional notation is not convenient for manipulation with a logic, since we wish to attach exact meanings to each component of each message and these meanings are not always apparent from the data contained in the messages. Thus, the aim of the protocol parser is to transform a protocol description into a form that is more suitable for manipulation and analysis. The parser examines all the lines of the form $P \longrightarrow Q : X$, scanning from the beginning of the list to the end:

1. If $Q = P$ an error is reported.
2. For each complete Y *using a shared secret* in a line of the form $P' \longrightarrow Q' : X'$, if:
   - Y does not appear in a line $Q' \longrightarrow P'' : X''$, *and*
   - A line $P'' \longrightarrow Q' : X''$ containing $*Y$ does not already appear.

   Then the parser inserts a star before Y.

The following example gives a demonstration of the protocol parser at work. The conventional protocol specification is on the left, and the parsed output on the right.

(1)	$A \longrightarrow B : \{X\}_K$		(1)	$A \longrightarrow B : *\{X\}_K$
(2)	$A \longrightarrow C : \{X\}_K$		(2)	$A \longrightarrow C : *\{X\}_K$
(3)	$C \longrightarrow A : \{X\}_K$	*is transformed to*	(3)	$C \longrightarrow A : \{X\}_K$
(4)	$C \longrightarrow B : \{X\}_K$		(4)	$C \longrightarrow B : \{X\}_K$
(5)	$B \longrightarrow E : \{X\}_K$		(5)	$B \longrightarrow E : *\{X\}_K$

Notice that the parser 'detects' possible backward and forward replays in the current session of the protocol run, as seen in messages (3) and (4) respectively. Possible replay attacks are not tagged with a star.

4.4 Inference Rules

GNY inference rules govern the reasoning about principals' beliefs and formulae which they possess. These rules do not claim properties such as completeness and new rules can be introduced where appropriate. There are no negations of any kind and thus contradicting conclusions cannot be reached unless the initial assumptions are self-contradicting. All inference rules have the form:

$$N.\ \frac{C_1, C_2, ..., C_n}{K_1, K_2, ..., K_m}$$

Essentially, this says that if statements $C_1, C_2, \ldots, C_n$ hold then so do $K_1, K_2, \ldots, K_m$, where $n \geq 1$ and $m \geq 1$. N is the inference rule number. An inference rule that applies to formula X also applies to formula $*X$. In other words, substituting $*X$ for X everywhere in an inference rule yields another inference rule. The reverse is not necessarily valid. The complete set of GNY inference rules is listed in Appendix A. In the subsections that follow we describe the salient features of the eight GNY rule sets.

4.4.1 Being-Told Rules

Being-told inference rules deal with the formulae which a principal receives. Every formula which a principal receives, as well as certain manipulations of that formula, are regarded as *being told* to that principal. The following are some examples of being-told inference rules:

T4. $\dfrac{P \lhd (X, Y)}{P \lhd X}$

If P is told a formula, then he is also told each of its cojoined components.

T5. $\dfrac{P \lhd F(X_0, X_1, ..., X_{n-1}), P \ni (X_0, ..., X_{i-1}, X_{i+1}, ..., X_{n-1})}{P \lhd X_i}$

If P is told the result of an n-ary function F and he possesses at least $n-1$ of the arguments, then he is also considered to have been told the missing argument.

T6. $\dfrac{P \lhd \{X\}_K, P \ni K}{P \lhd X}$

If P is told a formula encrypted with a key he possesses, then he is also considered to have been told the decrypted contents of that formula.

4.4.2 Possession Rules

Inference rules in this category specify the formulae a principal is capable of possessing by manipulating formulae which he already possesses. Examples of these inference rules include:

P1. $\dfrac{P \lhd X}{P \ni X}$

P possesses a formula which he is told.

P6. $\frac{P \ni X}{P \ni H(X)}$

P possesses a one-way hash function of a formula which he possesses.

P8. $\frac{P \ni X, P \ni +K}{P \ni \{X\}_{+K}}$

If P possesses a formula and a key, then he possesses the encryption and the decryption of the formula with the key. A very important point to note is that the rules P3, and P6 through to P9 can be applied indefinitely to produce what may seem quite trivial new possessions. In the case of automated analysis tools, such a situation may result in needless possessions being generated and may also result in the analyzer not terminating.

4.4.3 Eligibility Rules

Eligibility inference rules determine what formulae a principal is eligible to convey according to the formulae he possesses and the beliefs he holds. The following is a sample of some eligibility inference rules:

E1. $\frac{P \ni X}{P \propto X}$

P is eligible to repeat in future messages a formula which he possesses.

E2. $\frac{P \propto X}{P \propto *X}$

If P is eligible to convey a formula, then he is eligible to convey the formula with a preceding star. Recall that the protocol parser decides whether to insert a star in front of a formula based on the formula's position in the protocol description. Thus, a principal's being eligible to convey $*X$ does not necessarily mean that he is the first to convey the formula in the protocol.

E6. $\frac{P \propto X, P \ni (S, +K), P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \overset{+K}{\mapsto} Q, P \mid\equiv C}{P \propto \{X, <S>\}_{+K} \rightsquigarrow C}$

If P is eligible to convey a formula, he possesses a secret and believes it to be suitable for him and Q, he also possesses Q's public key, then he is eligible to convey the formula cojoined with the secret, encrypted with the public key, and with an extension to describe some of his beliefs.

4.4.4 Recognizability Rules

Inference rules in this category specify the formulae which a principal can believe to be recognizable, given his beliefs about the recognizability of other formulae. A principal believes a formula to be recognizable if he has an expectation of the value or structure of the formula. In either case, he may not possess part or all of the formula. In other words, during an execution he may not know the value of part or all of a formula before actually receiving it. All of the recognizability inference rules can be understood in terms of two basic principles which are illustrated by the following inference rules:

$$\textbf{R3}''.\ \frac{P \mid\equiv \phi(X),\ P \ni F(X)}{P \mid\equiv \phi(F(X))} \qquad \textbf{R10}.\ \frac{P \mid\equiv \phi(X),\ P \ni X,\ P \ni H(X)}{P \mid\equiv \phi(H(X))}$$

These two inference rules essentially state that a given formula can be recognized by a principal if it can be transformed into a recognizable formula or a recognizable formula can be transformed into it. The basic premise underlying R3″ is that a given formula constructed using an invertible function is recognizable if the result of applying the function's inverse to this formula is also recognizable. The premise underlying R10 is slightly different and implies that a formula constructed from a one-way function is recognizable if the application of the same one-way function to a recognizable formula produces a result identical to the formula in question. The fundamental difference between these two inference rules is that in R3″ no possession of a recognizable formula is implied, whereas R10 requires the possession of a recognizable formula for comparison purposes. Inference rules R4 through to R9 are all based on R3″ and R10 — R5, R6, R7 and R11 being based on R10 and R4, R8 and R9 being based on R3″.

4.4.5 Freshness Rules

Freshness inference rules specify the formulae which a principal can believe to be fresh, given his beliefs about the freshness of other formulae. A principal believes a formula to be fresh if it could only have been constructed after the occurrence of some fresh event. Essentially this means that it could only have been created by the application of some transformation to another fresh formula. A principal believes anything he has originated to be fresh if he could not have chosen the same formula for the same purpose before. As was done in Section 4.4.4, we can find two inference rules on the basis of which the other inference rules can be explained. These two inference rules are:

$$\textbf{F3}''.\ \frac{P \mid\equiv \sharp(X),\ P \ni F(X)}{P \mid\equiv \sharp(F(X))} \qquad \textbf{F4}.\ \frac{P \mid\equiv \sharp(X),\ P \ni X,\ P \ni H(X)}{P \mid\equiv \sharp(H(X))}$$

The postulates F5, F9 and F10 are based on F3″, while F6, F7, F8 and F19 are based on F4. Now, we will explain why in F13 through to F16 we have the recognizability requirement. Consider that F13 can be simplified using R4 to yield the following:

$$\textbf{F13}.\ \frac{P \mid\equiv \phi(\{X\}_K),\ P \mid\equiv \sharp(K)}{P \mid\equiv \sharp(\{X\}_K)}$$

Thus, F13 essentially states that if $\{X\}_K$ is believed to be recognizable and the encryption key K is believed to be fresh, then $\{X\}_K$ can also be considered as fresh, since its creation could only have occurred *after* the creation of the fresh key, K. The same reasoning can be applied to F14 using R5, F15 using R6 and F16 using R7. Furthermore, we can simplify F17 using R9 to obtain the following:

$$\textbf{F17}.\ \frac{P \mid\equiv \phi(\{X\}_{-K}),\ P \mid\equiv \sharp(-K)}{P \mid\equiv \sharp(\{X\}_{-K})}$$

Again, this means that if $\{X\}_{-K}$ is received it can be recognized and since the encryption key $-K$ is believed to be fresh, it can immediately be concluded that the encryption is fresh, since it could only have been created *after* the creation of the key, $-K$. We can derive a similar result using F18 and R8.

4.4.6 Conveyance Rules

Inference rules in this category govern the advancement of a principal's beliefs about other principal's states by examining formulae he is told. The freshness requirement in C1 through to C5 and C15 prevents reasoning about non-fresh formulae since they may be a replay during an execution. For reasoning about non-fresh formulae, C10 through to C14 and C16 replace the freshness requirement with the requirement that the recipient knows that he is not the first to originate a formula.

We can simplify (C1, C10), (C2, C11), (C4, C13), (C5, C14), C6 and (C15, C16) by applying the applicable recognizability inference rule, these being R4, R5, R9, R6, R8 and R4 respectively. For example, C2 can be simplified to the following using R5:

C2. $$\frac{P \lhd *\{X\}_K^{-1} \rightsquigarrow C, P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \sharp(X, K), P \mid\equiv \phi(\{X\}_K^{-1})}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K^{-1} \rightsquigarrow C, P \mid\equiv Q \ni (X, K)}$$

C10 can be simplified to the following using R4:

C10. $$\frac{P \lhd *\{X\}_K \rightsquigarrow C, P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \phi(\{X\}_K), P \mid\equiv P \dashv (\{X\}_K)}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K \rightsquigarrow C}$$

And, C6 is simplified to the following using R8:

C6. $$\frac{P \lhd *\{X\}_{-K} \rightsquigarrow C, P \mid\equiv \overset{+K}{\mapsto} Q, P \mid\equiv \phi(\{X\}_{-K})}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_{-K} \rightsquigarrow C}$$

In C7, the added premise that $P \mid\equiv \sharp(X, -K)$ ensures that the conclusion $P \mid\equiv Q \ni (X, -K)$ results. Finally, in laws C3 and C12 the recipient can check the authenticity of the hash by simply rehashing the components X and S. C17 enables splitting of message extensions, which are essentially conjunctions of one or more beliefs, while C8 enables dropping of message extensions completely. Similarly C9 enables splitting of conveyed formulae. Finally, C18 enables principals to derive beliefs about others' possessions based on what they have conveyed, and C19 enables splitting of formula conjunctions which a principal believes that someone else possesses.

4.4.7 Jurisdiction

Inference rules in this category reflect how different levels of mutual trust among principals affects their reasoning about beliefs held by other principals. J2 and J1 are used to determine and to take on the beliefs held by principals partaking in a protocol. J2 allows principal P to believe that principal Q believes in the validity of a statement contained in an extension to a formula which P believes that Q conveyed. For this inference rule to be applied, P must believe that Q is honest and competent and that the formula with the attached extension is fresh. Once P believes that Q has confidence in the validity of a statement, he can also believe in the validity of that statement if he believes that Q is an authority on the matter. This is expressed by J1.

4.4.8 Rationality Rules

Two rationality inference rules supplement those already introduced and help to reason about a principal's beliefs regarding the states of principals including himself.

Firstly, $\frac{P \mid\equiv P \mid\equiv C}{P \mid\equiv C}$. Also, if $\frac{C_1}{C_2}$ is an inference rule, then so is $\frac{P \mid\equiv C_1}{P \mid\equiv C_2}$.
Essentially, the rationality inference rules represent the view that some or all of the principals are rational in advancing their beliefs. For example, the following inference rule can be derived from P9:

$$\frac{Q \mid\equiv P \ni X, Q \mid\equiv P \ni -K}{Q \mid\equiv P \ni \{X\}_{-K}}$$

If Q believes that P possesses a formula and a secret key, then Q believes that P possesses the decryption of the formula with the key. However, note that the second rationality rule can be applied indefinitely to produce useless inference rules.

4.5 Determining Final Conditions

In order to perform a GNY analysis, a designer must know what goals the protocol under inspection should be expected to achieve [12]. There are two broad classes of security protocols that we will discuss. An authentication protocol verifies the identities of participants and then ensures that they agree on an encryption key for later use. Information exchange protocols seek to securely transfer information between participants, ensuring that authentication of the sender, non-repudiation and integrity are achieved. In this section we will present the recommended goals for authentication and information exchange protocols. With these goals in place, a GNY analysis will essentially involve a designer determining the class of the protocol that he wishes to analyze, and then ensuring that these goals are fulfilled. Using these goals, an analyzer can also make recommendations as to what the initial conditions for an analysis should be in certain cases.

4.5.1 Authentication Protocols

As a basis for this discussion we have used the ideas presented in [1, 35]. Authentication protocols are normally run as precursors to communication sessions which require confidentiality. Once an authentication protocol has executed, the participants should be convinced of each other's identity and share a session key which can be used to encrypt formulae to be transmitted [1]. Authentication protocols fall into two categories [35]. Protocols in the first category can be viewed as *only* accomplishing authentication, ensuring that each participant has a session key which he believes to be suitable. A protocol in the second category uses handshakes between the participants so that they can be convinced among themselves that each of them has successfully received the key and believes in its suitability. So, there are two primary successive phases that take place in an authentication protocol, the last phase being optional:

1. During the key exchange phase, principals exchange or negotiate a session key that will be used for further communication. This exchange can be arbitrated by an authentication server which shares a symmetric key with each of the principals. If an authentication server is not used, then public-key cryptography can be used to exchange a key or key components selected by one or both of the principals. If an authentication server is used, then three possibilities exist [35]:

   (a) The authentication server chooses the key and then distributes this key to each of the clients.

   (b) One client chooses the key and then relays it to the other client via the authentication server.

(c) Both clients choose the key by each selecting temporary keys and then constructing the final key from these two components, which are exchanged via the authentication server.

In a similar vein, if public-key cryptography is used, then either one client can choose the session key, or both can be involved. Once this phase of an authentication protocol has been completed, both clients will possess the session key and should believe in its suitability.

2. During the handshake phase, the principals ensure among themselves that each of them has successfully received the session key. A handshake can be one-way or two-way. In a one way handshake only one principal declares that he has successfully received the key. However, in a two-way handshake both principals declare this fact to each other. The following are examples of handshakes that use timestamps:

   - A one-way handshake in which A declares to B that he has successfully received the session key K.

     $(i) \quad A \longrightarrow B: \ \{A, B, T_a\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B$

   - A two-way handshake in which A and B declare to each other that they have successfully received the session key K.

     $(i) \quad A \longrightarrow B: \ \{A, B, T_a\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B$

     $(i+1) \quad B \longrightarrow A: \ \{B, A, T_b\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B$

   Handshakes that use nonces require an extra message when compared to those that use timestamps [35]. Often the first message in a handshake can be piggy-backed on a prior message. Once the handshake phase has completed, the principals who received a handshake should be convinced that the originator thereof possesses the key and that he believes in the suitability of the key.

The goals that we have referred to above can be formalized in GNY notation as eight end-conditions. Authentication protocols that use two-handshakes should achieve all of these goals, while those that use only a one-way handshake should achieve (1), (2) and either (3a) and (4a) or (3b) and (4b). Protocols that do not use handshakes must achieve at least (1) and (2). Consider an authentication protocol in which P and Q participate and the session key to be distributed is K. The recommended final beliefs and possessions appear below:

(1a) $P \ni K$; (2a) $P \mid\equiv P \overset{K}{\longleftrightarrow} Q$; (3a) $P \mid\equiv Q \ni K$; (4a) $P \mid\equiv Q \mid\equiv P \overset{K}{\longleftrightarrow} Q$

(1b) $Q \ni K$; (2b) $Q \mid\equiv P \overset{K}{\longleftrightarrow} Q$; (3b) $Q \mid\equiv P \ni K$; (4b) $Q \mid\equiv P \mid\equiv P \overset{K}{\longleftrightarrow} Q$

Possession (1) and belief (2) are derived from phase 1, while beliefs (3) and (4) are derived from phase 2. The being-told and possession laws can typically be used to derive (1), while conveyance laws and extensions can be used to derive (2). In this case, the recipient of a message which conveys trust in the key must believe that the sender is honest and competent, and that he has jurisdiction over the suitability of encryption keys. Belief (3) can be obtained through the application of conveyance laws to the handshake. For belief (4) to be derived, the sender of the handshake must attach an extension to the formulae encrypted with the exchanged session key. This extension must reveal his trust in the suitability of the key. Then, the conveyance and jurisdiction laws can be used to derive (4). Note that the recipient of the handshake must believe in the honesty and competence of the handshake sender. Throughout the construction of an authentication protocol, the designer must keep in mind that the contents of encrypted formulae should be fresh and recognizable to the recipient.

4.5.2 Information Exchange Protocols

For this study we define information exchange protocols to be the mechanism by which principals exchange formulae. The aim of an information exchange protocol is to securely transfer information between participants, ensuring that authentication of the sender, non-repudiation and integrity are all achieved. An exchange consists of three steps that take place between the sender and the receiver of a formula:

1. The formula to be exchanged is transmitted via a secure channel, created through the use of symmetric or public-key encryption. Both the sender and the receiver must believe in the suitability of the encryption key, regardless of whether it is symmetric or asymmetric. Trust in a key can be achieved by running an authentication protocol prior to the execution of the exchange protocol. If a public key is being used to encrypt the formula, then a secret token must be included with the formula to identify the sender. Both the sender and the receiver must believe that this token is a suitable secret.

2. The recipient receives the transmitted message and then verifies the source by successfully decrypting it, or in the case of public-key encryption decrypting it and then verifying the secret token. The contents of encrypted formulae must be fresh and recognizable, otherwise the GNY conveyance laws cannot derive suitable beliefs. At this point in the protocol, the recipient should be convinced of the origin of the formula that was conveyed and should possess it.

3. An acknowledgement message must be transmitted back to the sender to indicate that the formula has been successfully received. This message should be constructed so that the sender can conclude that the recipient believes he conveyed the formula. From this message, the sender should also be able to convince himself that the recipient possesses the information conveyed. There are numerous techniques in the GNY conveyance postulates which can be used. However, the simplest involves transmitting a fresh keyed-hash of the formula and a secret token.

The goals that we have referred to above can be formalized in GNY notation as four end-conditions. Every information exchange protocol should achieve these basic requirements when analyzed with the GNY logic. Consider an information exchange protocol in which the sender is P, the receiver is Q and the information to be exchanged is X. The recommended final beliefs and possessions follow below:

$$(1)\ \ Q \ni X; \quad (2)\ \ Q \mid\equiv P \mid\sim X; \quad (3)\ \ P \mid\equiv Q \ni X; \quad (4)\ \ P \mid\equiv Q \mid\equiv P \mid\sim X$$

Possession (1) and belief (2) are derived from the first message sent from the sender to the receiver, while beliefs (3) and (4) are derived from the acknowledgement message. The conveyance laws are typically used to derive (2), while simple manipulation of being-told and possession laws derive (1). In the case of belief (4), an extension can be used to convey (2) back to the sender. In this case, the sender must believe that the recipient is honest and competent so that he can believe that the recipient believes the extension. Lastly, belief (3) can be obtained by using conveyance laws, if the message is constructed appropriately.

4.6 Modifications to the Inference Rules

Since the formulation of the GNY logic in 1990, changes have been made to the inference rules to solve inconsistencies that resulted in unsound conclusions. The rule sets have also been expanded to

eliminate certain issues of incompleteness, modified to remove redundant premises and updated to allow for automated forward-chaining. In this section we will examine some of the changes made to the logic. The modifications that are described in Section 4.6.1 through to Section 4.6.4 are summarized from [53, 54], while those in the remaining sections have been added as a result of our own experiences with the logic.

4.6.1 Possession Premises in Freshness and Recognizability Rules

In the original GNY logic, the freshness and recognizability rules did not include a premise which stated that a principal can only obtain a belief in the freshness or recognizability of a formula if he possesses the formula. For example, the original F3′ appeared as follows:

$$\frac{P \mid\equiv \sharp(X)}{P \mid\equiv \sharp(X, Y)}$$

However, this rule, and the others formulated in a similar manner, are unsuitable for automated forward-chaining simply because they can be applied indefinitely to generate new beliefs. To overcome this problem, an additional premise of the form $P \ni X$ has been included in every freshness and recognizability rule with a conclusion of the form $P \mid\equiv \sharp(X)$ or $P \mid\equiv \phi(X)$. The resulting rules ensure that a principal can only obtain a belief in the freshness or recognizability of a formula if he possesses the formula. We will now discuss the rationale behind the modification to the freshness rules — the recognizability rules can be handled in a similar manner.

The basic purpose behind applying the GNY logic is to reason about the beliefs and possessions of a given principal. These beliefs and possessions are derived from messages which the principal receives during a protocol run. However, the conclusions obtained from the application of a freshness rule are of no practical value if they do not affect a principal's beliefs about others, or his beliefs regarding what he possesses.

The rule which enables a principal P to obtain beliefs from messages which he has received is the jurisdiction rule J2, which has a premise of the form $P \mid\equiv Q \mid\sim (X \rightsquigarrow C)$. This premise reflects the requirement that P can only obtain beliefs from messages sent by some well-known principal Q, and appears as the conclusion of the conveyance rules C1 through to C6, C10 through to C14, C15 and C16. Therefore, the statement $P \mid\equiv \sharp(X)$ is of significance in deriving P's beliefs about others only if it appears as a premise in one of these rules. Of these rules, only C1 through to C5, and C15 have a freshness requirement. Further, the premise set of each of these rules implies that P possesses each formula occurring in the freshness premise of the rule. We will now formally state and prove this property.

> *Write $\mathcal{S} \vdash C$ to denote the derivability of statement C from a set of statements $\mathcal{S}$. Let $K = \{\text{C1}, \text{C2}, \text{C3}, \text{C4}, \text{C5}, \text{C15}\}$, and let $\mathcal{K}$ denote the premise set of K. If $P \mid\equiv \sharp(X_1, \ldots, X_m)$ is the freshness premise of K, then $\mathcal{K} \vdash P \ni X$ for every $X \in \{X_1, \ldots, X_m\}$.*

The proof for C1 follows below. Note that the freshness premise in C1 has the form $P \mid\equiv \sharp(X, K)$. Since $P \mid\equiv \sharp(X, K)$ is used in this rule to denote $P \mid\equiv \sharp(X)$ or $P \mid\equiv \sharp(K)$, we must first replace C1 by two equivalent rules:

$$\textbf{C1}'. \quad \frac{P \lhd *\{X\}_K \rightsquigarrow C, P \ni K, P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \phi(X), P \mid\equiv \sharp(X)}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K \rightsquigarrow C, P \mid\equiv Q \ni (X, K)}$$

$$\textbf{C1}''. \quad \frac{P \lhd *\{X\}_K \rightsquigarrow C, P \ni K, P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \phi(X), P \mid\equiv \sharp(K)}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K \rightsquigarrow C, P \mid\equiv Q \ni (X, K)}$$

In C1′ we can apply T3, T2, T6 and P1 respectively to the first two premises to obtain $P \ni X$, while $P \ni K$ holds as a premise in C1″, trivially. Proofs for C4 and C15 can be conducted in a similar vein, while proofs for C2, C3 and C5 follow trivially.

4.6.2 An Unsound Rule

The original GNY inference rule set contained an unsound rule for R11. Let's call this rule R11′ and examine the unsound conclusion which can be achieved.

$$\textbf{R11}'. \quad \frac{P \ni H(X), P \ni X}{P \mid\equiv \phi(X)}$$

Assume that $P \ni X$. By applying P6 and R11′ we can derive the groundless conclusion that $P \mid\equiv \phi(X)$. This supposedly means that P's possession of X is sufficient for him to believe that X is recognizable. There is nothing wrong with P6 since it just states that a principal is capable of performing a calculation on a formula which he possesses. The problem lies with R11′. To fix R11′, we introduce an additional premise which states that P believes that the message digest of X is recognizable.

4.6.3 A Redundant Premise

The original GNY rule set included a secret S in the second premise of C4. However, this is unnecessary. From $P \lhd *\{X, < S >\}_{+K}$ and $P \ni -K$ it follows by applying rules T2, T7, T4 and P1 that $P \ni S$.

4.6.4 Unsound Rule Combinations

When used together, the freshness rules F13 and F6, imply a strange result. Let's examine such a scenario and see how an unsound result can be achieved. Suppose that for principal P, all of the following conditions hold:

- $P \mid\equiv \phi(X)$
- $P \mid\equiv \sharp(K)$
- $P \ni K$
- $P \mid\equiv \{X\}_K$

Then, by applying F13 we obtain $P \mid\equiv \sharp(\{X\}_K)$. By further applying F6, we obtain that $P \mid\equiv \sharp(X)$. This problem is not only confined to the freshness rules F13 and F6, but is manifested in a number of rule combinations from both the freshness and recognizability categories. The solution to this problem is to propose side conditions for several of the freshness and recognizability rules, as seen in Appendix A. These side conditions eliminate the derivation of unsound conclusions which occur in their absence.

4.6.5 Rules Regarding Identifying Secrets

A secret that is used between two principals for identification purposes will appear in a protocol specification in two possible forms, namely S or $< S >$, depending on the context in which it is being used. This dual representation can lead to difficulties when implementing the GNY rules in an automated analysis system, as the two representations will probably be viewed as different formulae, even though they represent the same item. Assume that the following conditions hold:

- $P \mid\equiv \sharp(X)$
- $P \ni X$
- $P \ni S$
- $P \ni H(X, < S >)$

We cannot apply F4 to get the result $P \mid\equiv \sharp(H(X, < S >))$, as P possesses the compound formula (X, S) by P3, not $(X, < S >)$, which is syntactically distinct. So, we have to add a rule that will allow us to derive that P possesses $< S >$ if he already possesses a secret S:

P10. $\dfrac{P \ni S, P \mid\equiv P \overset{S}{\longleftrightarrow} Q}{P \ni <S>}, S \notin \{K, +K, -K\}$

Now, assume that in the above conditions P believes S to be fresh instead of X. He possesses the compound formula $(X, < S >)$ by P10 and P3, however, he cannot believe it to be fresh as he doesn't believe in the freshness of either X or $< S >$. So, we must add another rule which will allow us to derive that P believes $< S >$ is fresh if he already believes S to be fresh:

F20. $\dfrac{P \mid\equiv \sharp(S); P \ni <S>}{P \mid\equiv \sharp(<S>)}, S \notin \{K, +K, -K\}$

We can also add a similar recognizability rule which allows us to derive that P believes $< S >$ is recognizable if he already believes S to be recognizable:

R12. $\dfrac{P \mid\equiv \phi(S); P \ni <S>}{P \mid\equiv \phi(<S>)}, S \notin \{K, +K, -K\}$

The following rule allows us to remove the angled brackets from an identifying secret after it has been extracted from an encryption through the use of either T6 or T7:

T9. $\dfrac{P \lhd <S>}{P \lhd S}$

Lastly, we need to have some way of mimicking the symmetrical nature of the shared secret operator:

C20. $\dfrac{P \mid\equiv P \overset{S}{\longleftrightarrow} Q}{P \mid\equiv Q \overset{S}{\longleftrightarrow} P}$

This rule prevents us from having to manually specify each form of the shared secret operator in the initial conditions.

4.6.6 Dropping Formula Extensions

An extension forms an integral part of a GNY formula. However, formulae with extensions can sometimes cause analysis difficulties if the extensions cannot be dropped or handled by the GNY rules. Current GNY rules allowing for the dropping or handling of extensions include T3, P2, R1, R2, F1, F2, C8 and C17. Assume that the following conditions hold:

- $P \lhd *\{X\}_K \rightsquigarrow C$
- $P \ni K$
- $P \mid\equiv P \overset{K}{\longleftrightarrow} Q$
- $P \mid\equiv \phi(X)$
- $P \mid\equiv P \dashv (\{X\}_K \rightsquigarrow C)$

Now, even though the premises are those found in C10, C10 cannot be applied as the final never-originated-here condition includes an extension. Thus, we add need to add the following rule:

C21. $\dfrac{P \mid\equiv Q \dashv (X \rightsquigarrow C)}{P \mid\equiv Q \dashv (X)}$

Now, after applying C21 to the above conditions, we will end up with the result $P \mid\equiv P \dashv (\{X\}_K)$, which will allow for the application of C10. Similarly, C21 can also be used in combination with C11 through to C14 and C16. Rule C22 has also been added to the postulate set to allow for the dropping of extensions that may be attached to secrets in shared secret suitability statements.

4.7 Example Analyses

This section analyzes two security protocols using the GNY logic. The first is an information exchange protocol that seeks to transfer a formula from one principal to another and the second is an authentication protocol that transfers a shared key through the use of public-key cryptography. Significant goals that are achieved during the analysis are highlighted by drawing a box around them.

4.7.1 An Information Exchange Protocol

The information-exchange protocol which we will analyze appears as follows:

(1) $A \longrightarrow B : \{T_a, B, X, S\}_{K_{ab}}$
(2) $B \longrightarrow A : H(X, < S >)$

The aim of this protocol is to transfer the formula X from A to B. The first message in the protocol transfers the information, while the second serves as a confirmation of the success of the transfer. The key K_{ab} is shared between the two principals and already held by both of them. A secret S is transmitted to B so that A will be able to identify the hash in the second message as originating from B. The timestamp T_a is used to ensure freshness of the encryption in the first message. The protocol parser produces the following output after extensions have been appended to formulae which use secrets:

(1) $A \longrightarrow B : *\{T_a, B, X, S\}_{K_{ab}} \rightsquigarrow A \overset{S}{\longleftrightarrow} B$
(2) $B \longrightarrow A : *H(X, < S >) \rightsquigarrow A \mid\sim X$

The initial assumptions are categorized and listed as follows:

Possessions:	$A \ni (T_a, B, X, S, K_{ab})$; $B \ni K_{ab}$
Recognizability:	$B \mid\equiv \phi(B)$
Freshness:	$B \mid\equiv \sharp(T_a)$; $A \mid\equiv \sharp(S)$
Shared Secrets:	$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$; $B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$; $A \mid\equiv A \overset{S}{\longleftrightarrow} B$
Trustworthiness:	$B \mid\equiv A \mid\Rightarrow A \mid\equiv *$; $A \mid\equiv B \mid\Rightarrow B \mid\equiv *$
Jurisdiction:	$B \mid\equiv A \mid\Rightarrow A \overset{S}{\longleftrightarrow} B$

Thus, A and B both possess the key K_{ab} and believe it to be suitable for encrypted communication. A initially possesses the secret S and believes it to be suitable for identification purposes between himself and B, while B believes that A is an authority on the suitability of S. A and B both consider each other to be competent and honest. The timestamp T_a must be believed to be fresh by B, while S is believed to be fresh by A. Lastly, B can recognize his own identifier.

For message 1, applying the inference rules E1, E5, E2 and T1:

$$B \lhd *\{T_a, B, X, N_a, S\}_{K_{ab}} \rightsquigarrow A \overset{S}{\longleftrightarrow} B \quad \ldots \ (1)$$

Now, applying T3, T2, T6, T4, P1, P4:

$$\boxed{B \ni X}$$

Then, from (1), F3′, R3′, C1, F5, J2 and J1:

$$B \mid\equiv A \overset{S}{\longleftrightarrow} B$$

Also, from (1), F3′, R3′, C1 and C9:

$$\boxed{B \mid\equiv A \mid\sim X}$$

For message 2, applying the inference rules E1, E4, E2 and T1:

$$A \lhd *H(X, < S >) \rightsquigarrow A \mid\sim X \quad \ldots \ (2)$$

Now, using F3′, R3′, C3, P1, P2, F4 and J2:

$$\boxed{A \mid\equiv B \mid\equiv A \mid\sim X}$$

Finally, from (2), F3′, R3′ and C3:

$$\boxed{A \mid\equiv B \ni X}$$

From this analysis we can determine that four important goals are attained by this protocol. Firstly, B obtains the component X and believes that A was the one who conveyed it. And secondly, A believes that B believes that A originated X and A also believes that B has successfully received X.

4.7.2 An Authentication Protocol

The authentication protocol which we will analyze appears as follows:

(1) $A \longrightarrow B : A, N_a$
(2) $B \longrightarrow A : \{B, \{N_a\}_{-K_b}, K, N_b\}_{+K_a}, \{N_a\}_K$
(3) $A \longrightarrow B : \{N_b\}_K$

The aim of this protocol is to transfer the session key K from B to A. B generates the key, while A requests it by initiating the protocol run. The two nonces, N_a and N_b, are used to provide freshness and the ability to identify that encryptions have not been corrupted. A two-way handshake takes place during the protocol run, the first handshake being embedded in the second message, and the last being embedded in the final message. The purpose of these handshakes is to convince the principals that each of them has successfully received the key and believes in its suitability. The protocol parser produces the following output after extensions have been appended to formulae which use secrets:

(1) $A \longrightarrow B : A, N_a$
(2) $B \longrightarrow A : \{B, *\{N_a\}_{-K_b} \rightsquigarrow A \overset{K}{\longleftrightarrow} B, K, N_b\}_{+K_a}, *\{N_a\}_K$
(3) $A \longrightarrow B : *\{N_b\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B$

The initial assumptions are categorized and listed as follows:

Possessions:	$A \ni (A, N_a, -K_a, +K_b)$; $B \ni (B, N_b, +K_a, -K_b)$; $\boxed{B \ni K}$
Recognizability:	$A \mid\equiv \phi(N_a)$; $B \mid\equiv \phi(N_b)$
Freshness:	$A \mid\equiv \sharp(N_a)$; $B \mid\equiv \sharp(N_b)$
Public Keys:	$B \mid\equiv \overset{+K_b}{\mapsto} B$; $B \mid\equiv \overset{+K_a}{\mapsto} A$; $A \mid\equiv \overset{+K_a}{\mapsto} A$; $A \mid\equiv \overset{+K_b}{\mapsto} B$
Shared Secrets:	$\boxed{B \mid\equiv A \overset{K}{\longleftrightarrow} B}$
Trustworthiness:	$A \mid\equiv B \mid\Rightarrow B \mid\equiv *$; $B \mid\equiv A \mid\Rightarrow A \mid\equiv *$
Jurisdiction:	$A \mid\equiv B \mid\Rightarrow A \overset{K}{\longleftrightarrow} B$

B possesses the shared key K, A's public key and his own private key, while A possesses his private key and B's public key. Both A and B regard the nonces which they originate to be recognizable and fresh, and they also trust in the suitability of their own and each other's public keys. B is considered by A to be an authority on the suitability of the shared key. A and B both consider each other to be honest and competent. Lastly, B believes that K is a suitable secret key.

For message 1, applying the postulates E1, T1 and P1:

$$B \ni (A, N_a)$$

For message 2, applying the postulates E1, E7, E2, E1, E3, E9 and T1:

$$A \lhd \{B, *\{N_a\}_{-K_b} \rightsquigarrow A \overset{K}{\longleftrightarrow} B, K, N_b\}_{+K_a}, *\{N_a\}_K \quad \ldots \ (1)$$

Now applying T4, T7 and T4:

$$A \lhd *\{N_a\}_{-K_b} \rightsquigarrow A \overset{K}{\longleftrightarrow} B$$

Then, using C6, F9 and J2:

$$\boxed{A \mid\equiv B \mid\equiv A \overset{K}{\longleftrightarrow} B}$$

Further, applying J1:

$$\boxed{A \mid\equiv A \overset{K}{\longleftrightarrow} B}$$

From (1), T4, T7, T4 (isolate K and N_b), P1 and P4:

$\boxed{A \ni K}$ and $A \ni N_b$

From (1) and T4:

$$A \lhd *\{N_a\}_K$$

Then applying C1:

$$\boxed{A \mid\equiv B \ni K}$$

For message 3, applying the postulates E1, E5 and T1:

$$B \lhd *\{N_b\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B \quad \ldots \quad (2)$$

Applying C1:

$$\boxed{B \mid\equiv A \ni K}$$

And, finally applying T2, T3, P1, F5 and J2:

$$\boxed{B \mid\equiv A \mid\equiv A \overset{K}{\longleftrightarrow} B}$$

In this example, two of the goals which we wish to achieve for this class of protocol are in our initial assumptions. Besides these two, another six significant goals are achieved. Firstly, A obtains the session key K and trusts in its suitability. A also believes that B trusts in the suitability of K and that B possesses K. Lastly, B is convinced that A possesses K and that A trusts in the suitability thereof.

4.8 Closing Remarks

GNY enhances the framework provided by BAN by defining a protocol description language and a protocol parser. It is also more general and does not need several default assumptions required by BAN. The notion of *recognizability* captures a recipient's expectation of the redundancy in an encrypted formula and does not presuppose how redundancy is provided. Also, it is not assumed that a principal is able to determine that he was not the first to originate certain formulae. Instead, backward replays of fresh messages are detected during the parsing and inference rule application process. However, if a principal has the ability to determine that he was not the first to originate certain formulae then he can express this in his initial beliefs.

Formula extensions enable the distinction between a principal's possessions and beliefs. Because of this distinction a principal can convey a formula which he possesses, even though he has no confidence in the content of the formula. This stands in contrast to BAN logic, which states that if a principal sends a formula, then the recipient can believe that the sender has confidence in the formula.

$$\frac{P \mid\equiv \sharp(X), P \mid\equiv Q \mid\sim X}{P \mid\equiv Q \mid\equiv X} \qquad \text{(BAN nonce verification rule)}$$

The use of extensions also permits the separate treatment of the contents of a formula and the information implied by the conveyance of such a formula. This emphasis also separates the reasoning about the physical world from reasoning about other principals' beliefs, bringing in the ability to reason at more than one level. What a protocol achieves depends on the level of mutual trust among the participants.

Principals may obviously only convey formulae which they possess at the time a message is constructed. In the same way, beliefs included in formula extensions must hold at the time the formula is conveyed. BAN does not include checks of this nature and thus infeasible results may be obtained when ill-designed protocols are analyzed. The notion of *eligibility* makes these checks automatic in the GNY analysis process thus unifying the detection of protocol inconsistencies and protocol analysis. However, eligibility laws do not exclude the fact that some principals may lie at runtime and convey messages with extensions irrespective of whether they believe the extension or not. This is possible because there is no general way in which a link can be established between what a principal believes and what a principal says he believes. It all boils down to a matter of trust on the side of the recipient.

A replay is any act in which an attacker retransmits a message that was originally transmitted by another principal. Replaying a message to the same recipient is a *forward replay*, while replaying a message to the originator is a *backward replay*. When carrying out a protocol analysis, one only wants to derive beliefs from and draw conclusions about formulae that are not replays. Formulae from which conclusions can be drawn using the conveyance inference rules are symmetric and private-key encryptions, keyed hashes and public-key encryptions containing a shared secret. By prefixing a star to these formulae during the parsing process one can be certain that they are not replays from the current session. The conveyance inference rules prevent reasoning about formulae which are possible replays by only reasoning about formulae prefixed by a star or formulae that a principal believes he can recognize as never having originated.

A protocol analysis consists of annotating the protocol with statements and manipulating these statements with the aid of the inference rules. A protocol is essentially a sequence of told-statements. An annotation for a protocol consists of a sequence of assertions (conjunctions of statements) inserted before the first told-statement and after each told-statement. The first assertion contains the assumptions and the last contains the conclusions. As in BAN, if the assumptions hold, then each assertion should hold after the execution of its respective protocol prefix. The assertions are derived by the syntactic application of the

inference rules to the derived statements. The goal of an analysis should be be to derive the final positions of each of the principals at the termination of a given protocol. This final position is represented by the corresponding assertions.

Chapter 5

Automated GNY Analysis with GYNGER

"Security, like correctness, is not an add-on feature."

— Andrew S. Tanenbaum

The inherent appeal in using modal logics stems from their simplicity and effectiveness for analyzing cryptographic protocols. Logics can be systematically applied to reason about the working of protocols, often helping to reveal missing assumptions, deficiencies or redundancies. This can then lead to the protocol, the assumptions or the original goals being re-evaluated, after which the inference rules can be reapplied to determine whether the goals are attainable after these modifications have been made.

However, the process of applying and reapplying the inference rules is often tedious and error-prone when carried out manually. Another problem is that the possibility of accidentally missing conclusions drawn from inference rules increases in logics such as GNY, which has more than eighty inference rules, some of them being quite complex. Thus, due to the difficultly of manually analyzing protocols, a number of automated logic-based analysis tools have been developed [20, 27, 61, 47, 14, 54]. Of these tools, the Prolog-based analyzer in [54] and the Objective CAML-based [69] analyzer in [61] both use GNY to analyze protocols, while the HOL-based analyzer in [14] uses a derivative of GNY, known as BGNY.

Our main aim in automating a GNY protocol analysis is to be able to determine whether one or more statements describing the goals of a specific protocol are derivable from a given set of initial assumptions. However, it is also desirable to generate all of the statements that are derivable for a given protocol as this would allow us to compare the final states of each principal. For this reason, a forward-chaining based approach would be the most suitable strategy [70], since it would result in the repeated application of the GNY inference rules to the set of statements consisting of the protocol messages, initial assumptions and derived statements, until all of the statements which are derivable have been generated. In contrast, backward-chaining would merely determine whether a given goal is achievable [70].

Initially we thought of integrating the analyzer described in [54] within the SPEAR II Framework. This decision was primarily motivated by the analyzer's simplicity and reliance on Prolog as an implementation language. However, upon closer examination we discovered limitations in the syntax and scope of the analyzer. These limitations were enough to severely hamper the quality of GNY analyses that we anticipated. Thus, we decided to construct our own Prolog-based forward-chaining GNY analyzer, named GYNGER, expanding on the concepts presented in [54]. However, because the authors of [54] were unable to provide us with a working copy of their analyzer, we had to resort to building GYNGER from scratch and could only use the minimal source code fragments alluded to in [54].

The aim of this chapter is to describe the automated GYNGER analysis tool which we have developed. We will briefly sketch some background information and then describe the GNYGER source code, showing how we implemented the GNY inference rules. Four sample protocol analyses which employ the tool will also be discussed. We conclude with a comparison of GYNGER and the analyzer described in [54].

5.1 Preliminary Issues

Since the analyzer will be employing forward-chaining techniques, a subset of the existing GNY rules which are incompatible with this approach will have to be removed. An informal proof will then be constructed to show that the remaining inference rules can be used to derive a finite number of conclusions in a finite number of steps based on the initial assumptions and messages of a given protocol.

5.1.1 Removal of Existing GNY Rules

Sixteen of the eighty-eight GNY rules have not been incorporated within GYNGER. However, the exclusion of these rules does not affect the useful inferences which can be derived, but instead helps to ensure that the logical statements derivable from a given protocol can be deduced in a finite number of steps.

The possession rules are useful for enforcing a possession consistency check, but are of no use otherwise. Thus, several of the possession rules have been removed, specifically P3 and P6 through to P9, since each of these rules can be applied indefinitely to produce trivial new possessions which have no bearing on an analysis. For example, consider the possession rule for shared keys:

P7. $$\frac{P \ni X, P \ni K}{P \ni \{X\}_K, P \ni \{X\}_K^{-1}}$$

This rule may be applied to an initial set containing $P \ni K$ and $P \ni X$ to indefinitely generate multitudes of new encryptions and decryptions, none of which would be pertinent to the analysis. Along with the selected possession rules, we also remove all of the eligibility rules, as they too can be applied without terminating. As a result of removing the eligibility rules, we have also removed T1, which contains an eligibility premise.

5.1.2 Finiteness of Derivations

In this section we will sketch a proof for the following statement based on the GNY rule set presented in Appendix A. This proof is similar to the one presented in [54].

> *The statements derivable from a finite set of idealized protocol steps and initial assumptions are finite in number, and are therefore derivable in a finite number of steps.*

We will use the notation $(\mathcal{D}/E)$ to denote a generic inference rule, where $\mathcal{D}$ is the set of premises in the rule, and E is the rule's conclusion. If a rule has multiple conclusions, then it is decomposed into separate rules, each with a single conclusion. Now, denote the modified set of rules by $\mathcal{R}$. Then, we define an operator ρ on a set of statements, $\mathcal{S}$, as follows:

$$\rho(\mathcal{S}) = \mathcal{S} \cup \{E : \exists(\mathcal{D}/E) \in \mathcal{R} \text{ such that } \mathcal{D} \subseteq \mathcal{S}\}$$

Thus, ρ returns $\mathcal{S}$ together with the statements derivable from $\mathcal{S}$ by applying the inference rules in $\mathcal{R}$ once. We now want to show that there exists an integer n such that

$$\rho^n(\mathcal{S}) = \rho^\infty(\mathcal{S}), \text{ where } \rho^m(\mathcal{S}) \text{ represents } \bigcup_{i=0}^{m} \rho^i(\mathcal{S}).$$

The key step in this proof is the definition of a relation $\prec$ over the set of statements which have the form $P \ni X$, $P \lhd X$ and $P \mid\!\equiv C$. We create seven subsidiary relations $\prec_\lhd$, $\prec_\ni$, $\prec_\ni^\lhd$, $\prec_{\mid\equiv}$, $\prec_{\mid\equiv}^\ni$, $\prec_{\mid\equiv}^\lhd$ and $\prec_\ni^{\mid\equiv}$ and use them as follows:

(1) $P \lhd X \prec P \lhd Y$, *if* $X \prec_\lhd Y$
(2) $P \ni X \prec P \ni Y$, *if* $X \prec_\ni Y$
(3) $P \ni X \prec P \lhd Y$, *if* $X \prec_\ni^\lhd Y$
(4) $P \mid\!\equiv C \prec P \mid\!\equiv D$, *if* $C \prec_{\mid\equiv} D$
(5) $P \mid\!\equiv C \prec P \ni X$, *if* $C \prec_{\mid\equiv}^\ni X$
(6) $P \mid\!\equiv C \prec P \lhd X$, *if* $C \prec_{\mid\equiv}^\lhd X$
(7) $P \ni X \prec P \mid\!\equiv C$, *if* $X \prec_\ni^{\mid\equiv} C$

The definition of $\prec_\lhd$ is read off the being-told rules T2, T3, T4, T5, T6, T7, T8 and T9 where T4 is used in its two symmetrical forms. The following clauses result:

(1) $X \prec_\lhd *X$
(2) $X \prec_\lhd X \rightsquigarrow C$
(3) (i) $X \prec_\lhd (X, Y)$
(ii) $X \prec_\lhd (Y, X)$
(4) $X_i \prec_\lhd F(X_0, X_1, \ldots, X_{n-1})$
(5) $X \prec_\lhd \{X\}_K$
(6) $X \prec_\lhd \{X\}_{+K}$
(7) $X \prec_\lhd \{X\}_{-K}$
(8) $S \prec_\lhd < S >$

The definition of $\prec_\ni$ is read off the possession rules P2, P4 and P5, where P4 is used in its two symmetrical forms. The following clauses are obtained:

(1) $X \prec_\ni X \rightsquigarrow C$
(2) (i) $X \prec_\ni (X, Y)$
(ii) $X \prec_\ni (Y, X)$
(3) $X_i \prec_\ni F(X_0, X_1, \ldots, X_{n-1})$

The definition of $\prec_\ni^\lhd$ consists of a single clause which is read off the possession rule P1:

(1) $X \prec_\ni^\lhd X$

The definition of $\prec_{\mid\equiv}$ is read off the conveyance rules C8, C9, C17 through to C22, where C9, C17 and C19 are used in their two symmetrical forms, the jurisdiction rules J1, J2 and J3, and finally, R2 and F1, yielding clauses as follows:

(1) $Q \mid\sim X \prec_{\mid\equiv} Q \mid\sim X \rightsquigarrow C$
(2) (i) $Q \mid\sim X \prec_{\mid\equiv} Q \mid\sim (X, Y)$
(ii) $Q \mid\sim X \prec_{\mid\equiv} Q \mid\sim (Y, X)$
(3) (i) $Q \mid\sim X \rightsquigarrow C \prec_{\mid\equiv} Q \mid\sim X \rightsquigarrow (C, C')$
(ii) $Q \mid\sim X \rightsquigarrow C \prec_{\mid\equiv} Q \mid\sim X \rightsquigarrow (C', C)$
(4) $Q \ni X \prec_{\mid\equiv} Q \mid\sim X$
(5) (i) $Q \ni X \prec_{\mid\equiv} Q \ni (X, Y)$
(ii) $Q \ni X \prec_{\mid\equiv} Q \ni (Y, X)$
(6) $Q \overset{S}{\longleftrightarrow} P \prec_{\mid\equiv} P \overset{S}{\longleftrightarrow} Q$
(7) $P \dashv (X) \prec_{\mid\equiv} P \dashv (X \rightsquigarrow C)$
(8) $P \overset{S}{\longleftrightarrow} Q \prec_{\mid\equiv} P \overset{S \rightsquigarrow C}{\longleftrightarrow} Q$
(9) $C \prec_{\mid\equiv} Q \mid\equiv C$
(10) $Q \mid\equiv C \prec_{\mid\equiv} Q \mid\sim (X \rightsquigarrow C)$
(11) $Q \mid\equiv C \prec_{\mid\equiv} Q \mid\equiv Q \mid\equiv C$
(12) $\phi(X) \prec_{\mid\equiv} \phi(X \rightsquigarrow C)$
(13) $\sharp(X) \prec_{\mid\equiv} \sharp(X \rightsquigarrow C)$

The definition of $\prec^{\ni}_{\mid\equiv}$ consists of two clauses. The first is obtained from the freshness postulates F2 through to F20 and the second from the recognizability postulates R1 and R3′ through to R12:

(1) $\sharp(X) \prec^{\ni}_{\mid\equiv} X$
(2) $\phi(X) \prec^{\ni}_{\mid\equiv} X$

The definition of $\prec^{\lhd}_{\mid\equiv}$ is read off the conveyance rules C1 through to C7 and C10 through to C16. Each rule adds as many clauses to the definition as there are conclusions, resulting in the following collection:

(1) (i) $Q \mid\sim X \prec^{\lhd}_{\mid\equiv} *\{X\}_K \rightsquigarrow C$
(ii) $Q \mid\sim \{X\}_K \rightsquigarrow C \prec^{\lhd}_{\mid\equiv} *\{X\}_K \rightsquigarrow C$
(iii) $Q \ni X \prec^{\lhd}_{\mid\equiv} *\{X\}_K \rightsquigarrow C$
(iv) $Q \ni K \prec^{\lhd}_{\mid\equiv} *\{X\}_K \rightsquigarrow C$
(2) (i) $Q \mid\sim X \prec^{\lhd}_{\mid\equiv} *\{X\}^{-1}_K \rightsquigarrow C$
(ii) $Q \mid\sim \{X\}^{-1}_K \rightsquigarrow C \prec^{\lhd}_{\mid\equiv} *\{X\}^{-1}_K \rightsquigarrow C$
(iii) $Q \ni X \prec^{\lhd}_{\mid\equiv} *\{X\}^{-1}_K \rightsquigarrow C$
(iv) $Q \ni K \prec^{\lhd}_{\mid\equiv} *\{X\}^{-1}_K \rightsquigarrow C$
(3) (i) $Q \mid\sim (X, S) \prec^{\lhd}_{\mid\equiv} *H(X, < S >) \rightsquigarrow C$
(ii) $Q \mid\sim H(X, < S >) \rightsquigarrow C \prec^{\lhd}_{\mid\equiv} *H(X, < S >) \rightsquigarrow C$
(iii) $Q \ni X \prec^{\lhd}_{\mid\equiv} *H(X, < S >) \rightsquigarrow C$
(iv) $Q \ni S \prec^{\lhd}_{\mid\equiv} *H(X, < S >) \rightsquigarrow C$
(4) (i) $Q \mid\sim (X, S) \prec^{\lhd}_{\mid\equiv} *\{X, < S >\}_{+K} \rightsquigarrow C$
(ii) $Q \mid\sim \{X, < S >\}_{+K} \rightsquigarrow C \prec^{\lhd}_{\mid\equiv} *\{X, < S >\}_{+K} \rightsquigarrow C$
(iii) $Q \ni X \prec^{\lhd}_{\mid\equiv} *\{X, < S >\}_{+K} \rightsquigarrow C$
(iv) $Q \ni S \prec^{\lhd}_{\mid\equiv} *\{X, < S >\}_{+K} \rightsquigarrow C$
(v) $Q \ni +K \prec^{\lhd}_{\mid\equiv} *\{X, < S >\}_{+K} \rightsquigarrow C$
(5) (i) $Q \mid\sim X \prec^{\lhd}_{\mid\equiv} *\{X\}_{-K} \rightsquigarrow C$
(ii) $Q \mid\sim \{X\}_{-K} \rightsquigarrow C \prec^{\lhd}_{\mid\equiv} *\{X\}_{-K} \rightsquigarrow C$
(6) (i) $Q \ni X \prec^{\lhd}_{\mid\equiv} *\{X\}_{-K}$

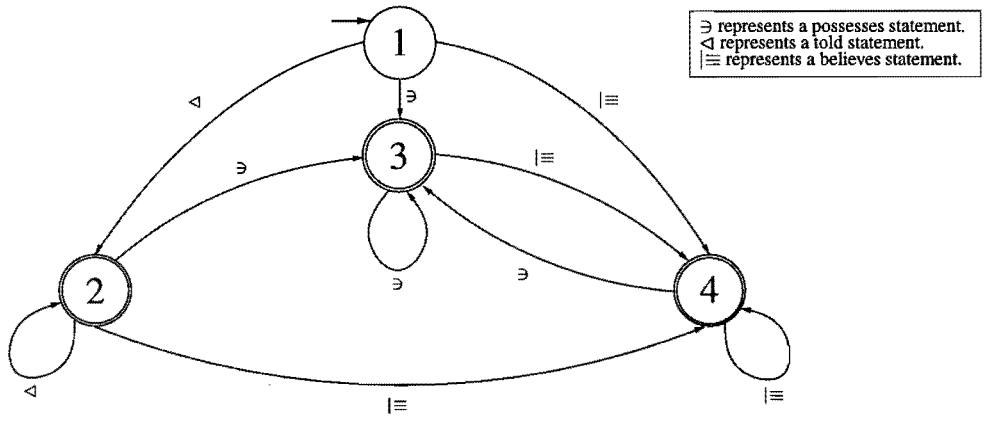


Figure 5.1: A finite state machine showing chains that can be formed for the $\prec$ relation.

(ii) $Q \ni -K \prec^{\lhd}_{|\equiv} *\{X\}_{-K}$

(7) (i) $Q \mid\sim (X, S) \prec^{\lhd}_{|\equiv} *\{X, < S >\}_K \rightsquigarrow C$

(ii) $Q \mid\sim \{X, < S >\}_K \rightsquigarrow C \prec^{\lhd}_{|\equiv} *\{X, < S >\}_K \rightsquigarrow C$

(iii) $Q \ni X \prec^{\lhd}_{|\equiv} *\{X, < S >\}_K \rightsquigarrow C$

(iv) $Q \ni S \prec^{\lhd}_{|\equiv} *\{X, < S >\}_K \rightsquigarrow C$

(v) $Q \ni K \prec^{\lhd}_{|\equiv} *\{X, < S >\}_K \rightsquigarrow C$

Finally, the definition of $\prec^{|\equiv}_{\ni}$ is read off the possession rule P10:

(1) $< S > \prec^{|\equiv}_{\ni} P \xleftrightarrow{S} Q$

Of the seven subsidiary relations, the most critical in the analysis are $\prec_{\lhd}$, $\prec_{\ni}$ and $\prec_{|\equiv}$. Each of these relations is well-founded, which means that they do not give rise to infinitely descending chains. For example, in the case of $\prec_{\lhd}$ the formula on the left in each clause is syntactically shorter than the formula on the right, so there cannot be infinitely descending chains with respect to this relation. Well-foundedness of $\prec_{\ni}$ and $\prec_{|\equiv}$ is proved just as easily. Note that we do not require well-foundedness of the remaining three subsidiary relations, as they cannot give rise to chains.

Thus, from the definition of $\prec$ above, it is clear that $\prec$ is also well-founded. In fact, any infinite descending chain with respect to $\prec$ must contain an infinite descending chain of statements of one of the three forms, namely $P \ni X$, $P \lhd X$ or $P \mid\equiv C$. To justify this fact, notice that the finite state machine in Figure 5.1 is based on the $\prec$ relation. If a chain relative to $\prec$ is infinite, then we will be looping repeatedly in either state 2, 3, or 4, since the loop between states 3 and 4 cannot be repeated infinitely and there are no other cycles. Thus, any infinite chain will consist exclusively of statements of the form $P \ni X$, $P \lhd X$ or $P \mid\equiv C$. However, a chain of one of these three forms must have arisen from an infinite descending chain with respect to one of the subsidiary relations, namely $\prec_{|\equiv}$, $\prec_{\lhd}$ or $\prec_{\ni}$, and this is impossible.

Another property of the seven subsidiary relations is that they are finitary, which means that given any statement C of the form $P \ni X$, $P \lhd X$ or $P \mid\equiv C'$, the set of statements $\{D : D \prec C\}$ is finite. This is obvious in the case of $\prec_{\lhd}$, and is equally easily proved for the other six relations. It follows that $\prec$ is also finitary. Since $\prec$ is well-founded and finitary, it follows by the contrapositive of König's Infinity Lemma[1] that for any C the set of statements $\{\mathrm{D} : \mathrm{D} \prec^* \mathrm{C}\}$, where $\prec^*$ denotes the transitive and reflexive

[1] *König's Infinity Lemma* states that a tree is infinite (has infinitely many edges) iff it has a node of infinite degree or it has an infinitely long path [48].

closure of $\prec$, is finite as well.

Now, the definitions of the subsidiary relations and $\prec$ are constructed so as to give a straight-forward guarantee that for each rule $(\mathcal{D}/E) \in \mathcal{R}$, there exists $C \in \mathcal{D}$ such that $E \prec C$; that is, in each rule the conclusion is smaller, with respect to $\prec$, than at least one of the premises. Thus, if $\mathcal{S}$ is the set of idealized analysis steps and initial assumptions of the protocol, and $C \in \rho^{\infty}(\mathcal{S})$, then there exists $S \in \mathcal{S}$ such that $C \prec^{*} S$. Thus,

$$\rho^{\infty}(\mathcal{S}) \subseteq \bigcup_{S \in \mathcal{S}} \{C : C \prec^{*} S\}$$

Since the right-hand side is a finite union of sets, given the assumption that $\mathcal{S}$ is finite, $\rho^{\infty}(\mathcal{S})$ is finite, as was required.

5.2 Implementing the Analyzer

The Prolog-based analyzer that we have created relies on a forward-chaining inference engine to generate all of the GNY beliefs and possessions that can result from the systematic application of the inference rules to a set of initial assumptions and message steps. To conduct an analysis, a user needs to specify the message steps and initial conditions. Upon completion of the analysis, the Prolog database can be queried to determine whether specific GNY statements are true, or alternatively all of the derived facts or a subset thereof can be displayed. If the user chooses, a proof can be generated for each fact in the database, showing all of the steps and inference rules that were required to generate the result.

5.2.1 Representing GNY Structures

The Prolog language is well-suited for implementing a GNY analyzer as it is straight-forward to map all of the GNY constructs into suitable Prolog counterparts which can be easily manipulated and queried.

5.2.1.1 Formulae and Statements

Formulae are the components which are used to construct protocol messages and typically contain constants such as principal names, nonces, shared keys, etc. These constants are represented by characters enclosed in single quotes. For example, the nonce N_a is denoted as `'Na'`. The remaining GNY formulas are represented by suitable Prolog counterparts:

GNY Formula	*Prolog Representation*
(X,Y)	`[X, Y]`
$\{X\}_K$ and $\{X\}_K^{-1}$	`encrypt(X, shared(K))` and `decrypt(X, shared(K))`
$\{X\}_{+K}$ and $\{X\}_{-K}$	`encrypt(X, public(K))` and `encrypt(X, private(K))`
$H(X)$	`hash(H, X)`
$F(X)$	`function(F, X)`
$< S >$	`identifyingSecret(S)`
$*X$	`star(X)`
$X \rightsquigarrow C$	`extension(X, C)`

Logical statements are represented by appropriately named Prolog structures:

GNY Statement	*Prolog Representation*
$P \lhd X$	`told(P, X)`
$P \ni X$	`possesses(P, X)`
$P \mid\sim X$	`conveyed(P, X)`
$\sharp(X)$	`fresh(X)`
$\phi(X)$	`recognizable(X)`
$P \overset{S}{\longleftrightarrow} Q$	`secret(P, S, Q)`
$\overset{+K}{\mapsto} Q$	`public(K, Q)`
$P \dashv (X)$	`neverOriginated(P, X)`
(C_1, C_2)	`[C1, C2]`
$P \mid\equiv C$	`believes(P, C)`
$P \mid\Longrightarrow C$	`controls(P, C)`
$P \mid\Longrightarrow P \mid\equiv *$	`honest(P)`

The translations shown above allow us to represent any formula or statement in GNY syntax by an appropriate Prolog-style counterpart. For example, $A \lhd (N_a, *\{X, T_b\}_K \rightsquigarrow \sharp(N_a))$ can be represented as `told('A', ['Na', extension(star(encrypt(['X', 'Tb'], shared('K'))), fresh('Na'))]).`

5.2.1.2 Storing Statements

Apart from representing GNY constructs in Prolog syntax, we also need to store the derivation information for statements obtained through the application the inference rules. The predicate `fact/3`, which defines an inference step, is used for this purpose, and appears as follows:

```
fact(Index, Statement, reason(PremiseList, Rule))
```

The argument `Statement` is bound to a derived statement, while the integer `Index` is used to reference instances of `fact/3`. In the last argument, `PremiseList` is a list containing the indices of the premises that were used in deriving `Statement` through the application of `Rule`. For consistency, the statements representing the idealized protocol messages and initial assumptions are also stored as instances of `fact/3`. In this case, `PremiseList` would be empty and `Rule` would be either `'Step'` or `'Assumption'`. The following are examples of `fact/3` clauses:

```
fact(1, told('A', encrypt(['Nb', 'data'], shared('K'))), reason([], 'Step')).
fact(2, possesses('A', shared('K')), reason([], 'Assumption')).
fact(3, told('A', ['Nb', 'data']), reason([1, 2], 'T6')).
```

Protocol goals are represented in a similar way with the `goal/2` predicate:

```
goal(Index, Statement)
```

Once an analysis is complete, a proof is constructed for each successful `goal/2` predicate defined for the protocol. As new statements are derived through the application of the inference rules, the maximum index among the `fact/3` predicates has to be extracted so that the newly derived statement will have a unique index that is one greater than the previous maximum. The predicate `getMaxFactIndex/1`, along with the helper predicate `getMaxInList/2`, is used for this task:

```
getMaxInList([Item], Max) :-
    Max is Item.

getMaxInList([Head | Tail], Max) :-
    getMaxInList(Tail, TempMax),
    Max is max(Head, TempMax).

getMaxFactIndex(MaxIndex) :-
    bagof(Index, X^Y^fact(Index, X, Y), IndexList),
    getMaxInList(IndexList, MaxIndex).
```

Essentially, `getMaxFactIndex/1` collects all of the indices within `fact/3` into a list and then finds the maximum in this list using `getMaxInList/2`. The maximum index is returned in the argument `MaxIndex`.

5.2.2 The Forward-Chaining Inference Engine

The GNY inference rules are all specified through the use of a `rules/0` predicate. For each GNY rule, there is at least one instance of the `rules/0` predicate. For example, the `rules/0` predicate for T6 follows below:

```
rules :-
    fact(Premise1, told(P, encrypt(X, shared(K))), _),
    fact(Premise2, possesses(P, shared(K)), _),
    Conclusion = told(P, X),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    PremiseIndices = [Premise1, Premise2],
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'T6'))),
    asserta(addedFacts).
```

The basic pattern followed in a typical instance of the `rules/0` predicate is to first check that all of the premises of the respective GNY rule are true and then to assert the conclusion in the Prolog database if it has not yet been asserted. After asserting the conclusion, the `addedFacts` atom is also asserted in the database to indicate that a conclusion was derived during the current cycle. To cycle through all of the inference rules, the `oneCycle/0` predicate is employed:

```
oneCycle :-
    rules,
    fail.
```

The `fail` clause in the `oneCycle/0` predicate ensures that back-tracking takes place, resulting in all of the `rules/0` predicates being executed as Prolog attempts to satisfy the decision point prior to the `fail` clause. To implement forward-chaining, the `forward/1` predicate is used:

```
forward(Cycle) :-
    Cycle > 0,
    done.

forward(Cycle) :-
    not(oneCycle),
    NextCycle is Cycle + 1,
    forward(NextCycle).
```

The `forward/1` predicate recursively calls itself until the `done/0` predicate is satisfied. Notice that since `oneCycle/0` always fails, we have to take the complement using `not` in the second instance of `forward/1`. The `done/0` predicate checks whether any new beliefs or possessions have been added to the Prolog database in the current cycle:

```
done :-
    not(retract(addedFacts));
    (
        retractall(addedFacts),
        fail
    ).
```

The `done/0` predicate tries to determine if the `addedFacts` atom has been asserted by attempting to retract it from the Prolog database. If `addedFacts` has been asserted, then all the remaining instances are retracted from the database, and the `done/0` predicate fails. Otherwise, if no atoms of `addedFacts` can initially be retracted, then `done/0` succeeds because of the *or* condition. Lastly, an analysis is begun by invoking the `analyze/0` predicate:

```
analyze :-
    fact(_, _, _),
    asserta(addedFacts),
    forward(1).
```

This predicate first checks whether any facts reside in the database and then starts the forward-chaining process by asserting the `addedFacts` atom and calling the `forward/1` predicate with an argument of 1 to initialize the cycle number. The reason why `addedFacts` is asserted is because we want to ensure that the second instance of `forward/1` is also invoked in the first cycle. If `addedFacts` was not asserted then `done/0` would succeed and forward-chaining would not commence.

5.2.3 Coding the GNY Rules

The analyzer implements seventy-two of the eighty-eight GNY rules listed in Appendix A, omitting those which inhibit forward-chaining. All of these rules require at least one instance of the `rules/0` predicate, some requiring more because of multiple conclusions. Thus, due to the size of the code that was written to implement the rules, we cannot show the Prolog source for every rule. However, we will discuss the important issues which emerged during development and illustrate these with sample code.

5.2.3.1 Being-Told and Possession Rules

The being-told rules T2, T3 and T6 through to T9, as well as the possession rules P1 and P2 are straightforward to implement. An example of how to code these rules in Prolog was illustrated in Section 5.2.2 using the source code for T6. However, (T4, P4), (T5, P5) and P10 require a bit more thought. When implementing T4 and P4, we have to consider that a list of formulae may contain more than two elements. In fact, it could contain quite a couple. So, we want to implement the intent of these two rules, while still making sure that we do not generate too many superfluous statements. The source code for T4 appears as follows:

```
rules :-
    fact(Premise, told(P, [X | _]), _),
    Conclusion = told(P, X),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    asserta(fact(NewIndex, Conclusion, reason([Premise], 'T4'))),
    asserta(addedFacts).

rules :-
    fact(Premise, told(P, [_ | Rest]), _),
    length(Rest, LengthOfRest),
    (
        LengthOfRest > 1,
        Conclusion = told(P, Rest);

        LengthOfRest =:= 1,
        getHead(Rest, Head),
        Conclusion = told(P, Head)
    ),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    asserta(fact(NewIndex, Conclusion, reason([Premise], 'T4'))),
    asserta(addedFacts).
```

This code ensures that if we are told a list of elements, then we are told the head element and the tail elements. For example, $P \triangleleft (X_1, X_2, X_3, X_4)$ results in the conclusions $P \triangleleft X_1$ and $P \triangleleft (X_2, X_3, X_4)$. Continuing in this manner, we would eventually end up with P being told each of the elements in the original list. Using P1, we could then derive that P possesses each of these elements. To determine whether a principal possesses a list of formulae, we merely need to determine whether he possesses each formula in the corresponding list. The above approach to T4 can be similarly applied to P4, as well as C9, C17 and C19. One more predicate implementing T4 deserves mention:

```
rules :-
    fact(Premise, told(P, [Head | Rest]), _),
    (
        Head = identifyingSecret(S),
        Items = Rest;

        member(identifyingSecret(S), [Head | Rest]),
        delete([Head | Rest], identifyingSecret(S), Items)
    ),
    length(Items, LengthOfItems),
    (
        LengthOfItems =:= 1,
        getHead(Items, ItemHead),
        Conclusion = told(P, ItemHead);

        LengthOfItems > 1,
        Conclusion = told(P, Items)
    ),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    asserta(fact(NewIndex, Conclusion, reason([Premise], 'T4'))),
    asserta(addedFacts).
```

This version of T4 ensures that if $P \triangleleft (X_1, X_2, \ldots, < S >, \ldots, X_n)$, then $P \triangleleft (X_1, X_2, \ldots, X_n)$ results as a conclusion. The removal of the secret helps in the implementation of C4 and C13, since it leads to the principal possessing all of the formulae, except for the identifying secret, thus allowing the freshness and recognizability postulates to be applied to these remaining formulae as one single compound formula — recall that the freshness and recognizability postulates can only be applied if the formula under scrutiny is possessed. Let's now examine the code for implementing T5:

```
rules :-
    fact(Premise, told(P, function(_, Args)), _),
    (
        not(is_list(Args)),
        Conclusion = told(P, Args),
        not(fact(_, Conclusion, _)),
        getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
        asserta(fact(NewIndex, Conclusion, reason([Premise], 'T5'))),
        asserta(addedFacts);

        is_list(Args),
        assertSingleArgTold(Premise, P, Args, 0)
    ).
```

This code makes use of the `assertSingleArgumentTold/4` predicate if the argument to the function is a list, for example `function('F', ['X', 'Y'])`. However, if the argument is atomic, then it will be asserted that the principal who was told the function was also told the argument. So, if we have $P \triangleleft F(X)$, then the conclusion $P \triangleleft X$ would result. The `assertSingleArgTold/4` predicate appears as follows:

```
assertSingleArgTold(Premise, P, Args, ArgsIndex) :-
    (
        length(Args, LengthOfArgs),
        ArgsIndex >= 0, ArgsIndex < LengthOfArgs,
        nth0(ArgsIndex, Args, Nth0Item),
        delete(Args, Nth0Item, SubsetArgs),
        (
            Conclusion = told(P, Nth0Item),
            not(fact(_, Conclusion, _)),
            possessesAll(P, SubsetArgs, PossessionPremiseIndices),
            getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
            append([Premise], PossessionPremiseIndices, Indices),
            asserta(fact(NewIndex, Conclusion, reason(Indices, 'T5'))),
            asserta(addedFacts);

            true
        ),
        succ(ArgsIndex, NextArgsIndex),
        assertSingleArgTold(Premise, P, Args, NextArgsIndex)
    );
    true.
```

The `assertSingleArgumentTold/4` predicate loops through each of the elements in `Args`, using `ArgsIndex` to keep track of the current position. On each iteration, a new argument list is created with the ith element deleted from the original list, where i is the current position. If all of the elements in this new list are possessed, then it is asserted that the principal is told the deleted element. The

implementation for P5 is very similar, however, the `assertSingleArgPossessed/4` predicate is used to assert a formula as being possessed, rather than told. The `possessesAll/3` predicate is used throughout the implementation, so it would be apt to examine it briefly:

```
possessesAllHelper(_, [], []).

possessesAllHelper(P, [HeadItem | MoreItems], PossessionIndices) :-
    possessesAll(P, MoreItems, TempPossessionIndices),
    fact(Index, possesses(P, HeadItem), _),
    append(TempPossessionIndices, [Index], PossessionIndices).

possessesAll(P, List, PossessionIndices) :-
    flatten(List, FlattenedList),
    possessesAllHelper(P, FlattenedList, PossessionIndices).
```

If the principal `P` possesses all of the elements in `List`, then `PossessionIndices` will contain all of the `fact/3` indices proving this point. Finally, the source code for P10 appears as follows:

```
rules :-
    fact(Premise1, possesses(P, S), _),
    fact(Premise2, believes(P, secret(P, S, _)), _),
    not(
        S = shared(K);
        S = public(K);
        S = private(K)
    ),
    Conclusion = possesses(P, identifyingSecret(S)),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    PremiseIndices = [Premise1, Premise2],
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'P10'))),
    asserta(addedFacts).
```

The important point to notice here is that the side-condition specified in the GNY rule is being checked by ensuring that `S` is not a shared, public or private key. This technique is also applied in the source code for the freshness and recognizability rules, almost all of which have a side-condition.

5.2.3.2 Freshness and Recognizability Rules

The implementation of the freshness and recognizability rules is so similar that if one rule set has been constructed, then the other can be coded by merely copying the Prolog source and substituting freshness conditions for recognizability conditions, or visa versa. For now, let's focus on the implementation of the recognizability rules. The implementation of R1 and R2 is straight-forward and requires no further explanation. However, the remaining rules require more attention, since they have to cater for both lists and atomic arguments. Consider the source code for R3′:

```
rules :-
    fact(Premise2, possesses(P, List), _),
    listMemberIsRecognizable(P, List, Premise1),
    Conclusion = believes(P, recognizable(List)),
    not(fact(_, Conclusion, _)),
```

```
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    PremiseIndices = [Premise1, Premise2],
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'R3\'')),
    asserta(addedFacts).
```

This rule makes use of `listMemberIsRecognizable/3`, a predicate that is used in most of the recognizability rules as well as the conveyance rules:

```
listMemberIsRecognizable(P, List, RecognizabilityIndex) :-
    recursiveMember(X, List),
    fact(RecognizabilityIndex, believes(P, recognizable(X)), _).
```

Notice that use is made of a predicate named `recursiveMember/2`. This predicate is similar to the built-in `member/2` predicate, however, it recursively checks a list to determine whether a given argument is a member thereof. For example, the following clauses are all true:

```
recursiveMember(x, [x, y])
recursiveMember(x, [[x, y], a])
recursiveMember(x, [[b, [x, y], c], a])
recursiveMember([x, y], [b, [x, y]])
```

So, `listMemberIsRecognizable/3` operates by determining whether the argument `List` contains an embedded formula which is recognizable. If it does, then `List` is also recognizable. Combining this condition with a possession requirement for `List` yields R3′. R3″ is very similar:

```
rules :-
    fact(Premise2, possesses(P, function(F, Args)), _),
    (
        fact(Premise1, believes(P, recognizable(Args)), _);
        listMemberIsRecognizable(P, Args, Premise1)
    ),
    Conclusion = believes(P, recognizable(function(F, Args))),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    PremiseIndices = [Premise1, Premise2],
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'R3\'\'')),
    asserta(addedFacts).
```

However, because a function can either contain a list or an atom as an argument, we have to include the two clauses which determines whether the argument is recognizable as it stands. The implementations of R5, R6, R7, R10 and R11 are all very similar. Below is the source code for R6:

```
rules :-
    fact(Premise4, possesses(P, encrypt(X, public(K))), _),
    (
        not(is_list(X)),
        fact(Premise2, possesses(P, X), _),
        PossessionIndices = [Premise2];

        is_list(X),
        possessesAll(P, X, PossessionIndices)
    ),
```

```
    fact(Premise3, possesses(P, public(K)), _),
    (
        fact(Premise1, believes(P, recognizable(X)), _);
        listMemberIsRecognizable(P, X, Premise1)
    ),
    not(X = encrypt(_, private(K))),
    Conclusion = believes(P, recognizable(encrypt(X, public(K)))),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    append([Premise1], PossessionIndices, TempPremiseIndices),
    append(TempPremiseIndices, [Premise3, Premise4], PremiseIndices),
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'R6'))),
    asserta(addedFacts).
```

Notice that a check is carried out to determine whether the argument to the encryption is an atom or a list. If it is a atomic, then we just need to determine whether it is possessed and recognizable. However, if the argument is a list then we need to ensure that all of the elements embedded within it are possessed and that at least one of these embedded elements is recognizable. The code for R4, R8 and R9 is slightly simpler. Below is the source for R4:

```
rules :-
    fact(Premise1, believes(P, recognizable(X)), _),
    fact(Premise2, possesses(P, shared(K)), _),
    fact(Premise3, possesses(P, encrypt(X, shared(K))), _),
    not(X = decrypt(_, shared(K))),
    Conclusion = believes(P, recognizable(encrypt(X, shared(K)))),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    PremiseIndices = [Premise1, Premise2, Premise3],
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'R4'))),
    asserta(addedFacts).
```

In the vast majority of GNY-based protocol analyses, a principal obtains possession of an encrypted formula via a received message. This essentially means that if $P \ni \{X\}_K$, $P \ni \{X\}_{+K}$ or $P \ni \{X\}_{-K}$ hold, then the likelihood of $P \lhd \{X\}_K$, $P \lhd \{X\}_{+K}$ or $P \lhd \{X\}_{-K}$ also being true is quite high. In fact the possession statements were most probably derived through the application of P1. Thus, by applying T6, T7 or T8, followed by P1 to one of the told statements, the statement $P \ni X$ would result, assuming that the the correct decryption key was possessed. So, in the case of R4, R8 and R9, this means that if X is a list, then P will most likely possess all of the list elements as one single compound formula. Thus, the recognizability of X will be tested at some point in time through the application of the recognizability rules, and as a result, we don't need to determine whether a given formula embedded in the list is recognizable by calling `listMemberIsRecognizable/3`.

The implementation of the freshness rules uses exactly the same principles as those employed in the recognizability rules. In fact, F1 through to F10, as well as F19 and F20 are each structurally identical to one of the recognizability rules. The only difference lies in whether the term freshness or recognizability is used in the premises or the conclusion. The remaining freshness rules are straight-forward to implement based on the information that has been presented in this section.

5.2.3.3 Conveyance Rules

The conveyance rules take up a substantial amount of Prolog source code due to the fact that there are twenty-two of them and some of these can use up to four instances of the `rules` predicate. A number of the rules are also quite complex and have to cater for atomic and list-type arguments to encryptions and hashes. Rules C8, C9 and C17 through to C22 are straight-forward to implement based on what has already been explained in the previous sections. However, the implementation of the remaining rules will require some more explanation. The source code below is used to determine whether the first conclusion of C1, namely $P \mid\equiv Q \mid\sim X$, holds:

```
rules :-
    fact(Premise1, told(P, star(encrypt(X, shared(K)))), _),
    fact(Premise2, possesses(P, shared(K)), _),
    fact(Premise3, believes(P, secret(P, shared(K), Q)), _),
    fact(Premise4, believes(P, recognizable(X)), _),
    (
        fact(Premise5, believes(P, fresh(X)), _);
        fact(Premise5, believes(P, fresh(shared(K))), _)
    ),
    (
        not(is_list(X)),
        Conclusion = believes(P, conveyed(Q, X));

        is_list(X),
        length(X, LengthOfX),
        (
            LengthOfX =:= 1,
            getHead(X, Head),
            Conclusion = believes(P, conveyed(Q, Head));

            LengthOfX > 1,
            Conclusion = believes(P, conveyed(Q, X))
        )
    ),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    PremiseIndices = [Premise1, Premise2, Premise3, Premise4, Premise5],
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'C1'))),
    asserta(addedFacts).
```

This `rules` predicate is one of four that has been created to implement C1. Notice that in this code fragment the being-told premise is $P \lhd *\{X\}_K$, instead of $P \lhd *\{X\}_K \rightsquigarrow C$. The former premise was selected since the conclusion of the above instance of `rules` does not make use of any formula extensions. As a result, this rule can be applied to symmetrical encryptions with extensions as well as those without extensions, since those with extensions will have them removed through the application of T3. However, the second conclusion of C1, namely $P \mid\equiv Q \mid\sim \{X\}_K \rightsquigarrow C$, has an extension, and thus the `rules` instance which implements this case requires that the being-told premise include an extension. For completeness sake, an instance of the `rules` predicate which produces the conclusion $P \mid\equiv Q \mid\sim \{X\}_K$ has also been implemented, and in this case the being-told premise is extensionless. In fact, an extensionless being-told premise is also used for the instance of `rules` which determines whether the last conclusion of C1, namely $P \mid\equiv Q \ni (X, K)$, holds. Another important point to note is that the predicates `listMemberIsRecognizable/3` and `listMemberIsFresh/3` are

not invoked in the C1 implementation. This is due to the fact that the statement $P \ni X$ can be derived, and thus P possesses all of the formulae within X as one single compound formula if X is a list. Finally, notice that the Prolog code caters for both atomic and list-type elements. To illustrate this fact, consider the following being-told statements in Prolog-syntax:

```
told('P', star(encrypt('X', shared('K'))))
told('P', star(encrypt(['X'], shared('K'))))
told('P', star(encrypt(['X', 'Y'], shared('K'))))
```

Assuming that the remaining premises hold, then the conclusion produced by the application of C1 to each of the above statements is listed below:

```
believes('P', conveyed('Q', 'X'))
believes('P', conveyed('Q', 'X'))
believes('P', conveyed('Q', ['X', 'Y']))
```

Notice that a single-element list is reinterpreted as an atomic element. The reason for this is that in the context of communication protocols, the concept of a single-element list doesn't make too much sense, since a list of elements is usually considered to contain at least two components. Rules C10, C6 and C7 are structurally similar to C1 and employ all of the concepts which we have described. The following code fragment is taken from the `rules` predicate which determines whether the first conclusion of C3, namely $P \mid\equiv Q \mid\sim (X, S)$, holds:

```
fact(Premise1, told(P, star(hash(_, ArgsWithSecret))), _),
(
    ArgsWithSecret = identifyingSecret(S),
    Args = S,
    PossessionIndices = [];

    member(identifyingSecret(S), ArgsWithSecret),
    delete(ArgsWithSecret, identifyingSecret(S), Args),
    possessesAll(P, Args, PossessionIndices)
),
fact(PremiseIndex3, possesses(P, S), _),
fact(PremiseIndex4, believes(P, secret(P, S, Q)), _),
(
    listMemberIsFresh(P, Args, PremiseIndex5);
    fact(PremiseIndex5, believes(P, fresh(S)), _)
)
```

This code fragment checks whether all of the premises of C3 are true. As in the case with C1, the being-told premise only includes an extension if it will appear in the conclusion. Notice that the two arguments to the hash, namely X and S, are represented as one argument, namely `ArgsWithSecret`. If X is non-empty, then the variable `Args` is initialized with the contents of X, otherwise, `Args` contains S. Notice how the Prolog code ensures that S is always present as an argument to the hash and, more specifically, that it is being used as an identifying secret. An important point to note is that the code which implements checking of the C3 freshness premise makes used of the `listMemberIsFresh/3` predicate when examining X, since X is not necessarily possessed as a single compound formula. Now, consider the following being-told Prolog-style statements:

```
told('P', star(hash('H', identifyingSecret('S'))))
told('P', star(hash('H', [identifyingSecret('S')])))
told('P', star(hash('H', ['X', identifyingSecret('S')])))
told('P', star(hash('H', ['X', identifyingSecret('S'), 'Y'])))
```

Assuming that the remaining premises hold, then the conclusion produced by the application of C3 to each of the above statements is listed below:

```
believes('P', conveyed('Q', 'S'))
believes('P', conveyed('Q', 'S'))
believes('P', conveyed('Q', ['X', 'S']))
believes('P', conveyed('Q', ['X', 'Y', 'S']))
```

As in the case of C1, a single-element list is reinterpreted as an atomic element. Notice that a single-element list can only contain an identifying secret, otherwise the being-told premise of C3 will fail. The rules C5, C12 and C14 all have a similar structure to C3 and all employ the concepts which we have just described. The following fragment of code implements the freshness premise of C4:

```
length(Args, LengthOfArgs),
(
    LengthOfArgs =:= 1,
    getHead(Args, Head),
    fact(Premise6, believes(P, fresh(Head)), _);

    LengthOfArgs > 1,
    fact(Premise6, believes(P, fresh(Args)), _)
);
fact(Premise6, believes(P, fresh(S)), _);
fact(Premise6, believes(P, fresh(public(K))), _)
```

The variable `Args` contains all of the arguments to the public-key encryption, excluding the identifying secret, and is derived as illustrated in the code fragment for C3 shown above. An important point to note is that C4 only requires possession of a decryption key, and hence the the `possessesAll/3` predicate is not employed in its implementation. Also, due to the special instance of T4 described in Section 5.2.3.1 all of the formulae within `Args` are possessed as one single compound formula if `Args` is a list. As a result of this fact, the `listMemberIsRecognizable/3` and `listMemberIsFresh/3` predicates do not need to be used, since the recognizability and freshness of `Args` will be determined through the application of the recognizability and freshness postulates. The remainder of C4 is implemented using the techniques which have already been described in this section. C4 is also used as a model for other conveyance implementations, namely C13, C15 and C16, all of which employ the same source code structure. The following fragment of code is taken from the implementation of C2:

```
fact(PremiseIndex1, told(P, star(decrypt(Args, shared(K)))), _),
(
    not(is_list(Args)),
    fact(PremiseIndex2, possesses(P, Args), _),
    PossessionIndices = [PremiseIndex2];

    is_list(Args),
    possessesAll(P, Args, PossessionIndices)
),
```

```
fact(PremiseIndex3, possesses(P, shared(K)), _),
fact(PremiseIndex4, believes(P, secret(P, shared(K), Q)), _),
(
    fact(PremiseIndex5, believes(P, recognizable(Args)), _);
    listMemberIsRecognizable(P, Args, PremiseIndex5)
),
(
    (
        fact(PremiseIndex6, believes(P, fresh(Args)), _);
        listMemberIsFresh(P, Args, PremiseIndex6)
    );

    fact(PremiseIndex6, believes(P, fresh(shared(K))), _)
)
```

This code fragment determines whether all of the premises of C2 hold. Notice that the code checks whether `Args` is a list, and if it is uses the `possessesAll/3` predicate to ensure that all of the elements therein are possessed. Due to the fact that `Args` is not necessarily possessed as a single compound formula, the `listMemberIsFresh/3` and `listMemberIsRecognizable/3` predicates have to be used to determine whether an element of `Args` is fresh or recognizable respectively. However, just in case `Args` is possessed as a single compound formula, the freshness or recognizability check is also carried out by determining whether a `fact/3` predicate indicating freshness or recognizability of `Args` exists. C11 is implemented in the same way as C2.

5.2.3.4 Jurisdiction Rules

The implementation of the jurisdiction rules is straight-forward and holds no surprises. The code for J2 appears as follows:

```
rules :-
    fact(Premise1, believes(P, honest(Q)), _),
    fact(Premise2, believes(P, conveyed(Q, extension(X, C))), _),
    fact(Premise3, believes(P, fresh(X)), _),
    Conclusion = believes(P, believes(Q, C)),
    not(fact(_, Conclusion, _)),
    getMaxFactIndex(MaxIndex), NewIndex is MaxIndex + 1,
    PremiseIndices = [Premise1, Premise2, Premise3],
    asserta(fact(NewIndex, Conclusion, reason(PremiseIndices, 'J2'))),
    asserta(addedFacts).
```

The source code for J1 and J3 can be generated in a similar manner.

5.2.4 Coding the Proof Generator

Assume that we have analyzed a protocol and ended up with a number of derived statements. Now, only a subset of these derived statements will be of any use when proving the validity of any of the other statements. So, with this in mind, consider the following ordered subset of `fact/3` predicates, all of which contribute to the proof of the final statement:

```
fact(1, told('P', encrypt(['X', 'Y', 'Z'], public('K'))), reason([], 'Step')).
```

```
fact(2, possesses('P', private('K')), reason([], 'Assumption')).
fact(4, told('P', ['X', 'Y', 'Z']), reason([1, 2], 'T7')).
fact(12, told('P', ['Y', 'Z']), reason([4], 'T4')).
fact(16, told('P', 'Z'), reason([12], 'T4')).
```

Now, we want to produce a proof where each line is numbered in a sequential fashion, with the first line starting at 1. We also want to indicate which lines were involved in the application of a particular inference rule. So, essentially we want to re-map the `fact/3` indices and format the proof list into something which appears as follows:

```
1. told(P, encrypt([X, Y, Z], public(k)))  {Step}
2. possesses(P, private(K))  {Assumption}
3. told(P, [X, Y, Z])  {1, 2, T7}
4. told(P, [Y, Z])  {3, T4}
5. told(P, Z)  {4, T4}
```

If we translate the above Prolog statements into an English syntax we can end up with a more readable proof which a larger segment of people would more easily be able to interpret:

```
1. P was told E(+K : X, Y, Z).  {Step}
2. P possesses -K.  {Assumption}
3. P was told (X, Y, Z).  {1, 2, T7}
4. P was told (Y, Z).  {3, T4}
5. P was told Z.  {4, T4}
```

The above example illustrates the style and quality of the GNY proofs which the analyzer is able to generate. In the subsections that follow we will explain how we have coded the proof generation engine of the analyzer. The discussion will be illustrated with the code fragments that have been constructed to aid in this process.

5.2.4.1 Converting Prolog-Based GNY Statements into English

In the text that follows we partition all of the GNY statements into four classes. We will explain which statements fall into each class and how the conversion to English is coded for the class using the `prologGNYToEnglish/3` predicate. Note that in the code fragments a reference to the variable `Statement` denotes the Prolog-style GNY statement, while the resultant English text is returned in the variable `English`.

Class A: *Formulae Only*

The GNY statements $\sharp(X)$ and $\phi(X)$ fall into this category. The following code segment shows how the conversion is carried out for a statement regarding freshness:

```
Statement = fresh(X),
prologFormulaToStandardNotation(X, XStd),
(
    is_list(X),
    length(X, LengthOfX),
    (
        LengthOfX =:= 1,
        concat_atom([XStd, ' is fresh'], English);
```

```
            LengthOfX > 1,
            concat_atom(['(', XStd, ') is fresh'], English)
        );

        not(is_list(X)),
        concat_atom([XStd, ' is fresh'], English)
    );
```

The code fragment for statements regarding recognizability can be generated by simple substitution. In the code fragment above, the `prologFormulaToStandardNotation/2` predicate ensures that a formula is converted to an appropriate textual representation. Details on this predicate follow in Section 5.2.4.2.

Class B: *Principals and Formulae*

In this category we find the GNY statements $P \lhd X$, $P \ni X$, $P \mid\sim X$ and $P \vdash X$. The code segment illustrating how to carry out a conversion for a told statement follows:

```
Statement = told(P, X),
prologFormulaToStandardNotation(X, XStd),
(
    is_list(X),
    length(X, LengthOfX),
    (
        LengthOfX =:= 1,
        concat_atom([P, ' was told ', XStd], English);

        LengthOfX > 1,
        concat_atom([P, ' was told (', XStd, ')'], English)
    );

    not(is_list(X)),
    concat_atom([P, ' was told ', XStd], English)
);
```

In a similar way, we can code the conversion to English for the remaining statements in this category.

Class C: *Principals and Statements*

The GNY statements $P \mid\equiv C$ and $P \mid\Longrightarrow C$ fall into this category. The following code segment details how the conversion is carried out for a statement regarding a principal's beliefs:

```
Statement = believes(P, C),
(
    is_list(C),
    prologGNYListToEnglish(C, BelievesEnglish),
    concat_atom([P, ' believes that (', BelievesEnglish, ')'], English);

    not(is_list(C)),
    prologGNYToEnglish(noFullStop, C, BelievesEnglish),
    concat_atom([P, ' believes that ', BelievesEnglish], English)
);
```

The GNY controls statement can be coded in a similar way. Notice the reference to the predicate `prologGNYListToEnglish/2`. This predicate takes a list of Prolog-style GNY statements and then cycles through this list, converting each statement into its English equivalent.

Class D: *Principals and Atomic Formulae*

The last category contains the statements $P \mid\Longrightarrow P \mid\equiv *, \overset{+K}{\mapsto} Q$, $P \overset{S}{\longleftrightarrow} Q$ and $P \overset{K}{\longleftrightarrow} Q$. The code for converting all of these Prolog statements to English is shown below:

```
Statement = honest(P),
concat_atom([P, ' is trustworthy'], English);

Statement = public(K, P),
concat_atom(['+', K, ' is a suitable public key for ', P], English);

Statement = secret(P, S, Q),
concat_atom([S, ' is a suitable secret for use between ', P, ' and ', Q], English);

Statement = secret(P, shared(K), Q),
concat_atom([K, ' is a suitable secret for use between ', P, ' and ', Q], English);
```

Out of all of the classes, this is the most straight-forward to convert into English since no calls to helper predicates are necessary during a conversion.

5.2.4.2 Converting Prolog-Based GNY Formulae into GYPSIE Representation

The predicate `prologFormulaToStandardNotation/2` is used to convert a formula into its analogous GYPSIE representation. We use the GYPSIE notation since it is clear and easy to read. Maintaining consistency for integration with GYPSIE is also a priority. In the text that follows, we divide all of the GNY formulae into three classes and then elaborate on how each of the formulae in these classes are converted into GYPSIE notation. Note that the variable `Formula` represents the formula being converted and the variable `StandardNotation` contains the result of the conversion.

Class A: *Atomic Formulae*

The formulae in this class consist of secrets and keys, namely formulae of the form $< S >$, K, $+K$ and $-K$. Sample Prolog code to convert an identifying secret is shown below:

```
Formula = identifyingSecret(S),
concat_atom(['<', S, '>'], StandardNotation);
```

Of course, the other conversions are just as simple to implement.

Class B: *Compound Formulae*

The formulae in this class comprise of encryptions, decryptions, functions, hashes and formulae with a star. These are translated as expected, however the encryption syntax, which is similar to that in [23], might need some explanation:

Prolog Representation	*GYPSIE Notation*
`encrypt(X, shared(K))`	`E(K : X)`
`decrypt(X, shared(K))`	`D(K : X)`
`encrypt(X, public(K))`	`E(+K : X)`
`encrypt(X, private(K))`	`E(-K : X)`

With this in mind, sample code for converting a public key encryption is shown below:

```
Formula = encrypt(X, public(K)),
prologFormulaToStandardNotation(X, XText),
concat_atom(['E(+', K, ' : ', XText, ')'], StdNotation);
```

The other conversions are just as straight-forward. Notice that a recursive call takes place since the compound formulae contain other formulae that also need to be converted. In fact, one of the very first checks made by `prologFormulaToStandardNotation/2` is whether the formula to be converted is a list. If it is, then `listToStandardNotation/2` is called. This predicate takes a list as a parameter and then converts each formula in the list to its GYPSIE representation, returning the resultant list in the second argument.

Class C: *Formulae with Statements*

The final class consists of formulae with an attached extension. In this case the extension statement must be converted into English and the formula into its corresponding GYPSIE notation. These two results can then be combined to form the output.

5.2.4.3 Generating a Proof

Before we can generate a GNY proof for a derived statement, the subset of `fact/3` predicates that are involved in the proof must be extracted and sorted in ascending order by their indices. Any duplicates in this list must also be removed. To determine all of the `fact/3` indices involved, we use the `proof/2` predicate:

```
proof(Index, ProofList) :-
    fact(Index, _, reason(Premises, _)),
    append(Premises, ProofOfPremises, ProofList),
    prooflist(Premises, ProofOfPremises).

prooflist([], []).

prooflist([Premise | Premises], TotalProof) :-
    proof(Premise, ProofOfPremise),
    prooflist(Premises, ProofOfPremises),
    append(ProofOfPremise, ProofOfPremises, TotalProof).
```

The `proof/2` predicate appends the list of indices in the `fact/3` predicate denoted by `Index` to the list of indices generated by `prooflist/2`, returning the result in `ProofList`. In the same vein, the `prooflist/2` predicate takes a list of `fact/3` indices as an argument and then for each element in the list recursively extracts the `fact/3` indices that are involved in proving it, returning the result in `TotalProof`. This process ensures that all of the indices applicable to the proof for a given statement have been acquired. The list that is returned by `proof/2` is not sorted and may contain duplicates. However, the built-in `sort/2` predicate can be used to sort the list and remove duplicates at the same time. Re-mapping indices is accomplished by the `printShiftedReasonPremises/2` predicate, however, let's first examine a predicate that assists in this cause:

```
getIndex1(List, Element, Index) :-
    findIndex1(List, Element, 1, Index).

findIndex1([], _, _, ActualIndex) :-
    ActualIndex is -1.
```

```
findIndex1([Head | Tail], Element, TestIndex, ActualIndex) :-
    Element = Head,
    ActualIndex is TestIndex;

    succ(TestIndex, NextTestIndex),
    findIndex1(Tail, Element, NextTestIndex, ActualIndex).
```

The `getIndex1/3` predicate returns the index of an element in a list, the index position being based on a starting offset of 1. For example, in the list `[7, 8, 9]`, the indices of 7 and 9 are 1 and 3 respectively. This predicate is used extensively for determining the re-mapped premise indices when generating a GNY proof. Specifically, `getIndex1/3` is called from `shiftAndPrintReasonPremise/2`, which is in turn called from `printShiftedReasonPremises/2`:

```
shiftAndPrintReasonPremise(ReasonList, ReasonPremise) :-
    getIndex1(ReasonList, ReasonPremise, ShiftedPremise),
    write(ShiftedPremise), write(', ').

printShiftedReasonPremises(_, []).

printShiftedReasonPremises(ReasonList, [HeadReason | Reasons]) :-
    shiftAndPrintReasonPremise(ReasonList, HeadReason),
    printShiftedReasonPremises(ReasonList, Reasons).
```

The `printShiftedReasonPremises/2` predicate takes two arguments, the first argument containing a sorted list of all of the `fact/3` indices involved in a particular proof, and the second argument containing a subset of `fact/3` indices referenced in a particular line of the same proof. It then recursively cycles through each of the elements in the second list, using `shiftAndPrintReasonPremise/2` to re-map each element and write it to the output device. The re-mapping of the `fact/3` index is carried out by using the `getIndex1/3` predicate to determine the position of the index in the first list that was passed to `printShiftedReasonPremises/2`. For example, if the first list is `[3, 7, 17, 21]`, then this means that the `fact/3` indices involved in the proof are 3, 7, 17 and 21 and that fact 3 is found on line 1 of the proof, fact 7 on line 2, fact 17 on line 3 and fact 21 on line 4. Thus, using `getIndex1/3` to re-map the `fact/3` indices to their position in the first list yields the correct result. To write out an English-style proof with re-mapped indices we use the `writeOutProofInEnglish/3` predicate:

```
writeOutProofInEnglish(_, _, []).

writeOutProofInEnglish(Line, ReasonList, [Reason | Reasons]) :-
    fact(Reason, Statement, reason(ReasonPremises, Justification)),
    prologGNYToEnglish(Statement, EnglishStatement),
    write(Line), write('. '), write(EnglishStatement), write('  {'),
    (
        (Justification = 'Step'; Justification = 'Assumption'),
        write(Justification);

        printShiftedReasonPremises(ReasonList, ReasonPremises),
        write(Justification),
        true
    ),
    write('}'), nl,
    succ(Line, NextLine),
    writeOutProofInEnglish(NextLine, ReasonList, Reasons).
```

Finally, linking everything together is the `explainProofInEnglish/1` predicate which either takes a `fact/3` index or a statement in Prolog syntax as an argument and then generates a proof if the statement is valid:

```
explainProofInEnglish(Statement) :-
    fact(Index, Statement, _),
    explainProofInEnglish(Index).

explainProofInEnglish(Index) :-
    integer(Index),
    proof(Index, TempProofList),
    append([Index], TempProofList, UnsortedProofList),
    sort(UnsortedProofList, ProofList),
    writeOutProofInEnglish(1, ProofList, ProofList).
```

So, once an analysis has been completed, a proof can be generated in an English-style syntax by merely calling the `explainProofInEnglish/1` predicate. Two other predicates named `explainProof/1` and `writeOutProof/3` don't convert a proof into English but instead leave all of the GNY statements in Prolog.

5.2.5 Running the Analyzer

Before conducting an automated GNY-based analysis, the messages and initial assumptions pertaining to the protocol to be analyzed must be specified in a Prolog file in the form of `fact/3` predicates. Any target goals must also be specified in the same file by using `goal/2` predicates. The following is an example of how such an analysis specification file may appear:

```
% Idealized Protocol Steps:
fact(1, told('Q', 'Np'), reason([], 'Step')).
fact(2, told('P', star(encrypt('Np', shared('K')))), reason([], 'Step')).

% Initial Assumptions:
fact(3, possesses('P', shared('K')), reason([], 'Assumption')).
fact(4, believes('P', secret('P', shared('K'), 'Q')), reason([], 'Assumption')).
fact(5, believes('P', recognizable('Np')), reason([], 'Assumption')).
fact(6, believes('P', fresh('Np')), reason([], 'Assumption')).

% Protocol Goals:
goal(1, believes('P', conveyed('Q', 'Np'))).
goal(2, believes('Q', recognizable('Np'))).
goal(3, possesses('Q', 'Np')).
```

Every analysis specification file must also contain a `go/0` predicate. This predicate uses the built-in Prolog predicate `consult/1` to load the clauses contained in the source file *gny.pl*. This file contains the source code for the GYNGER analyzer. The other two predicates invoked in `go/0` are both defined within the GYNGER analyzer, `analyze/0` being used to begin the analysis process, and `printAnalysisResults/1` being used to display the results thereof. The code for `go/0` appears below:

```
go :-
    consult(gny),
    analyze,
    printAnalysisResults('results.txt').
```

Thus, the `go/0` predicate loads the predicates stored in *gny.pl*, carries out an analysis and then pipes all of the derived statements, as well as the proofs for the successful goals, to the file specified in the argument to `printAnalysisResults/1`. If a goal fails, then a proof cannot be generated, and the text 'FAILED!' appears instead of a proof. Output generated from the analysis of the above protocol definition appears below (for the sake of brevity, we have omitted most of the derived statements):

```
[1] Proof for P believes that Q once conveyed Np:
1. P was told *E(K : Np).  {Step}
2. P possesses K.  {Assumption}
3. P believes that K is a suitable secret for use between P and Q.  {Assumption}
4. P believes that Np is recognizable.  {Assumption}
5. P believes that Np is fresh.  {Assumption}
6. P believes that Q once conveyed Np.  {1, 2, 3, 4, 5, C1}

[2] Proof for Q believes that Np is recognizable:
FAILED!

[3] Proof for Q possesses Np:
1. Q was told Np.  {Step}
2. Q possesses Np.  {1, P1}

All Derived Statements:
P was told Np.
P was told E(K : Np).
Q was told Np.
P was told *E(K : Np).
  .
  .
  .
P believes that E(K : Np) is recognizable.
P believes that K is a suitable secret for use between P and Q.
P believes that Np is recognizable.
P believes that Np is fresh.
```

Once an analysis specification file has been defined, executing the analyzer requires the following command when using SWI-Prolog for Windows [83]:

```
plwin -f <analysis_file.pl> -g go -t halt
```

A Prolog system must obviously be available and the *gny.pl* containing GYNGER file must reside in the directory specified within the `go/0` predicate.

5.3 Experiments with the Analyzer

During the implementation of the analyzer, each rule was tested individually to ensure that it produced the correct results for different inputs. However, to test the analyzer under more 'realistic operating conditions' we decided to use it to analyze four cryptographic protocols of the calibre that would be found in real-world systems. Four protocols were tested in total. The Information Exchange and Authentication Protocols are taken from Chapter 4, while the Voting and Needham-Schroeder Protocols are from [33]. In the sections that follow we will give a formal description of each protocol, show the initial conditions that were used and the goals which the analyzer proved correct.

5.3.1 A Voting Protocol

This protocol can be used to coordinate elections in a distributed system. Principal Q coordinates the elections in which principals $P_1, \ldots, P_n$ take part. The coordinator shares a secret, S_i with each participant, and each participant is only allowed to vote once. Q determines the winner according to some agreed-upon procedure. The following idealized protocol definition was used by the analyzer:

(1) $Q \longrightarrow P_i : N_q$
(2) $P_i \longrightarrow Q : P_i, N_i, v_i, *H(N_q, < S_i >, v_i)$
(3) $Q \longrightarrow P_i : result, *H(N_i, < S_i >, result)$

The initial assumptions that were used for the analysis appear as follows:

Possessions: $P_i \ni (S_i, N_i, v_i)$; $Q \ni (S_i, N_q)$
Freshness: $P_i \mid\equiv \sharp(N_i)$; $Q \mid\equiv \sharp(N_q)$
Shared Secrets: $P_i \mid\equiv Q \overset{S_i}{\longleftrightarrow} P_i$ $Q \mid\equiv Q \overset{S_i}{\longleftrightarrow} P_i$

Finally, upon running the analyzer, 67 statements were derived after five passes through the GNY rule set. Four significant goals that were found to be true are shown below:

1. $Q \mid\equiv P_i \mid\sim v_i$
2. $P_i \mid\equiv Q \mid\sim result$
3. $Q \mid\equiv \sharp(H(N_q, < S_i >, v_i))$
4. $P_i \mid\equiv \sharp(H(N_i, < S_i >, result))$

The proof of these four goals may be found in Appendix B.

5.3.2 An Information Exchange Protocol

Information exchange protocols seek to securely transfer information between participants, ensuring that authentication of the sender, non-repudiation and integrity are achieved. The idealized protocol definition for the information-exchange protocol which we will analyze appears as follows:

(1) $A \longrightarrow B : *\{T_a, B, X, S\}_{K_{ab}} \rightsquigarrow A \overset{S}{\longleftrightarrow} B$
(2) $B \longrightarrow A : *H(X, < S >) \rightsquigarrow A \mid\sim X$

The initial assumptions that were employed for the analysis were as follows:

Possessions: $A \ni (T_a, B, X, S, K_{ab})$; $B \ni K_{ab}$
Recognizability: $B \mid\equiv \phi(B)$
Freshness: $B \mid\equiv \sharp(T_a)$; $A \mid\equiv \sharp(S)$
Shared Secrets: $A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$; $B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$; $A \mid\equiv A \overset{S}{\longleftrightarrow} B$
Trustworthiness: $B \mid\equiv A \mid\Rightarrow A \mid\equiv *$; $A \mid\equiv B \mid\Rightarrow B \mid\equiv *$
Jurisdiction: $B \mid\equiv A \mid\Rightarrow A \overset{S}{\longleftrightarrow} B$

Finally, once the analysis had been completed, the analyzer had derived 94 statements after six passes through the GNY rule set. Four significant goals that were found to be true appear below:

1. $B \ni X$ 2. $B \mid\equiv A \mid\sim X$
3. $A \mid\equiv B \mid\equiv A \mid\sim X$ 4. $A \mid\equiv B \ni X$

The proof for each of these four goals may be found in Appendix B.

5.3.3 An Authentication Protocol

Authentication protocols are normally run as precursors to communication sessions which require confidentiality. Once an authentication protocol has executed, the participants should be convinced of each other's identity and share a key which can be used to encrypt formulae to be transmitted. The idealized representation of the authentication protocol which the analyzer examined appears as follows:

(1) $A \longrightarrow B : A, N_a$
(2) $B \longrightarrow A : \{B, *\{N_a\}_{-K_b} \rightsquigarrow A \overset{K}{\longleftrightarrow} B, K, N_b\}_{+K_a}, *\{N_a\}_K$
(3) $A \longrightarrow B : *\{N_b\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B$

The initial assumptions that were used for the analysis appear below:

Possessions: $A \ni (A, N_a, -K_a, +K_b)$; $B \ni (B, N_b, +K_a, -K_b, K)$
Recognizability: $A \mid\equiv \phi(N_a)$; $B \mid\equiv \phi(N_b)$
Freshness: $A \mid\equiv \sharp(N_a)$; $B \mid\equiv \sharp(N_b)$
Public Keys: $B \mid\equiv \overset{+K_b}{\mapsto} B$; $B \mid\equiv \overset{+K_a}{\mapsto} A$; $A \mid\equiv \overset{+K_a}{\mapsto} A$; $A \mid\equiv \overset{+K_b}{\mapsto} B$
Shared Secrets: $B \mid\equiv A \overset{K}{\longleftrightarrow} B$
Trustworthiness: $A \mid\equiv B \mid\Rightarrow B \mid\equiv *$; $B \mid\equiv A \mid\Rightarrow A \mid\equiv *$
Jurisdiction: $A \mid\equiv B \mid\Rightarrow A \overset{K}{\longleftrightarrow} B$

Upon completion of the analysis, 105 statements had been generated after eleven passes through the GNY rule set. Six significant goals that were found to be true are listed below:

1. $A \ni K$ 2. $A \mid\equiv B \ni K$
3. $B \mid\equiv A \ni K$ 4. $A \mid\equiv A \overset{K}{\longleftrightarrow} B$
5. $A \mid\equiv B \mid\equiv A \overset{K}{\longleftrightarrow} B$ 6. $B \mid\equiv A \mid\equiv A \overset{K}{\longleftrightarrow} B$

A proof generated by the analyzer is given in Appendix B for each of these goals.

5.3.4 The Needham-Schroeder Protocol

The protocol presented in this section is based on the original Needham-Schroeder protocol [63] which has influenced the design of many authentication protocols. The protocol makes use of an authentication server S to provide two principals, P and Q, with a suitable session key. Below is the idealized representation of the protocol that was used by the analyzer:

(1) $P \longrightarrow Q : P$
(2) $Q \longrightarrow P : *\{P, N_{q_1}\}_{K_{qs}}$

(3) $P \longrightarrow S : P, Q, N_p, *\{P, N_{q_1}\}_{K_{qs}}$
(4) $S \longrightarrow P : *\{N_p, Q, K, *\{K, N_{q_1}, P\}_{K_{qs}} \rightsquigarrow P \overset{K}{\longleftrightarrow} Q\}_{K_{ps}} \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$
(5) $P \longrightarrow Q : N_{p_1}, *\{K, N_{q_1}, P\}_{K_{qs}} \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$
(6) $Q \longrightarrow P : *\{N_q, N_{p_1}\}_K \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$
(7) $P \longrightarrow Q : *\{F(N_q)\}_K \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$

The initial assumptions that were used in the analysis appear below:

Possessions:	$P \ni (P, Q, N_p, N_{p_1}, K_{ps});\ \ Q \ni (N_q, N_{q_1}, K_{qs});\ \ S \ni (K_{ps}, K_{qs}, K)$
Recognizability:	$P \mid\equiv \phi(N_{p_1});\ \ P \mid\equiv \phi(N_p);$
	$Q \mid\equiv \phi(N_{q_1});\ \ Q \mid\equiv \phi(N_q)$
Freshness:	$P \mid\equiv \sharp(N_{p_1});\ \ P \mid\equiv \sharp(N_p)$
	$Q \mid\equiv \sharp(N_{q_1});\ \ Q \mid\equiv \sharp(N_q)$
Shared Secrets:	$P \mid\equiv P \overset{K_{ps}}{\longleftrightarrow} S;\ \ Q \mid\equiv P \overset{K_{qs}}{\longleftrightarrow} S;\ \ S \mid\equiv P \overset{K}{\longleftrightarrow} Q$
Trustworthiness:	$P \mid\equiv S \mid\Rightarrow S \mid\equiv *;\ \ P \mid\equiv Q \mid\Rightarrow Q \mid\equiv *$
	$Q \mid\equiv S \mid\Rightarrow S \mid\equiv *;\ \ Q \mid\equiv P \mid\Rightarrow P \mid\equiv *$
Jurisdiction:	$P \mid\equiv S \mid\Rightarrow P \overset{K}{\longleftrightarrow} Q;\ \ Q \mid\equiv S \mid\Rightarrow P \overset{K}{\longleftrightarrow} Q$

After the analysis had run to completion, 215 facts had been generated as a result of eleven passes through the GNY rule set. Ten of these goals that merit mention are shown below:

1. $P \ni K$	2. $Q \ni K$
3. $P \mid\equiv P \overset{K}{\longleftrightarrow} Q$	4. $Q \mid\equiv P \overset{K}{\longleftrightarrow} Q$
5. $P \mid\equiv S \mid\sim K$	6. $Q \mid\equiv S \mid\sim K$
7. $P \mid\equiv Q \mid\equiv P \overset{K}{\longleftrightarrow} Q$	8. $Q \mid\equiv P \mid\equiv P \overset{K}{\longleftrightarrow} Q$
9. $P \mid\equiv Q \ni K$	10. $Q \mid\equiv P \ni K$

The proof for each of these goals may be found in Appendix B.

5.4 Closing Remarks

In this chapter we have described the GYNGER GNY analyzer which we implemented using approximately 3600 lines of Prolog source code. As a reference point for this implementation we used the publicly-available details of the GNY analyzer of Mathuria, Safavi-Naini and Nickolas [54]. However, because their analyzer only implemented a limited subset of the GNY rules and was not available for distribution, GYNGER was coded from scratch. The most significant feature of GYNGER is that it automates the tedious application of GNY inference rules, allowing all derivable GNY statements to be generated quickly, accurately and efficiently. To conduct an analysis with GYNGER a protocol engineer needs to specify a protocol's messages, initial assumptions and target goals in a Prolog-style GNY syntax. The GNY rule set is then imported and employed in the analysis, after which a proof is generated in an *English-style* syntax for every successful goal that was specified. This English-style proof lists all of the statements involved in the derivation of the successful goal, indicating the postulates that were used and the premises which were employed in the postulate's application. The fact that the proof is in an English-style syntax makes it more readable and comprehensible.

GYNGER incorporates all of the functionality of Mathuria, Safavi-Naini and Nickolas's analyzer, and improves on certain areas that were lacking. In particular, GYNGER implements the latest GNY rule set as specified in Gong's thesis [33], combined with improvements recommended in [53, 54] and some of our own additions. In total, GYNGER implements seventy-two GNY rules, while the analyzer of Mathuria, Safavi-Naini and Nickolas implements forty-eight. GYNGER also checks the side conditions that have been placed on some of the freshness and recognizability rules to ensure that sound conclusions are obtained. It is not known whether Mathuria, Safavi-Naini and Nickolas's analyzer examines these conditions as the tool could not be tested in any way. Each GNY rule implemented in GYNGER was tested individually by specifying all of the premises using `fact/3` predicates, running the analyzer, and then examining the results. Four cryptographic protocols were also analyzed, including the Needham-Schroeder protocol [63] and a voting protocol [33]. In each case, GYNGER was able to prove all of the goals that we had specified. For each goal that was proved GYNGER was also able to provide a corresponding English-style proof, something that the analyzer of Mathuria, Safavi-Naini and Nickolas could not do.

Certain GNY constructs are also represented differently by GYNGER when compared to Mathuria, Safavi-Naini and Nickolas's syntax. Since GYNGER uses characters enclosed in single quotes to represent formulae and principals, the original case is preserved. Mathuria, Safavi-Naini and Nickolas simply represent formulae and principals by lowercase characters, thus N_a and P are represented as `na` and `p` respectively. Another fundamental distinction is the manner in which functions and hashes are represented. GYNGER's syntax includes the name of the function or hash as a parameter, while Mathuria, Safavi-Naini and Nickolas hard-code the function name. For example, Mathuria, Safavi-Naini and Nickolas use the syntax `inc(x, delta)` to represent the function $inc(x, delta)$, while GYNGER represents it as `function('inc', ['x', 'delta'])`. A disadvantage of the former representation is that the GNY rules related to functions and hashes have to be continually upgraded to incorporate every function and hash that appears in a protocol, since the name of the function or hash cannot be a variable. A useful advantage of the latter representation is that multiple arguments are represented as a list, and thus the GNY rules don't need to be rewritten for functions and hashes with differing numbers of parameters.

In the description of their analyzer, Mathuria, Safavi-Naini and Nickolas fail to mention how they represent identifying secrets embedded in hashes and encryptions. These secrets are crucial to the operation of some of the conveyance rules. We have choosen to represent identifying secrets by using the notation `identifyingSecret('S')` to refer to $< S >$. When referring to a formula which has an extension, Mathuria, Safavi-Naini and Nickolas use the notation `ext(X, C)` to represent $X \rightsquigarrow C$. GYNGER uses the same technique, however, the keyword employed is `extension`. To represent a formula without an extension, Mathuria, Safavi-Naini and Nickolas use the representation `ext(X, nil)`, while we just write `X`. Although our representation is not as formal, it is less tedious from an end-user perspective. Since it is based on the latest version of GNY, GYNGER also implements the 'never-originated-here' binary operator, representing the statement $P \dashv (X)$ as `neverOriginated('P', 'X')`.

The notation of eligibility was added to the GNY logic to ensure the consistency of a protocol specification to be analyzed. However, the eligibility postulates are not conducive to forward-chaining since their application would result in an infinitely executing analysis. For this reason, neither GYNGER nor the analyzer of Mathuria, Safavi-Naini and Nickolas implements eligibility checking. This means that the consistency check for a protocol to be analyzed must be carried out by another analysis tool. To a large degree this situation reflects the current state of security protocol analysis, as there is no single tool or application that can examine all aspects of a protocol's security. For this reason, a multi-dimensional approach such as the one offered by SPEAR II is exceptionally useful, as it incorporates a diverse range of tools and techniques which all work together to examine different aspects of a protocol's security and

correctness. A possible addition to the SPEAR II Framework would be a system to examine protocol consistency using eligibility-type constructs. Thus, when using GYNGER within the SPEAR II environment, a protocol engineer would have the option of first checking the protocol's consistency and then performing an analysis with GYNGER.

Chapter 6

Visual GNY

"Cryptography is like literacy in the Dark Ages. Infinitely potent, for good and ill ... yet basically an intellectual construct, an idea, which by its nature will resist efforts to restrict it to bureaucrats and others who deem only themselves worthy of such Privilege."

— Vin McLellan

Before conducting a GNY analysis, a protocol engineer needs to create a protocol specification and determine its initial conditions, goals and formula extensions. If the analysis is to be automated, then this information must be supplied in a format accepted by the analysis system. However, supplying this representation of the information is not always a simple and straight-forward task and its prompt, efficient and error-free delivery often depends on the type of software being used.

Consider an analysis system which employs a text editor in combination with a Prolog analyzer such as GYNGER. In order to use this system, a protocol engineer must be acquainted with the Prolog syntax used to represent GNY statements. The necessary GNY statements must then be constructed in the text editor and the Prolog analyzer must be executed, using the text saved in the editor as input. Any syntactic errors that are detected during compilation must then be corrected. When experimenting with the GYNGER analyzer we used it in combination with a text editor and found that the following issues surfaced rather frequently:

- Incorrect formulae names, predicates and mismatched brackets were used.
- Construction of complex statements was cumbersome due to a lack of visualization aids.
- Lists of formulae were sometimes not enclosed in square brackets.
- Incorrect syntax was sometimes employed for encryptions, and functions.

Not all of these syntactic and semantic errors were detected by the Prolog compiler. In fact, we were often able to isolate many of these undetectable errors only because we knew what output the analyzer was meant to generate, since we had already carried out all of the analyses by hand. However, an automated analysis system is meant to remove manual analysis and not make it mandatory for error detection! Thus, based on our experience with GYNGER, we can conclude that a specification environment which eliminates the need for syntax recall and checks formula names for validity would aid greatly in making the construction of GNY statements simpler and less pain-staking. In fact, in an ideal world, an interleaving layer of software would always be present between the user and an analysis tool so as to facilitate the construction of syntactically correct input.

One of the most well-established findings in memory research is that people can recognize material far more easily than they can recall it [68]. This fact has clearly been applied in the design of graphical user interfaces over the past decade. For example, many user interfaces now make use of an extensive range of menus containing text or iconic lists of operations, options, files and so on. Instead of having to recall a name or a particular combination of function keys to perform an operation, users only need to scan through a menu until they recognize the name or the icon representing the operation which is required.

So, if we make use of a graphical user interface with sufficient cognitive aids and then combine this interface with the GYNGER analyzer, we could possibly improve the speed and quality of GNY analyses, while at the same time decreasing the difficulty which some individuals experience while conducting such analyses. The GYNGER analyzer already solves a large number of problems associated with the actual mechanics of an analysis, such as postulate application, constructing proofs for successful goals and determining goals which failed. However, what is still required is an environment from which to manage, structure, organize and conduct a GNY analysis.

The aim of this chapter is to describe the Visual GNY environment which we have created to facilitate GNY-based protocol analysis. We will briefly introduce the problems associated with manual GNY analysis, before moving on to describe two graphical interfaces for BAN and GNY-based analyses. Thereafter we will discuss the structured tree approach that we have implemented, followed by an overview of the Visual GNY environment. User experiments that were conducted in the environment will then be described. This will be followed by an examination of key portions of the Visual GNY implementation. In effect, we will attempt to show through the course of this chapter how the Visual GNY environment helps to distance protocol engineers from the syntactical element of GNY analysis, allowing them to focus more on the associated semantics and distil the critical issues that arise during protocol analysis.

6.1 GNY Analysis Difficulties

GNY analyses can appear complicated to the uninitiated, or non-mathematically inclined individuals. While teaching students in security courses at the University of Cape Town how to analyze protocols with GNY, we noticed that many of them balked or got bogged down in syntactic issues, instead of focusing on the actual analysis. This apprehension regarding the GNY syntax, as well as the size of the postulate set, often restrained individuals from effectively utilizing the logic to uncover protocol flaws. The following list describes some of the difficulties that people struggle with when they carry out GNY analyses by hand:

1. *Incorrect protocol parsing:* Parsing a protocol correctly is essential for ensuring that the conveyance rules can be properly applied. However, protocols are often parsed incorrectly by not inserting stars in the correct locations, or by inserting stars in front of incorrect or inappropriate formulae. A common mistake made in this area is that individuals forget to insert stars in front of formulae that are embedded within encryptions and functions.

2. *Semantically incorrect statements:* An understanding of the GNY syntax does not necessarily imply that semantically correct statements will be constructed. Two common semantic mistakes include writing the statement $A \mid\equiv \sharp(N_a, T_b)$ when $A \mid\equiv \sharp(N_a)$ *and* $A \mid\equiv \sharp(T_b)$ is intended, and using an incorrect or inappropriate type, such as a nonce or timestamp, in a shared secret or public key suitability statement. Semantic mistakes occasionally stem from transcription errors. However, their presence, more often than not, implies a lack of understanding regarding the fundamentals of GNY analysis.

3. *Transcription errors:* Because a GNY analysis often includes numerous derived statements and formulae, it is not difficult to commit transcription errors and copy information incorrectly between steps. These errors are then propagated through the remainder of the analysis, resulting in incorrect conclusions being derived. Because an analysis can sometimes be tedious, individuals also tend to lose interest, resulting in them losing perspective of what is taking place. Also, when an analysis spans multiple pages, as is often the case, flipping between these pages can cause frustration, causing errors to creep in.

4. *Syntactically incorrect statements:* Students usually take some time to fully grasp the notation and syntactic issues related to GNY analysis. What they essentially try to resolve when learning the syntax is how to translate a given GNY statement in mathematical notation into an understandable English statement which they can relate to their existing knowledge. GNY notation which normally causes difficulties includes the shared secret, public key and jurisdiction syntax, as well as the notation for indicating trust. We noticed that once students had carried out a number of analyses, their mastery of the syntax improved rapidly. However, this knowledge was usually forgotten after a few weeks.

5. *Confusion from information overload:* A standard GNY analysis often includes a plethora of initial beliefs, possessions and target goals to keep in check, not to mention all of the statements that are deduced from the application of the postulates. While utilizing these preconditions and derived statements one sometimes loses track of what is taking place in the analysis. A major defense against confusion and disorientation is structure and organization. However, when one is in a hurry or under pressure, beneficial practices are not always applied, resulting in GNY preconditions not being used and errors being made in proofs.

6. *Not working towards specific goals:* When carrying out a GNY analysis, it is vital to spend time beforehand contemplating what goals the protocol under inspection should achieve. Merely drafting some initial conditions and then blindly applying the postulates does not always produce useful results. For example, an analysis of the Needham-Schroeder protocol results in over two hundred derived statements, but only a handful of these actually tell us about what the protocol achieves (see Section 5.3.4 for details). Taking time out to consider the purpose of a protocol also helps to produce more meaningful and optimal initial belief and possession sets.

7. *Incorrect postulate application:* The GNY postulate set which we presently use contains seventy rules. Applying these postulates systematically can often be a tedious and error-prone process. Certain postulates are sometimes applied incorrectly. For example, we have encountered analyses where T2 is applied to a statement of the form $A \lhd *\{*\{X, N_a\}_K, T_s\}_{K_{as}}$ to produce $A \lhd \{\{X, N_a\}_K, T_s\}_{K_{as}}$, thus incorrectly losing the star attached to the embedded encryption. Sometimes individuals do not know of the existence of a particular postulate and do not apply it during a particular analysis. In fact, most people do not want to be bothered with searching through the entire postulate set during every step in an analysis to determine which postulates can be applied as it is such a tedious task. Also, when carrying out an analysis under pressure, a postulate might be applied even though its premises are not properly fulfilled.

GNY analysis has its usability issues, just like most other formal methods. However, by leveraging specially developed tools and techniques, a large portion of the difficulties that individuals encounter can be resolved. An automated analysis system such as GYNGER solves the transcription and postulate application problems — assuming that GNY statements have been input into the analyzer free from

errors. Parsing a protocol and inserting stars where necessary can be carried out by the environment wherein the protocol has been specified. The remaining issues, specifically enforcing syntax, structuring analysis information, and to some extent ensuring semantic correctness, can be supported through the use of a Visual GNY specification environment. The process of guiding a user in the choice of protocol goals is more complicated, but we will demonstrate how such a system can be incorporated within the Visual GNY framework.

6.2 Graphically Representing GNY Statements

As we have shown, conducting a GNY analysis manually is often tedious, error-prone and not very productive in the majority of situations. Analysis tools for GNY logic exist, however none of these tools has a graphical front-end to specify the GNY statements needed to define initial beliefs, possessions and goals. We want to create a system which most people who have been schooled in GNY at some point in time can walk up to and use almost immediately. These users must obviously understand the functioning and semantic issues associated with logic-based analysis and the graphical front-end to the analyzer should not impede them from effectively putting this knowledge into practice.

6.2.1 Objectives of a Graphical GNY Representation

A graphical environment supporting the creation of GNY statements should fulfil the following set of criteria. Some of these issues pertain to a GNY analysis environment, while others describe properties of the graphical representation:

1. The user should not be expected to recall the GNY syntax and associated mathematical symbols. However, she should have a working knowledge of GNY and be acquainted with the associated semantics.
2. The representation of GNY beliefs and possessions should be as concise as possible, yet it should be possible to easily view and construct all of the defined statements.
3. Belief and possession statements which have been constructed should be structured and organized so that it is easy to locate them for later modification or referral.
4. The specification environment must allow all possible GNY statements to be constructed. However, no syntactically incorrect statements or type errors should be permitted.
5. The efficiency and effectiveness of an individual using the environment should not be hampered or restrained in any way. The specification environment should also be intuitive and simple to use, requiring minimal keystrokes and mouse clicks.
6. There should be a clear separation between the initial conditions, extensions, goals and analysis results, with possessions and beliefs being distinguished within each of these categories.

To represent GNY statements, we have chosen to make use of a tree-based view with pop-up menus being used to add formulae, principals and belief categories. This approach imposes a hierarchical structure on the GNY statements and makes the representation of these statements as concise as possible. A tree-based view combined with pop-up menus also aids users in specifying GNY beliefs and possessions by ensuring that they do not have to recall any cryptic GNY syntax, notation or symbols.

6.2.2 Environments for Constructing BAN and GNY Statements

There are, with all likelihood, a number of graphical techniques which can be employed to specify GNY statements. Each of these will obviously have its own advantages and disadvantages. In this section we will examine two specification environments with which we experimented before developing our structured tree approach. While reading, bear in mind that the system which we wish to use for GNY statement specification must be simple to use, uncluttered, and able to provide the user with appropriate guidance so that syntactically correct statements employing the proper types can be constructed.

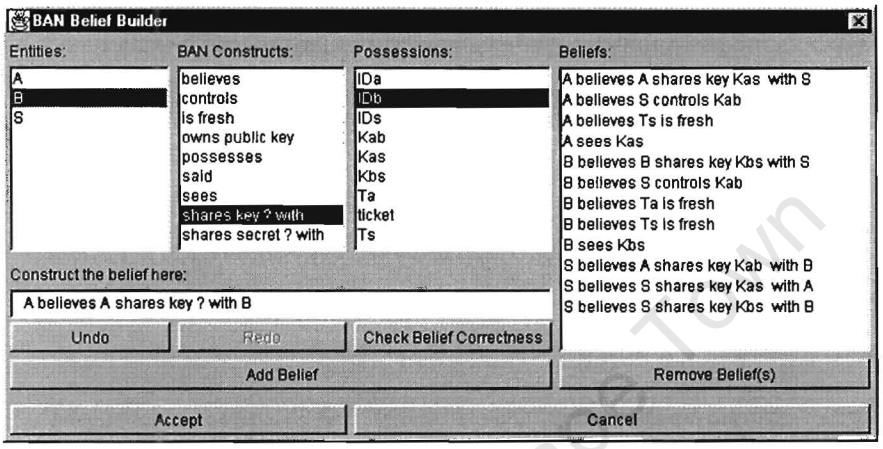


Figure 6.1: The SPEAR I BAN Builder dialog.

6.2.2.1 SPEAR I BAN Builder

The BAN Builder dialog, illustrated in Figure 6.1, is used in the SPEAR I application (described in Section 2.3) to specify initial BAN beliefs. A BAN statement is constructed by double-clicking on the text contained within the *Entities*, *BAN Constructs* or *Possessions* list boxes. When double-clicking the text, it is appended to the end of the statement contained within the edit box, situated below the list boxes. All of the BAN statements which have already been constructed are listed in the *Beliefs* list box. Beliefs are added to and removed from this list box by using the *Add Belief* and *Remove Belief(s)* buttons.

No guidance is currently provided within this environment to ensure that syntactically correct statements are constructed. In fact, the text within the three left-most list boxes can be double-clicked in any order, allowing meaningless statements to be produced. The *Check Belief Correctness* button can be used to initiate a BAN parser which examines the statement being constructed for syntactic correctness, producing appropriate error messages if it is malformed. However, it is not mandatory to perform a syntactic check, with the result that any arbitrary statement can be added to the list of beliefs.

A degree of guidance could be provided by enabling and disabling the three left-most list boxes appropriately. For example, when specifying the statement `Na is fresh`, only the *Entities* and *Possessions* list boxes should initially be enabled. Then, once `Na` has been added to the statement text, only the *BAN Constructs* box should be enabled so that the construct `is fresh` can be appended to the existing text. Upon completion of this statement, all of the list boxes should be disabled. Furthermore, within the *BAN Constructs* list box, selected constructs should be disabled or hidden so that a statement such as `Na said Kas` cannot be constructed. Inappropriate types within the *Possessions* list should also be rendered invisible or disabled.

A problem with this specification system emerges when statements such as `A believes A shares key Kab with B` are created. Essentially, the issue boils down to the fact that this statement is composed of five components, namely `A`, `believes`, `A`, `shares key Kab with`, and `B`. Now, the fourth component contains the name of the shared key, but this name is embedded within the text. Thus, the user has to edit the text manually so that the statement can be completed. In the interim, a question mark is displayed, as shown in Figure 6.1. Another problem is that there is no way of specifying a statement such as `(HalfOne, HalfTwo) is fresh`, without manually modifying the statement in the edit box.

Another issue at hand is that there is no clear, logical layout of the BAN statements within the *Beliefs* list box. All of the beliefs are just lumped together in no particular order. The statements within the list box could be ordered, however, this would require more list boxes to separate different categories, leading to more components within the dialog, thereby increasing the visual clutter. A tree-based view could be used to structure the beliefs into different categories, such as freshness, delegation and possession statements, thereby making it easier to access and view the different categories. Alternatively, tabbed panes could be used to distinguish these categories. The three left-most list boxes could also be replaced with a single contextualized pop-up menu, that presents only the appropriate options available when constructing a statement.

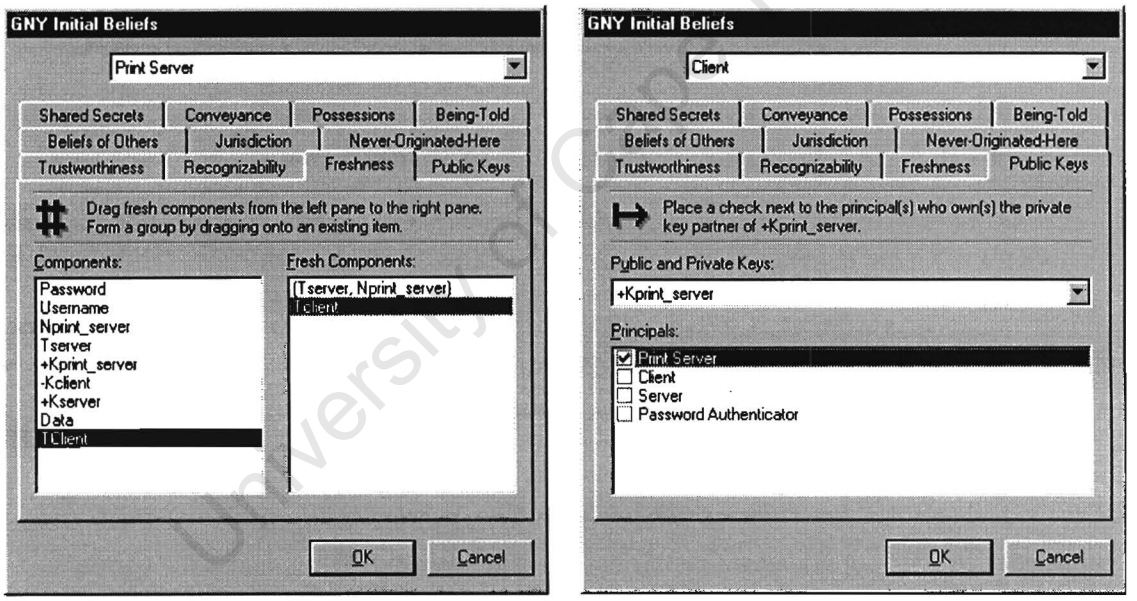


Figure 6.2: Two views from a tabbed pane-based GNY specification environment.

6.2.2.2 Tabbed Pane Dialog for Specifying GNY Statements

Another approach that can be used to specify GNY statements is a tabbed pane-based environment. Such a system consists of a number of tabbed panes within a dialog box, each tab representing a belief category. Within each of these tabbed panes the user provides the information that is required for a specific statement in that category. For example, a freshness pane would require that the formula considered to be fresh be specified, while a public key suitability pane would require that the public key and the principal who believes in its suitability be defined. Figure 6.2 shows two tabbed pane-based dialogs illustrating the layout of the freshness and public key belief categories.

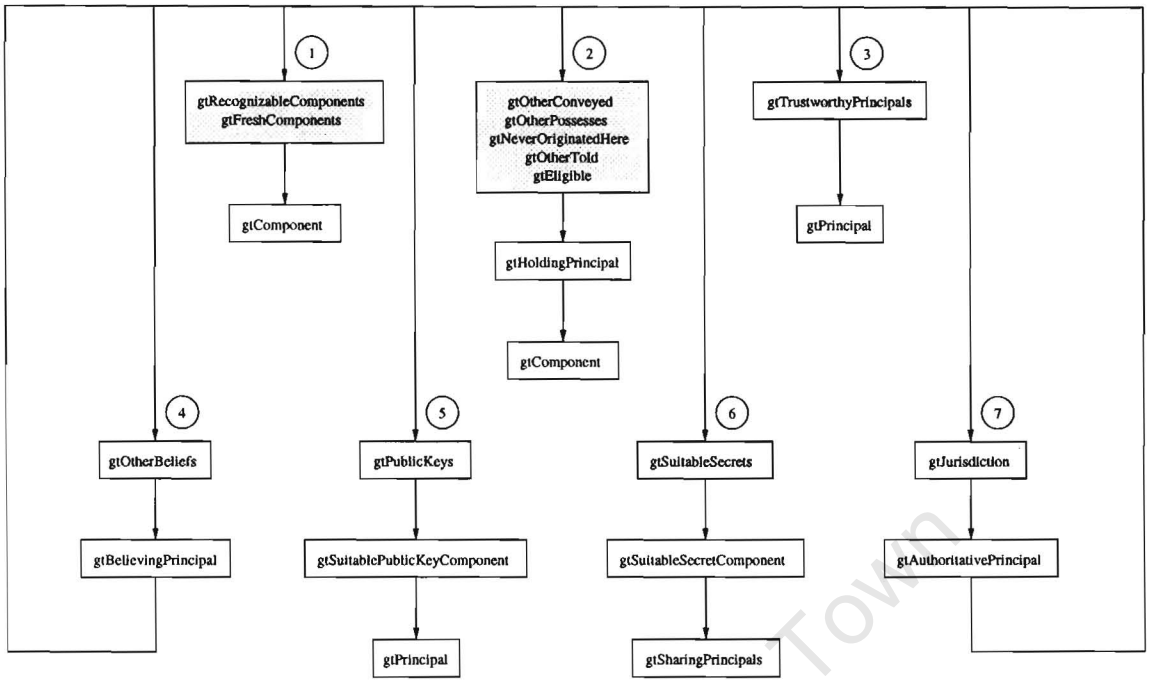


Figure 6.3: Relative hierarchy of tree nodes within our structured GNY tree representation.

In theory the tabbed pane-based environment is well laid out and logical, however, in practice it can be difficult to use as there is too much text, and far too many tabs. Also, as the number of GNY statements increases, some of the panes can become cluttered and confusing, depending on how they have been structured. In fact, a fundamental problem is that there is no uniformity in the layout of the panes since different panes will utilize different input techniques. Laying out all of the components within each pane is also quite a tedious task from a development and usability point of view.

However, a more significant difficultly emerges when representing jurisdiction statements and beliefs of other principals. Essentially, the question boils down to how we represent statements of the form $A \mid\equiv B \mid\equiv C$ or $A \mid\equiv B \mid\Rightarrow C$, where C can be *any* GNY statement. In effect, to allow for the specification of these statements, another tabbed pane dialog has to be invoked from the *Jurisdiction* or *Beliefs of Others* tabbed pane so that the nested statement can be constructed. This could result in a cascade of dialog boxes and a large amount of confusion for most users as the number of nested statements increases.

6.2.3 Using a Structured Tree to Represent GNY Statements

So far we have examined two specification environments, one for constructing BAN statements, and another for constructing GNY statements. The BAN specification environment allows a user to create BAN text strings by double-clicking BAN constructs contained within list boxes, while the tabbed pane-based environment utilizes forms within tabbed panes to gather the information needed to construct a given GNY statement. The GNY representation technique which we will now present allows any GNY statement to be viewed as a tree-like structure. Each node within this tree will have an assigned type, and its position within the tree will be determined by that type. We will see that this approach leads to a clean, concise, consistent and uncluttered graphical representation.

When using our structured tree approach, a set of GNY statements will be represented by of a collection of trees, each of the statements of a particular type belonging to the same tree. For example, a set of statements that only contains freshness and conveyance statements would be represented by two trees, each containing only freshness and conveyance statements respectively. Before we describe our structured tree approach any further, we need to partition all possible GNY statements into seven groups. All of the statements within a given group share the same structured tree representation.

Group 1: $\sharp(X), \phi(X)$
Group 2: $P \mid\sim X, P \ni X, P \dashv (X), P \lhd X, P \propto X$
Group 3: $P \mid\Rightarrow P \mid\equiv *$
Group 4: $P \mid\equiv C$
Group 5: $\overset{+K}{\mapsto} P$
Group 6: $P \overset{S}{\longleftrightarrow} Q$
Group 7: $P \mid\Rightarrow C$

The illustration in Figure 6.3 lists the GNY types in each group and the manner in which nodes belonging to a given group are structured. For example, when representing a statement in Group 5, the root node must be of type *gtPublicKeys*, followed by a child node of type *gtSuitablePublicKeyComponent*, terminating with a child node of type *gtPrincipal*. GNY types, prefaced by the letters *gt*, are used to identify tree nodes so that the correct captions, icons and pop-up menus will be displayed. Thus, if a node is of type *gtPrincipal*, then the principal's name and a principal icon will be used to represent the node in the tree-view. However, if a node is of type *gtFreshComponents*, then the text 'Fresh Components' will be used as the node caption and the icon for the freshness category will be displayed beside the text. A collection of graphical GNY statements is illustrated in Figure 6.4.

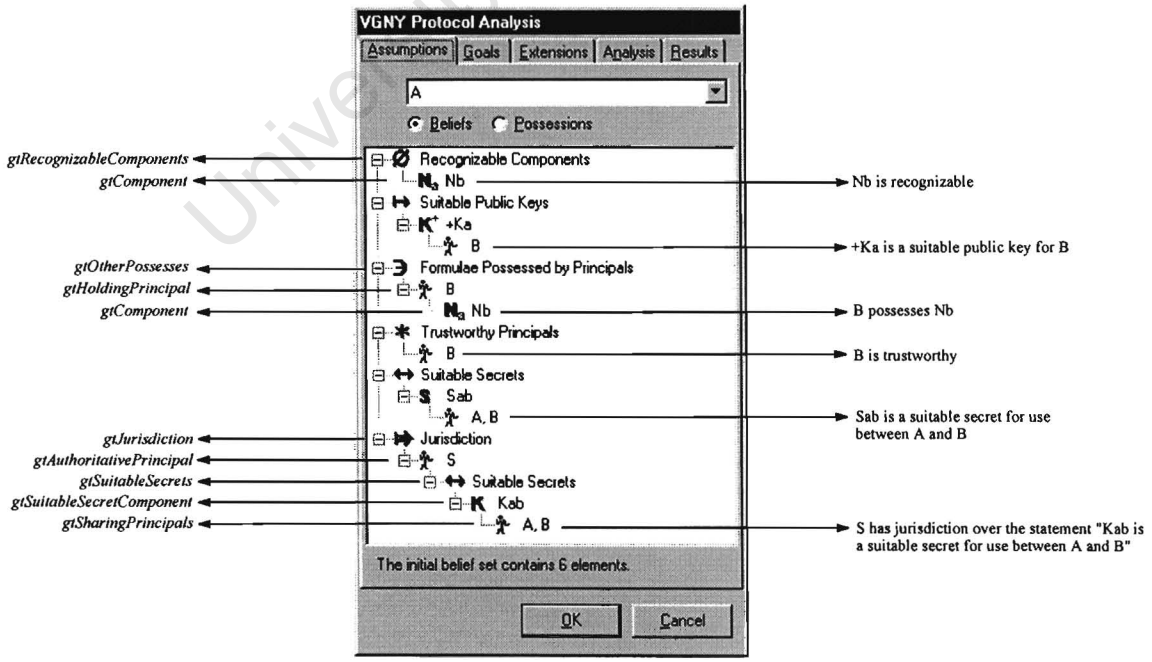


Figure 6.4: GNY statements specified in our structured tree-view.

The statements represented by the structured trees of Figure 6.4 form the initial belief set of principal A, as indicated by the tabbed pane, combo-box and radio button selections. Thus, every statement within these trees has the implicit prefix 'A believes that'. When using structured trees to store extensions, there is no implied prefix since an extension statement is merely an expression that must be believed to be true before a formula can be transmitted. An important point to note is that we do not require structured trees to specify the possession set for a given principal, since possession statements only have one operator and two operands, the left operand being the principal name and the right operand a formula. In fact, all that we require is a collection of one-node trees, each node having type *gtComponent*. Thus, when determining the GNY statement represented by a particular node in this collection of trees, we merely need to prefix the statement 'A possesses' to the name of the formula represented by each tree node.

Table 6.1 below lists the icons and captions which are used to represent nodes in a structured tree. In the interests of brevity we have omitted all of the component and principal types. The icon used for a component node is the same as that used for the corresponding formula in the GYPSIE component view, while the caption is the textual representation of the formula. In the case of a principal type, the picture of a person is used as the icon, while the caption contains the name of the principal.

GNY Type	Icon	Caption
gtFreshComponents		Fresh Components
gtRecognizableComponents		Recognizable Components
gtTrustworthyPrincipals		Trustworthy Principals
gtSuitablePublicKeys		Suitable Public Keys
gtSuitableSecrets		Suitable Secrets
gtOtherPossesses		Formulae Possessed by Principals
gtOtherConveyed		Formulae Conveyed by Principals
gtJurisdiction		Jurisdiction
gtOtherBeliefs		Beliefs of Principals
gtNeverOriginatedHere		Never Originated Here Components

Table 6.1: Icons and captions used for representing nodes in the structured tree.

At this point we should mention that the Visual GNY environment which we have created does not represent statements of the form $A \mid\equiv B \lhd X$ and $A \propto X$, even though our structured tree approach is capable of representing them. We have never encountered an expression of the form $A \mid\equiv B \lhd X$ in a GNY analysis. This expression is not used in any premises of the postulate set, neither is it the conclusion of any postulate. In fact, the expression $A \mid\equiv B \ni X$ is of greater significance and can be derived from $A \mid\equiv B \lhd X$ by using rationality and P1. Now, the statements in the Visual GNY environment will serve as input to the GYNGER GNY analyzer. However, GYNGER does not use eligibility statements, since it is a forward-chaining analyzer and the application of the eligibility postulates would result in an infinite loop. In fact, eligibility statements do not need to be specified in any principal's initial or target belief sets since they are only used during the GNY postulate application process to ensure consistency within a protocol specification.

6.2.4 Completeness of the Structured Tree Representation

We will now give an informal proof to show that any GNY statement can be represented by the structured GNY statement tree. We define the statement '$A \Rightarrow B \Rightarrow C$' to represent a tree where A is the root, B is a child of A and C is a child of B. So, assume that P and Q range over principals, C ranges over statements, X ranges over formulae, $+K$ is a public key and S is a shared secret. The $\square$ symbol ranges over all of the captions listed in Table 6.1, while $\star$ ranges over all GNY operators. From the context, it will be clear what $\square$ and $\star$ represent.

1. The eligibility, told, possession and conveyance statements all have the form $Q \star X$ and can be represented in the tree as $\square \Rightarrow Q \Rightarrow X$.
2. Freshness and recognizability statements both have the form $\star(X)$ and are displayed in the tree as $\square \Rightarrow X$.
3. The belief in the suitability of a secret shared between P and Q can be represented in the tree as $\square \Rightarrow S \Rightarrow (P, Q)$.
4. The belief in the suitability of a public key belonging to Q can be represented as $\square \Rightarrow +K \Rightarrow Q$.
5. The fact that principal Q is considered to be honest and competent can simply be represented as $\square \Rightarrow Q$ in the tree. Notice that the cryptic GNY syntax for this fact is totally removed.
6. Jurisdiction and belief statements both have the form $Q \star C$. Since C is a statement, it can be any one of those which we have already described in this informal proof. To represent jurisdiction or belief we merely use the arrangement $\square \Rightarrow Q \Rightarrow C$ in the tree.

Thus, all GNY statements can be represented in the tree by structuring nodes correctly. The enforcement of this structure is handled by pop-up menus and as a result only syntactically correct GNY statements can be generated.

6.3 Overview of the Visual GNY Environment

The structured tree approach which we have developed is implemented in the Visual GNY environment which is part of the SPEAR II Framework. For each principal within a given protocol specification, up to four sets of structured trees are created, two for the storage of initial beliefs and possessions, and another two for the storage of target beliefs and possessions. A further four sets of trees can be used to store the successful beliefs, successful possessions, failed beliefs and failed possessions for each principal upon the completion of a successful GNY analysis. A set of structured trees is also created for every formula that has extensions, these extensions being defined in the Visual GNY environment.

The Visual GNY environment runs in tandem with the GYPSIE protocol specification environment and the GYNGER protocol analyzer. Principals and formulae specified in GYPSIE are imported and used for constructing GNY statements, while completed GNY statements are exported to GYNGER for analysis. Results from completed GNY analyses are retrieved from GYNGER so that they can be partitioned and displayed appropriately. All of the GNY statements derived during an analysis, as well as the proofs for successful goals, are stored within the Visual GNY environment and are accessible through the *Results* pane, illustrated in Figure 6.5.

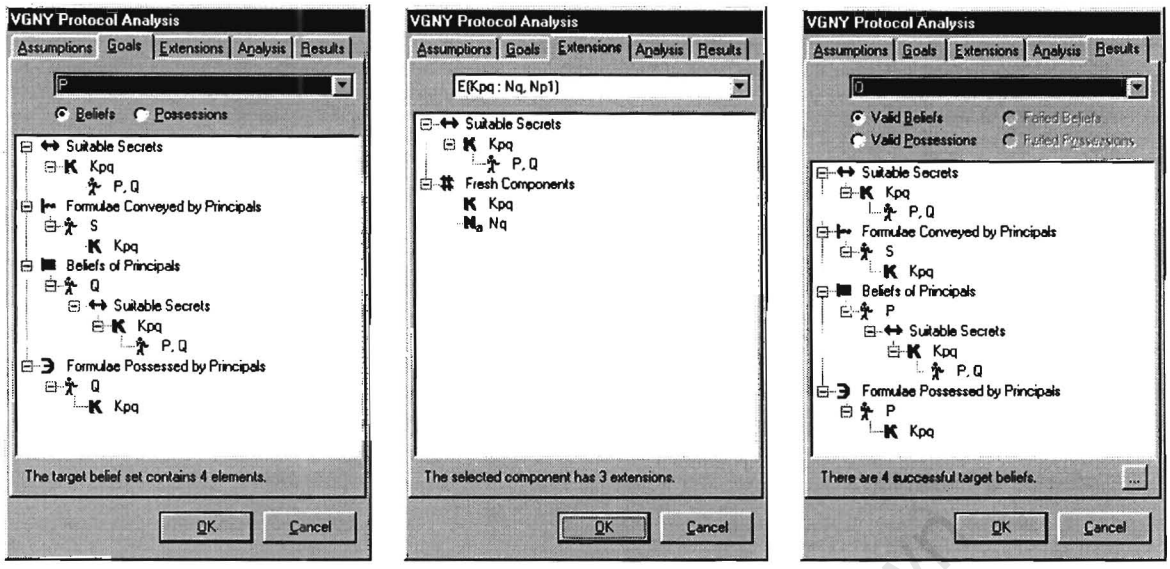


Figure 6.5: The Goals, Extensions and Results Panes from the Visual GNY environment.

6.3.1 The Visual GNY Interface

The Visual GNY interface is composed of five tabbed panes. Within each of these tabbed panes, a drop-down combo-box and a selection of radio buttons are used to select the appropriate set of structured trees to modify or view. The currently selected set of structured trees is displayed in a tree-view component centered within the client area of the tabbed pane. Changing either the combo-box or radio button selection changes the set of structured trees being displayed in the tree-view. A label situated below the tree-view indicates the number of GNY statements represented by the set of structured trees displayed in the tree-view. All interaction with the structured tree takes place through pop-up menus that are dynamically constructed depending on the selected tree node.

The *Assumptions* tabbed pane, illustrated in Figure 6.4, is used to specify the initial belief and possession sets of principals involved in a protocol. The *Goals* tabbed pane, displayed in Figure 6.5, is structured in the same way as the *Assumptions* pane, but is used to store the target belief and possession sets of a given principal. In both the *Goals* and the *Assumptions* panes, the radio buttons are used to switch between the belief and possession sets, while the combo-box is used to select the believing or possessing principal. The *Extensions* tabbed pane, also shown in Figure 6.5, allows a user to specify extension statements that are attached to a formula specified in the protocol messages. This tabbed pane does not include any radio buttons, as only one set of structured trees is ever used to store the extension statements attached to a formula. The combo-box is used to select the formula to which a user wishes to append extensions.

The *Analysis* pane allows one to invoke a GNY analysis using the GYNGER protocol analyzer. Within this tabbed pane information such as the location of the Prolog interpreter, working directories, results files and the location of the GNY rules in Prolog format must be supplied. Finally, the *Results* tabbed pane, shown in Figure 6.5, displays the outcome of a GNY analysis. Radio buttons are used to switch between the valid beliefs, valid possessions, failed beliefs and failed possession sets for the principal selected by the drop-down combo-box. If one of these sets is missing, then the radio button for that set is disabled, thereby allowing one to obtain a quick indication of what occurred during the analysis. The proof for a valid goal can be obtained by right-clicking its structured tree representation, while all of the derived statements are obtained by clicking the button in the lower right corner of the *Results* pane.

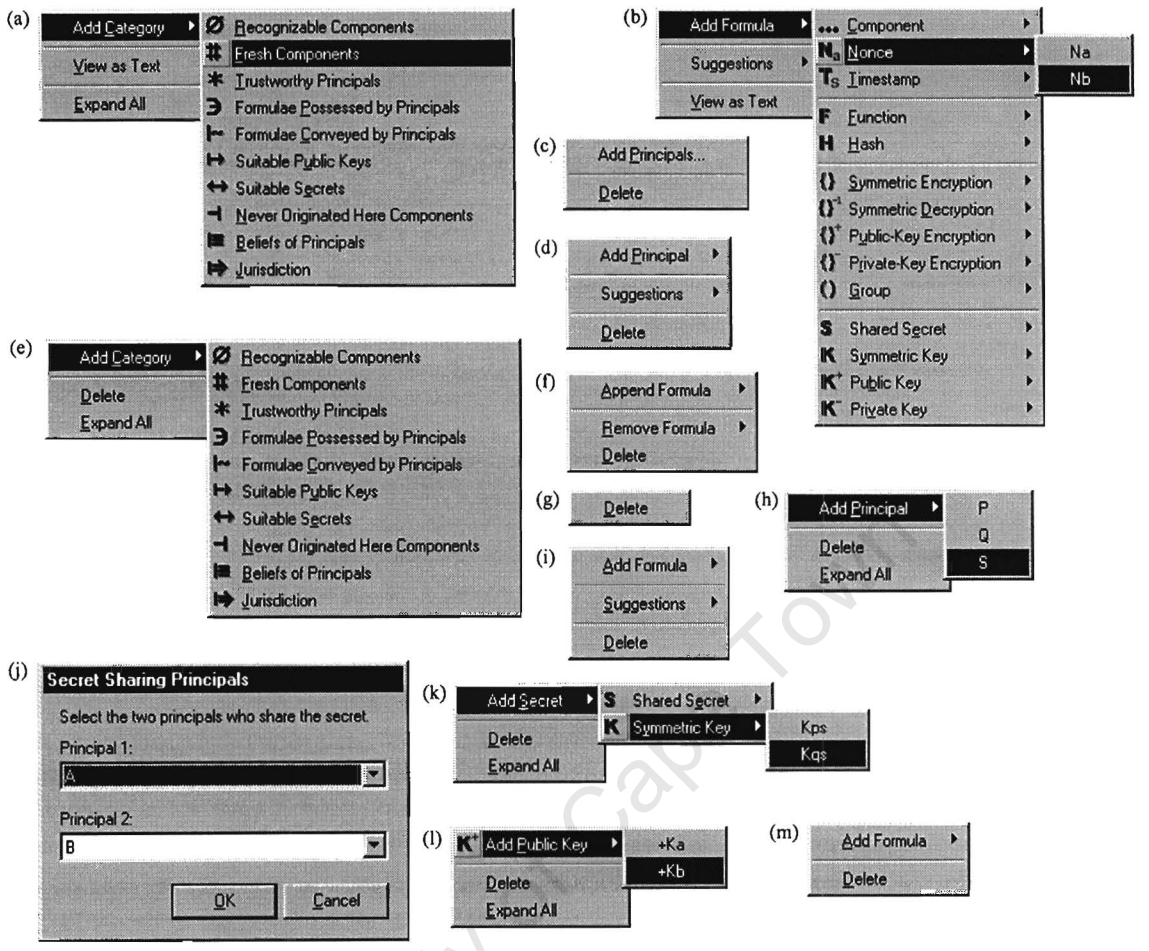


Figure 6.6: Pop-up menus and a dialog used in the Visual GNY environment.

6.3.2 Contextualized Pop-Up Menus

When using the Visual GNY interface, all that is required to specify a GNY statement is a pointing device, such as a mouse or trackball. The power of the Visual GNY interface stems from the fact that a user is 'guided' while constructing a GNY statement. This guidance is implemented through the use of pop-up menus. For example, to add the statement $P \mid\equiv Q \ni K_{pq}$ to principal P's target belief set, the user right-clicks in an open area of the tree-view representing P's target beliefs. A pop-up menu will then be displayed from which she can specify that she wants to add a 'Formulae Possessed by Principals' belief category. A parent-less tree node of type *gtOtherPossesses* will then be inserted in the tree-view. When right-clicking on this node, the user will be presented with a pop-up menu that contains a list of principals. She selects principal Q from the list and a tree node of type *gtHoldingPrincipal* is added to the tree-view, the node of type *gtOtherPossesses* being its parent. Then, when right-clicking on the node of type *gtHoldingPrincipalNode*, a list of formulae that have been specified in the GYPSIE environment is displayed, the formulae being sorted by their type. Upon selecting K_{pq} from the list of symmetric keys, a tree node of type *gtComponent* is added as a child of the node of type *gtHoldingPrincipal*. At this point, right-clicking on the formula node only presents a *Delete* option. Also, once the formula node has been added, the statement counter at the bottom of the dialog is updated.

The pop-up menus and dialog box used to construct a structured tree representing a GNY statement are presented in Figure 6.6. When right-clicking on a tree node, the pop-up menu created to service that node will be displayed. This pop-up menu selection is based on the GNY type of the tree node. In the list that follows we will describe each pop-up menu, and list the GNY types that it services.

(a) This pop-up menu is presented when right-clicking on an open area of space in a tree-view meant to contain a set of initial beliefs, target beliefs or extensions. The *Add Category* menu item is used to create and insert a structured tree root that will contain tree nodes making up statements of a specific type, such as freshness or conveyance statements. The *View as Text* command converts all of the the structured trees in the tree-view into English-style GNY text and then appends an implicit prefix to each, dumping the results in a dialog box containing a text viewer. Lastly, the *Expand All* item expands all tree nodes, ensuring that every one of their descendant nodes are visible.

(b) This pop-up menu is displayed when right-clicking on an open area of space in a tree-view meant to contain a set of initial or target possessions. The *Add Formula* command is used to add a formula to one of these possession sets. Notice how the existing formulae in the protocol are sorted by type using an extra layer of menus, so that it is easy to locate a given item. The same menu structure is used by the GYPSIE component view. The *Suggestions* menu-item contains a list of formulae which are likely to be in a principal's initial possession set. This saves the user from having to search through the formula list presented by the *Add Formula* command. Of course, this menu item is hidden when right-clicking in a target possession set tree-view. Lastly, the *View as Text* command displays all of the formulae in the possessions tree-view in a dialog box containing a text viewer, appending the applicable implicit prefix to each.

(c) This pop-up menu appears when right-clicking a tree node of type *gtSuitableSecretComponent*. The *Add Principals* command is used to invoke the dialog labelled as (j) in Figure 6.6, allowing a user to specify the two principals who share an identifying secret or symmetric key. The *Delete* menu item causes the node to be deleted.

(d) This pop-up menu is invoked when right-clicking on a node of type *gtTrustworthyPrincipals* or *gtSuitablePublicKeyComponent*. The *Add Principal* item presents a list of principals involved in the protocol. When selecting one of these principals, it is appended as a child to the node. The *Suggestions* menu item is only displayed when the tree node type is *gtSuitablePublicKeyComponent*. In this case, a list of principals who possess the private key partner of the public key node that was right-clicked is displayed. If there are no such principals, then the *Suggestions* menu item is hidden. Lastly, the *Delete* option allows one to remove the node.

(e) This pop-up menu appears when right-clicking on a tree node of type *gtBelievingPrincipal* or *gtAuthoritativePrincipal*. The *Add Category* menu item presents a list of GNY statement categories that can be added to the selected node, allowing the root of a new nested GNY statement to be created. Because we can add new belief categories in this fashion, the height of a node of type *gtBelievingPrincipal* or *gtAuthoritativePrincipal* can vary, while all other nodes must have a fixed height, since they do not include further nested statements. The *Delete* menu item causes the node and all of its descendants to be deleted, while the *Expand All* command expands all of the descendant tree nodes, ensuring that every descendant node is visible in the tree-view.

(f) This pop-up menu is displayed when right-clicking on a node of type *gtComponent*. Now, once a formula node has been added to the structured tree, it cannot grow vertically by gaining any children, however, it can still grow horizontally. Essentially, what we mean is that once a formula such as N_a has been added to the structured tree, another formula can be appended to form a compound formula such as $(N_a, Data)$. Thus, statements such as $P \mid\equiv \phi(X, Y)$ can be created, even if the compound formula (X, Y) does not exist in the protocol specification — only the components X and Y need to be defined. So, to append a formula to the node that was right-clicked, the *Append Formula* menu item is used. This menu item displays a list of formulae similar to the one used for the *Add Formula* menu item in (b). To remove a formula from a compound expression, the *Remove Formula* command is used. This command presents a list of the components within the compound expression, allowing each to be deleted individually. For example, if we right-clicked on the compound formula (X, Y, Z), then the *Remove Formula* menu item would contain the elements X, Y and Z respectively. If the formula that was right-clicked is not a compound formula, then the *Remove Formula* menu item is not displayed. A very important point to note is that the *Append Formula* menu item is *not* displayed when right-clicking on a possession node belonging to a target possession set. This is because GYNGER does not implement P3 and P6 through to P9 as they cause an infinite loop when analyzing protocols using forward chaining. Thus, all target possession statements are enforced to be of the form $P \ni X$, where X is defined within the protocol specification. The *Delete* command removes the formula node from the structured tree.

(g) This pop-up menu appears when right-clicking on a node of type *gtPrincipal* or *gtSharingPrincipals*. The only command present on this menu is the *Delete* command, which, as expected, deletes the principal node from a structured tree.

(h) This pop-up menu is invoked when right-clicking on a node of type *gtOtherPossesses*, *gtOtherConveyed*, *gtOtherBeliefs*, *gtJurisdiction* or *gtNeverOriginatedHere*. The *Add Principal* menu item presents a list of principals involved in the protocol. When selecting one of these principals, it is appended as a child to the node. The *Delete* command, deletes the principal node and all of its descendants from a structured tree, while the *Expand All* command expands all of the descendant tree nodes so that every one of these nodes is visible. Notice that this pop-up menu is similar to (d). However, because the *gtOtherPossesses*, *gtOtherConveyed*, *gtOtherBeliefs*, *gtJurisdiction* and *gtNeverOriginatedHere* nodes contain nodes that also contain children, the *Expand All* command is present.

(i) This pop-up menu is invoked when right-clicking on a node of type *gtRecognizableComponents* or *gtFreshComponents*. The *Add Formula* menu item displays a list of formulae similar to the one used for the *Add Formula* menu item in (b). When selecting a formula from this list, it is added as a child of the node which was right-clicked. The *Suggestions* menu item contains formulae that are likely candidates for freshness or recognizability in an initial belief set. Obviously, this set of formulae is based on the focused principal. The *Suggestions* menu item is hidden if there are no eligible formulae for suggestion, or the node which was right-clicked is part of a structured tree that represents an extension statement. Lastly, the *Delete* command removes the principal node from a structured tree.

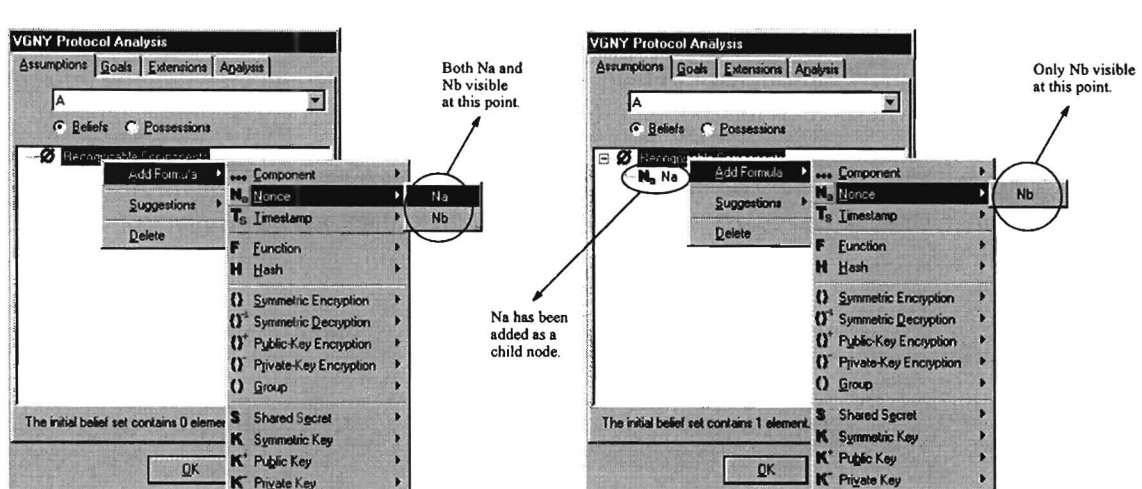


Figure 6.7: Illustration of the dynamic update of pop-up menus.

(k) This pop-up menu appears when right-clicking on a node of type *gtSuitableSecrets*. The *Add Secret* menu item contains a list of shared secrets and symmetric keys used within the protocol specification. These two formula types are partitioned to aid searching and semantic understanding. Because we only allow shared secrets and symmetric keys to be used within a shared secret suitability statement, type errors cannot be committed. The *Delete* menu item deletes the node and all of its descendants. The *Expand All* command expands all descendant nodes.

(l) This pop-up menu is used when right-clicking on a node of type *gtSuitablePublicKeys*. The *Add Public Key* menu item contains all of the public keys created within the protocol specification. When clicking on one of these public keys, it is added as a child node of the *gtSuitablePublicKeys* node. As in (k), because we only allow public keys to be used within a public key suitability statement, type errors cannot be committed. The *Delete* menu item deletes the node and all of its descendants. The *Expand All* command expands all descendant nodes.

(m) This pop-up menu is displayed when right-clicking on a node of type *gtHoldingPrincipal*. The *Add Formula* menu item displays a list of formulae similar to the one used for the *Add Formula* menu item in (b). Finally, the *Delete* menu item is used to remove the principal node from a structured tree displayed in the tree-view.

The dialog box labelled as (j) in Figure 6.6 is used to define the two principals who share an identifying secret or symmetric key. We have chosen to use a dialog box for this operation so that it can be completed in one step, as opposed to having to define each of the two principals separately using a pop-up menu. The dialog also prevents users from specifying identical names for the sharing principals by disabling the *OK* button if this is the case. Another way to ensure distinct principal names is by deleting the name of the highlighted principal in Combo-Box 1, from the principal list in Combo-Box 2, and vice-versa. Before this deletion takes place, the list must be updated with the names of all of the principals involved in the protocol. However, when using this approach a user will be prevented from swapping the order of principal names if there are only two principals in the specification, since Combo-Box 1 and Combo-Box 2 will only contain one principal name each.

An important point to note about the pop-up menus is that their content is updated dynamically. Consider the following example, illustrated in Figure 6.7. Assume we create a node with type *gtRecognizableComponents* that will serve as the root of a structured tree to store recognizability statements. Also, assume that there are two nonces defined in the protocol specification, namely N_a and N_b. When initially right-clicking on the *gtRecognizableComponents* node both nonces will be visible in the resultant pop-up menu. Assume that we click on N_a so that it is added as a child to the *gtRecognizableComponents* node. Now, the next time we right-click on the *gtRecognizableComponents* node only N_b will be displayed, since N_a has already been added to the *gtRecognizableComponents* node. Essentially, the principle used is that a formula, principal or belief category is only available for selection from a pop-up menu if it has not yet been added as a child of the node which was right-clicked. As a result of this fact, individuals cannot create duplicate GNY statements in a structured tree.

6.3.3 Enforcing Syntactic and Semantic Correctness

A significant advantage of the Visual GNY environment is that it ensures that syntactically correct statements are constructed. Because the pop-up menus enforce the predefined order of the nodes in the structured tree, it is not possible to specify a tree node that has an inappropriate type or to graft a node into an incorrect location. In fact, it is also not possible to specify an incomplete tree, since a user will not be allowed to change to another structured tree set, or press the *OK* button, unless all of the structured trees are complete. If a given tree is incomplete, then the node requiring a child will be highlighted and a message box indicating this fact will be displayed. The dynamic construction of the contextualized pop-up menus also ensures that no duplicate GNY statements can be generated. This elimination of duplication helps to ensure that an efficient and more optimal set of GNY statements are exported to the GYNGER analyzer. Besides this, it also helps to eliminate confusion by ensuring that there is only one copy of any given statement.

Enforcing semantic correctness is a lot more difficult than ensuring syntactic compliance. Some semantic checking has been added into the Visual GNY environment, but it is not capable of eliminating every semantic error. When specifying the principals who share a secret, the specification dialog ensures that the two principal names are distinct, thus preventing nonsensical statements, such as $A \mid\equiv B \overset{S}{\longleftrightarrow} B$ from being generated. Type correctness is enforced for the suitable secret and suitable public key statements by ensuring that a user is only able to use components of the appropriate type when constructing these expressions with pop-up menus. Because formula and principal names are imported from the protocol specification, all constructed GNY statements should refer to components that exist in the protocol specification. If a user imports a formula into a structured tree, and then removes all of its instances from the protocol specification, a bright red question mark is displayed as the structured tree node icon and the statement represented by the node is not considered as valid or exportable.

6.3.4 Exporting Visual GNY Statements

The ability to convert structured trees into a format which is compatible with an external GNY analysis tool or usable by a protocol engineer is fundamental to the operation of the Visual GNY environment. The structured trees defined within the Visual GNY interface can be exported to text, LaTeX and Prolog-style formats. The textual format displays each GNY statement in an English-style syntax, so that a statement such as $A \mid\equiv \sharp(N_a)$ is represented by the text string "A believes that Na is fresh". When exporting to LaTeX, each of the structured trees is translated into native GNY mathematical notation. As can be imagined, this feature is exceptionally useful for type-setting LaTeX documents which contain

GNY statements. Finally, the Prolog-style output is directly compatible with GYNGER, allowing all of the GNY statements constructed in the visual interface to be used for automated analysis without any tedious manual translation.

When constructing the export algorithms, we encountered some interesting issues related to the representation of GNY statements. These issues are very subtle, yet they need to be addressed to ensure that the output produced when mapping from the structured trees is both correct and usable. Consider the following set of GNY statements:

(1) $A \lhd *H(\{N_a\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B, < S >)$
(2) $A \ni (\{N_a\}_K, S)$
(3) $A \mid\equiv A \overset{K}{\longleftrightarrow} B$
(4) $A \mid\equiv \sharp(\{N_a\}_K)$

Now, we would like to apply C3, but this is not possible because the encryption formulae in the statements (2) and (4) do not contain an extension and thus do not match the encryption passed as an argument to the hash. This may sound pedantic, however, to an analyzer like GYNGER, issues such as this are crucial, as the extension cannot be dropped from the encryption in the hash by using any of the GNY postulates. Also, there is no postulate which allows us to add an extension to the encryptions in (2) and (4), and even if there was, a forward-chaining analyzer would not be able to use it as it would cause an infinite loop. So, the solution in this case is to append the extension attached to the encryption in (1) to the all of its instances, resulting in the encryptions in statements (2) and (4) each being augmented with the same extension.

So, in practice this means that when exporting GNY statements, any formulae which have extensions must be exported with these extensions attached. An extension cannot be removed from the formula to which it refers. During the export phase, the formula and the extension must be viewed as one inseparable item. Later, during an analysis, the extensions can be removed through the use of the appropriate GNY postulates, if possible. This principle works well in practice, however, there is a situation when it fails. Assume that an extension statement refers to the formula to which it is attached. If this is the case, then we obtain an infinitely long string referring to the formula. For example, if N_a has the extension 'N_a is fresh', then this would be represented as $N_a \rightsquigarrow \sharp(N_a \rightsquigarrow \sharp(N_a \rightsquigarrow \sharp(\ldots)))$!

There are two solutions to the self-referencing extension problem which we have just described. The first solution is to not allow a formula extension to contain references to the formula to which it is attached. This rule is simple to implement in Visual GNY by modifying the algorithms which dynamically update the pop-up menus. However, it is also rather limiting. We want to be able to construct formulae such as $N_a \rightsquigarrow \sharp(N_a)$, as it makes sense logically and semantically — N_a can only be transmitted if it is believed to be fresh by the sender. A second solution is to ban the use of extensions within extensions. We have implemented this approach within the Visual GNY environment and found that it works well, preventing circular references. Essentially, the choice as to which solution is used to prevent circular references boils down to a question of compromise. We have chosen to ban nested extensions, at the expense that certain esoteric analyses may not be possible.

There are two cases in which we do not attach extensions to formulae when exporting structured GNY trees. The first is when a key is used in an encryption statement, such as $\{X\}_K$. In this case the formula representing the encryption key, K, does not have an attached extension. However, the remaining instances of K which are not used in encryption statements would include an extension if there was one, for example $A \mid\equiv \sharp(K \rightsquigarrow B \ni K)$. The second instance in which we do not use extensions is in the case

of public key suitability statements, such as $A \mid\equiv \overset{+K}{\mapsto} B$. In this situation, even if $+K$ has an extension, we do not write it. However, this is the only case in which $+K$ is written without its extension. In other statements, such as $A \mid\equiv \phi(H(X, +K \rightsquigarrow B \mid\sim X, Z))$, the extension is included.

Now, a natural question to ask is whether our approach of always attaching extensions, except in the situations outlined above, results in analysis difficulties. The answer is that with the existing GNY postulate set, resolution is facilitated, as there are laws that allow for extensions to be dropped. In other situations, such as formulae embedded within hashes, keeping extensions does not result in difficulties. Consider the following two sets of GNY statements:

Set 1:	*Set 2*:
$P \lhd *\{X\}_K$	$P \lhd *H(X, < S \rightsquigarrow C'' >)$
$P \ni K \rightsquigarrow C'$	$P \ni (X, S \rightsquigarrow C'')$
$P \mid\equiv P \overset{K \rightsquigarrow C'}{\longleftrightarrow} Q$	$P \mid\equiv P \overset{S \rightsquigarrow C''}{\longleftrightarrow} Q$
$P \mid\equiv \phi(X)$	$P \mid\equiv \sharp(S \rightsquigarrow C'')$
$P \mid\equiv \sharp(K \rightsquigarrow C)$	

In the analysis on the left, we have to drop extension C' so that C1 can be applied. The extension is dropped as the representation of the symmetric key K must be consistant throughout, especially for an analyzer such as GYNGER. The postulates used to drop the extensions are P2, C22 and F2. In the analysis on the right, no extensions need to be dropped and C3 can be applied immediately. To see that leaving the extension out for public key suitability states does not result in problems, one merely needs to examine the postulates that employ this construct, viz C4 through to C7, C13 and C14. Finally, as a slight aside, let us clarify the GYNGER-compatible Prolog-style syntax for two GNY constructs that Visual GNY produces:

- `hash('H', ['X', identifyingSecret(extension('S', fresh('X'))) ])` is the same as $H(X, < S \rightsquigarrow \sharp(X) >)$.
- `extension(star(encrypt('N_a', shared('K'))), secret('A', 'K', 'B'))` corresponds to $*\{N_a\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B$.

Now, let's briefly examine how the Visual GNY environment exports GNY statements to the text, LaTeX and Prolog formats which it supports. Obtaining the GNY statements from the structured trees is accomplished through calls to the appropriate structured tree APIs, which will be described later. The exported output contains all of the defined GNY statements and analysis results, if any, in the order that follows: initial belief statements, initial possession statements, target belief statements, target possession statements, valid possession statements, failed possession statements, valid belief statements and valid possession statements. As we have mentioned, extensions are appended to formulae for which an extension set has been created in the *Extensions* tabbed pane.

Along with the GNY statements defined in the Visual GNY environment, the list of protocol messages is also output using being-told statements. This list is created through interaction with the GYPSIE protocol specification environment wherein the messages, their receivers and relative order are defined. The GYPSIE API calls allow this protocol output to be generated with or without stars. In fact, the protocol parsing and appending of stars is all carried out by GYPSIE since it is easily automated. No user-interaction is required for protocol parsing, ensuring that this analysis phase is free from errors.

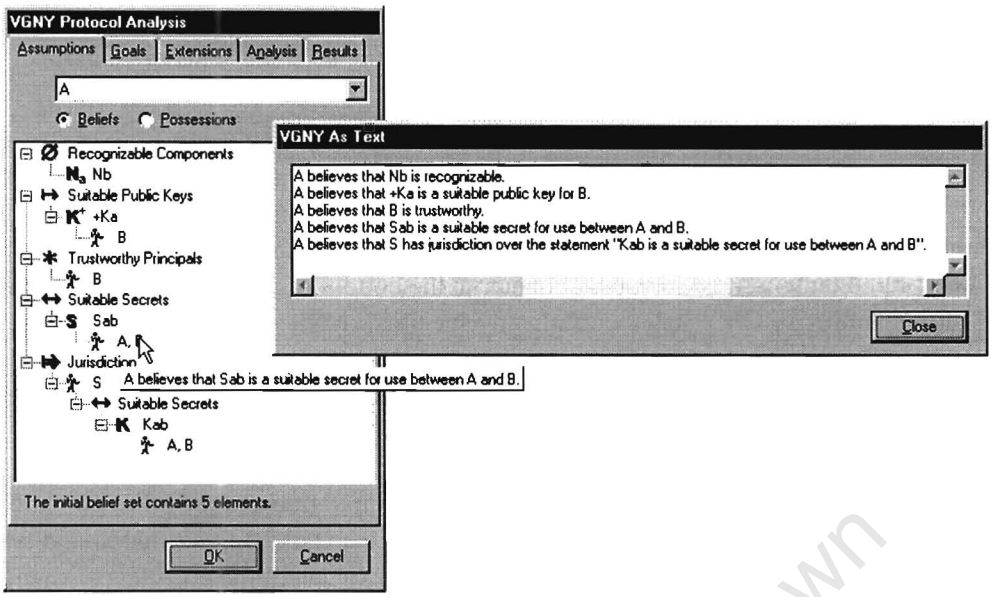


Figure 6.8: The *View as Text* facility and Visual GNY tooltip cue in action.

A subtle point to consider is that stars cannot be removed from a formula if it is embedded in a hash. Consider the following set of GNY statements:

(1) $A \lhd H(*\{X\}_K, N_a)$
(2) $A \ni (\{X\}_K, N_a)$
(3) $A \mid\equiv \sharp(N_a)$

Applying F3′ to (2) and (3), results in $A \mid\equiv \sharp(\{X\}_K, N_a)$, while applying P1 to (1) gives us the statement $A \ni H(*\{X\}_K, N_a)$. However, we cannot apply F4 as P possesses the compound formula, $(\{X\}_K, N_a)$ not $(*\{X\}_K, N_a)$, which is syntactically distinct. So, a way around this issue is to generate *two* lists of protocol messages in being-told format. The list first includes stars, while the second does not. This approach is used when generating Prolog output for GYNGER to ensure that the automated analysis derives all possible resultant GNY statements. However, in the text and LaTeX output, we only use the first list of messages which contains stars, as we do not expect this exported information to be used with any automated analysis tools.

6.3.5 Organizing and Managing Statement Construction

Within the Visual GNY environment, we have tried to create an environment that facilitates simple and straight-forward construction of GNY statements. Now, while working in one of the tree-views, a user might need to know the GNY statement which is represented by a specific set of nodes. To facilitate such a query, a feature which displays the GNY statement represented by a given node through the use of tooltips has been created. When hovering over a valid terminal node of a structured tree, the GNY statement which this node represents is displayed, as illustrated in Figure 6.8. This feature prevents users from having to navigate a tree and derive the GNY statement which it represents. Tooltips also give an indication of what the implicit prefix for a given set of structured is. For example, the implicit prefix of the structured trees illustrated in Figure 6.8 is 'A believes that'. To view all of the GNY statements

represented by the set of structured trees, a user can right-click in any open area and select the *View as Text* option from the resultant pop-up menu. A dialog containing an English-style list of GNY statements in an edit box is then displayed, as illustrated in Figure 6.8.

Another way in which the Visual GNY environment 'guides' a user is by helping her to structure and order the analysis process. The tabbed panes give an indication of the information required for an analysis, and are roughly laid out in the order that this information would be supplied. Belief and possession sets for a given principal are grouped into a single tabbed pane, only one set being visible at a time through the selection of radio buttons. The statement counter at the bottom of a tabbed pane also helps to give an indication of the number of GNY statements already specified. Nodes within a given structured tree can be expanded or collapsed as required. If a node contains children then a clickable token is displayed to its left. Clicking on this token allows the node to be collapsed or expanded, thus allowing a user to control the amount of information which is presented. In this way the level of detail provided by the interface can be varied appropriately, allowing a user to control any possible disorientation to some degree. Also, because only one type of statement occurs in any given structured tree, the user does not have to 'search around' for similar statements, as is the case with the SPEAR I BAN Builder dialog and some manual paper-based analyses. The *Analysis* results pane allows a user to determine at a glance whether there are any valid or failed possessions and beliefs. This is accomplished through the use of enabled and disabled radio buttons in the *Results* pane. All of these results sets can be viewed by selecting the appropriate radio button, if it is enabled.

6.3.6 Suggested Statement Completion

Within the Visual GNY environment, we have attempted to add some primitive intelligence to aid with the construction of GNY statements. This help is provided through the *Suggestions* menu field found on some of the pop-up menus. In the list below, we describe the rationale behind the suggestions we have automated:

- *Initial Possession Sets:* Anything that a principal originates is suggested to be part of his initial possession set. For example, if we have the message specification $A \longrightarrow B : X, \{Y\}_K, H(Z)$, then the components X, Y, $\{Y\}_K$, K, Z and $H(Z)$ are placed within the appropriate *Suggestions* menu item if these items were not received in a prior transmission.
- *Recognizable Formula:* Any nonce that a principal originates is suggested to be recognizable and placed within the appropriate *Suggestions* menu item.
- *Fresh Formulae:* Any timestamp that a principal receives or any nonce which he originates is suggested to be fresh and also inserted within the appropriate *Suggestions* menu item.
- *Public Key Owners:* Assume that a GNY possession statement exists which states that a given principal possesses a private key. Then, when constructing a suitable public key statement, the owner of the private key is also suggested to be the owner of the public key and placed within the appropriate *Suggestions* menu.

We have tried to provide hints that will not introduce errors into the GNY specification being constructed. As would be imagined, the addition of these suggestions, especially the initial possession set suggestions, help to speed up the creation of GNY analysis preconditions. In future, we envisage smarter heuristics being used in conjunction with more suggestions menus.

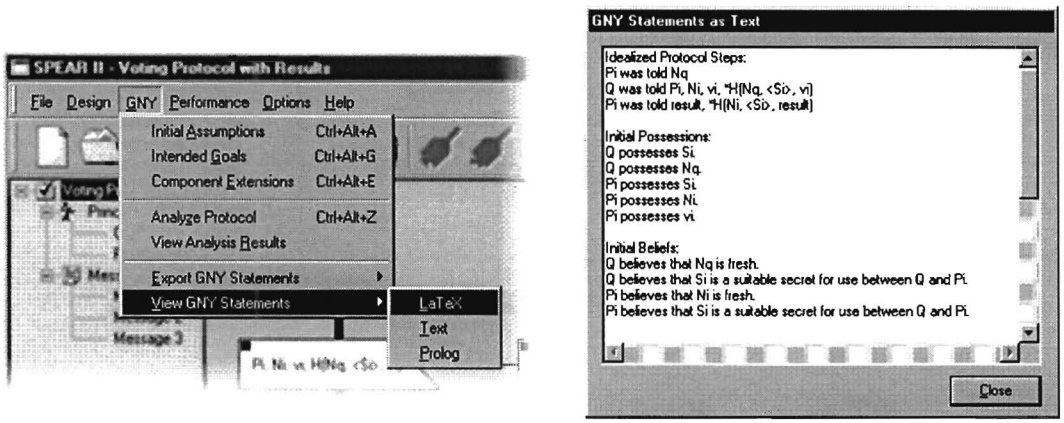


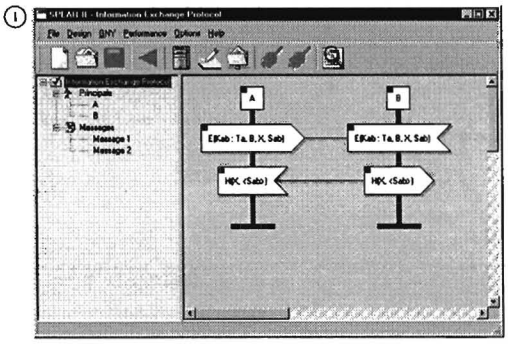
Figure 6.9: Pull-down menu commands and GNY statements being viewed as text.

6.3.7 Integration within the SPEAR II Framework

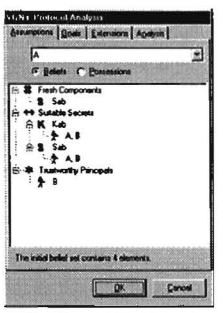
The Visual GNY feature set is accessible within the SPEAR II Framework through the use of pull-down menus. Within the main SPEAR II window, the *GNY* menu shown in Figure 6.9 allows one to have direct access to each of the Visual GNY tabbed panes. The top four items on the pull-down menu give access to the *Assumptions*, *Goals*, *Extensions* and *Analysis* panes respectively. When selecting one of these menu items, the Visual GNY environment appears in the foreground with the respective pane in focus. All of the pull-down menu items are dimmed and disabled if no messages have been defined within the GYPSIE environment. However, all of these menu items, excluding the *View Analysis Results* item, will be enabled if at least one message has been defined.

To export or view the current set of GNY statements, the *Export GNY Statements* and *View GNY Statements* pull-down menu items are employed. Both of these menu items reveal a submenu presenting one with the option of generating LaTeX, English-style text or GYNGER-compatible Prolog output. If an item is selected from the *Export GNY Statements* submenu, then all of the constructed GNY statements and the idealized message passing specification will be output to a file of the user's choice. However, if an item is selected from the *View GNY Statements* submenu, then a dialog is displayed containing the text that would have been exported to file. An example of viewing GNY statements as English-style text is shown in Figure 6.9. An advantage of the export and view menu items is that they allow one to obtain a concise summary of the idealized protocol message passing specification and the GNY statements that have been constructed. If this feature did not exist, then collating the information would require having to tediously cycle through each of the tabbed panes while switching between the radio buttons and combo-boxes that controls the set structured trees being displayed.

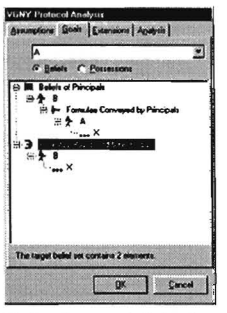
If an analysis has recently taken place, the *View Analysis Results* menu item will be enabled if the results were saved by pressing the *OK* button when closing the Visual GNY environment. If no results are available for viewing the menu item will be dimmed and disabled. The analysis results from the most recent analysis session are also included in the statements that are exported or viewed. However, these results are not included in the output if any of the associated initial assumptions were deleted after the analysis occurred. In a similar vein, the *View Analysis Results* menu item is dimmed and disabled if there are no analysis results to view in the *Analysis* tabbed pane. This could be due to no analysis having been conducted or the results being invalidated due to the deletion of an initial assumption.



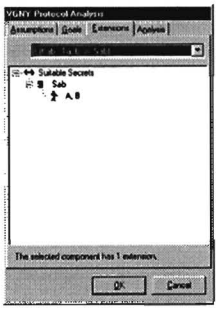
Define the protocol specification in GYPSIE.



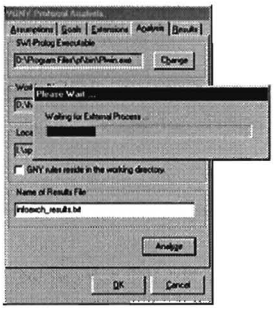
Define the initial belief and possession sets for each principal.



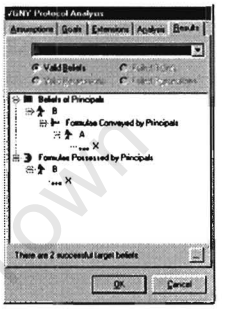
Define the target belief and possession sets for each principal.



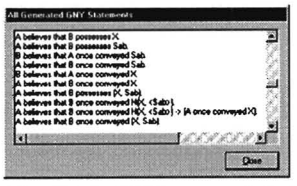
Attach extensions to selected formulae.



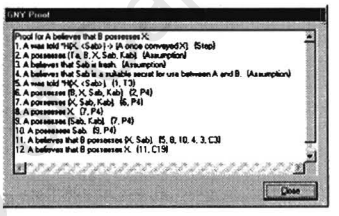
Provide execution details for the GYNGER analyzer and then invoke the analysis.



View valid and failed possession and belief sets for each principal.



View all derived GNY statements in English-style textual form.



View the annotated English-style GNY proof for a successful possession or belief statement.

Figure 6.10: Steps undertaken when conducting a GNY protocol analysis.

6.4 Conducting an Analysis with the Visual GNY Environment

The SPEAR II Framework currently facilitates protocol engineering by making it easier to specify protocol messages, construct the GNY statements related thereto and conduct rigorous GNY-based analyses. However, at this stage the Visual GNY environment does not contain heuristics or artificial intelligence techniques that aid in suggesting goal statements, nor does it offer any form of advice or guidance when a specified goal fails to be achieved. Certain basic heuristics, such as ensuring that an encryption always contains recognizable and fresh formulae, and suggestions for completing specific statements, are not difficult to incorporate within an analysis environment of this nature. However, features that carry out more complex tasks, such as proposing formula extensions or jurisdiction statements, require more advanced implementations and theory. In effect, the responsibility of performing an analysis in a purposeful and targeted manner rests with the protocol engineer — the SPEAR II Framework being a tool which can assist her in this endeavour. In fact, the major contribution of the current SPEAR II Framework to security protocol engineering is that it removes the tedium and syntactical issues associated with protocol design and analysis, making a given analysis session easier to manage, conduct and conclude.

6.4.1 A Typical Analysis Session

In Figure 6.10 we sketch the steps that are undertaken during a typical analysis session. Such a session normally begins by specifying the principals, messages and formulae of the protocol in question within the GYPSIE specification environment. Once this phase has been completed, the Visual GNY environment is invoked and the initial assumptions and goals of each principal are specified as required. Extensions are also appended to formulae. Once all of the necessary preconditions have been defined, details such as the location of the Prolog interpreter, the location of the GNY rules Prolog source, working directories and output files are defined within the *Analysis* tabbed pane. If these details are correctly defined, then the *Analyze* button in the *Analysis* pane is enabled so that it can be pressed to invoked the analysis process. Upon the initiation of the analysis process, the structured GNY trees are all translated into GYNGER-compatible Prolog syntax and run through the analyzer. The Visual GNY environment monitors the analysis thread, and when it is complete, retrieves the results from the output files, parses these results, and then constructs the appropriate structured trees to display in the *Results* pane. Proofs and the list of all derived statements are also stored.

As we can see, a typical analysis session is very visual, with the graphical environment being used as much as possible to aid and guide the user. The *Results* tab is only displayed if an analysis has been conducted, and is automatically hidden if any deletions are made from the GNY preconditions. To view the proof for a successful target goal, one merely needs to right-click on the terminal node of the statement's structured tree representation and then select the *View GNY Proof* menu item. To view all of the GNY statements derived during the most recent analysis, the button in the lower right of the *Analysis* tabbed pane is pressed. All of the constructed GNY statements and analysis results are saved together with the GYPSIE protocol specification. The analysis results are also saved to an output file defined within the *Analysis* tabbed pane. The undo and redo feature within GYPSIE is very useful for protocol analysis, since it allows a user to conduct analyses on variations of the same protocol. For instance, an analysis can be conducted with a certain formula contained within the protocol messages. This formula can then be deleted and another analysis conducted, with the two results being compared at the end. If the first results are better, then the deletion of the formula can be undone. In this way, we can determine whether a given formula is redundant with respect to its effect on helping to achieve protocol goals.

6.4.2 Issues Introduced by Subprotocols

The GYPSIE environment includes the ability to specify subprotocols within a protocol specification. Subprotocols are equivalent to subroutines in a programming sense and are either invoked sequentially by the protocol execution thread or selectively when certain conditions are fulfilled. Now, when generating being-told statements and constructing structured trees in the Visual GNY environment, we exclude references to formulae and messages in embedded subprotocols, as illustrated in Figure 6.11. This approach essentially limits a protocol engineer to analyzing only those messages that are visible within a GYPSIE specification window. An advantage of this limitation is the simplicity that results, as the exchange of derived statements between subprotocols and their parents is not necessary during an analysis. In fact, GYNGER can only deal with initial conditions that are specified at the start of a protocol analysis and no statements can be injected while the analyzer is executing, thus preventing such exchanges from being used at all. This simplistic approach also solves the problem of what to do with the messages that belong to subprotocols which are only executed selectively, since these messages are not included in an analysis. However, even though analyzing terminal subprotocols gives an indication of what each achieves, we would still like to view an entire protocol analysis in perspective and see how the components interact.

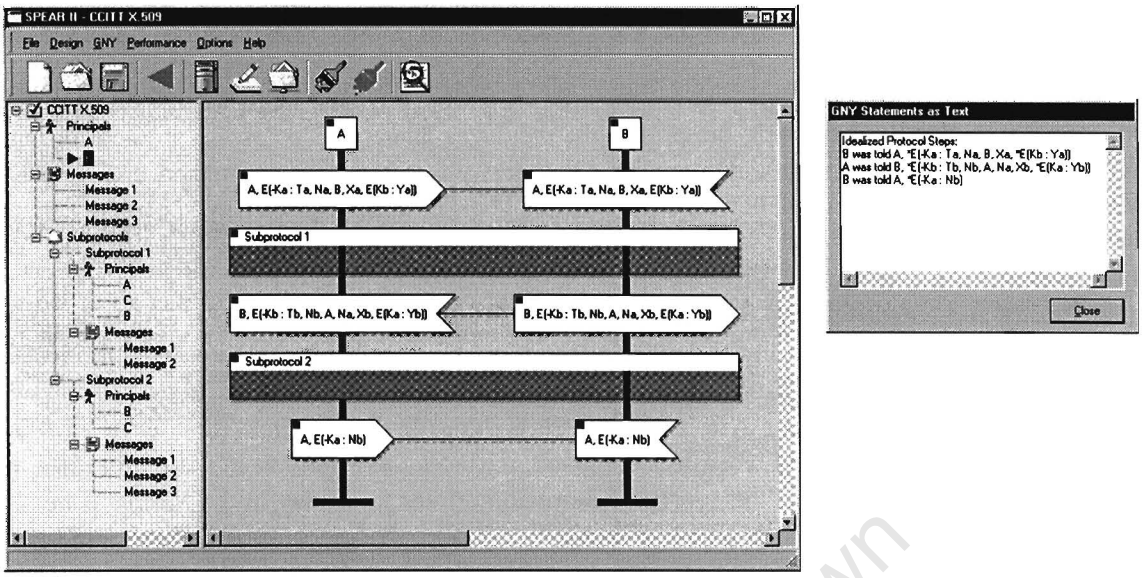


Figure 6.11: A GYPSIE specification including subprotocols and the resultant being-told statements.

The *Flatten Subprotocol Hierarchy* feature, described in detail in Section 3.2.1.6, can be used to extract messages from embedded subprotocols so that all of these messages are present together in the topmost layer of the subprotocol hierarchy. In this way, the entire protocol can be examined by the GYNGER analyzer as there are no nested messages, all subprotocols being deleted after the extraction process. In the case of selectively executed subprotocols, a protocol engineer would have to decide which of these messages to include in the list of messages that constitute the protocol scenario under inspection. Thus, the approach that we have created to analyze a protocol that includes a subprotocol hierarchy requires the extraction of all messages to the same hierarchical layer, after which these messages are structured and selected appropriately to form a viable scenario to be analyzed by GYNGER.

6.5 Experiments with the Visual GNY Environment

In order to examine the suitability of the Visual GNY environment for specifying GNY statements, we decided to conduct a number of user experiments. Within these experiments, we decided to pit the Visual GNY environment against manual, hand-written GNY statement construction. Our major objective was to determine whether Visual GNY facilitates the effective construction of syntactically and semantically correct GNY expressions. We also wanted to gain an understanding of how users who had never been schooled in GNY analysis techniques would be able to use the structured tree approach, as opposed to those who had completed some vestige of a course in security protocol analysis. The subjects involved in the experiments consisted of two groups of users, namely those educated in GNY analysis techniques and those who had never even heard of GNY. Each of the GNY novices had completed a course in network security principles. None of those who had been educated in GNY analysis techniques had carried out a GNY analysis for the last six months. In total, we involved fifteen educated users, and five novice users in the experiments. The number of novice users was rather small. However, this did not present a problem as we were more interested in how the educated users responded to the environment, since they are more representative of those who will use GNY analysis in the workplace or research arena due to their prior exposure. In effect, the novice users were merely tested for comparative reasons.

The experiment that we developed took the form of three tests. The first test required users to translate a given set of English-style GNY statements into both mathematical GNY notation and structured trees. During this test, individuals made use of Visual GNY to construct the trees. The second test required the translation of GNY statements in mathematical-style notation into English-style expressions. Finally, during the third test individuals had to convert structured trees into equivalent English-style GNY statements. In each test, we ensured that every type of GNY statement was exercised. Before testing the novices, we gave each of them a brief introduction to GNY analysis techniques. Both groups of users were also briefly instructed in how to use the Visual GNY interface. The Visual GNY tooltips feature and the ability to view all of the structured trees as English-style text were both disabled.

Prior to conducting the experiments, we realized that certain individuals might not be able to recall or remember the meaning of a large portion of the GNY notation. Such a situation might totally bias the results in favour of Visual GNY. So, we decided to give the test subjects the option of using the Visual GNY pop-up menu captions as a reference for the mathematical-style GNY notation. Since the pop-up menus only provide descriptive text and a mathematical-style GNY icon for each type of GNY statement, the examination of an individual's ability to recall the mathematical-style GNY syntax, construct coherent GNY statements and understand them semantically was not biased by allowing this type of referencing. In fact, when presented with the option of using Visual GNY as a reference, every one of the test subjects accepted, indicating their apprehension regarding the GNY notation. In this respect, we can say that the Visual GNY environment assisted in the construction and interpretation of the hand-written mathematical-style statements to some degree. The experiments test sheet can be found in Appendix D, while the results for each of the tests appears in Table 6.2.

	15 Educated Users		**5 Novice Users**	
Task	**Sample Mean**	**95% Confidence Interval for Population Mean**	**Sample Mean**	**95% Confidence Interval for Population Mean**
English to GNY	78.46%	(76.91%, 80.02%)	72.31%	(71.47%, 73.15%)
English to Visual GNY	98.46%	(98.15%, 98.77%)	100.00%	(100.00%, 100.00%)
GNY to English	87.22%	(86.36%, 88.08%)	85.00%	(84.75%, 85.25%)
Visual GNY to English	87.78%	(86.82%, 88.73%)	91.67%	(91.28%, 92.06%)

Table 6.2: Results of tests pertaining to the accuracy of GNY statement construction.

Listed within Table 6.2 are the average scores that were obtained by users from each group for the respective tests. From the data obtained, we also calculated the 95% confidence interval indicating where the population mean should lie. This computation assumes that the set of users are a representative sample of our envisaged user base. Since the sample size for our set of novice users consisted of only five individuals, the confidence intervals obtained are not as accurate as those of the educated users, which had a larger sample of fifteen individuals. The following is a sample of what some of the test subjects had to say about the Visual GNY environment:

> "[Visual GNY] makes understanding GNY straight-forward. However, the hierarchical structure may be confusing for some people. On the whole, it seems beneficial with a short learning curve."
>
> "I felt that constructing GNY statements was much easier using Visual GNY. There was not the overhead of having to remember the plethora of GNY symbols. I think Visual GNY's power definitely lies in the ease with which you can rapidly and easily construct statements."

> "Visual GNY really accelerates the learning of GNY expressions and syntax."
>
> "As a tool to introduce the layout of GNY expressions, and to allow for quick construction of these statements in a visual environment, [Visual GNY] looks very effective."

An important conclusion that we can derive from the first two tests is that Visual GNY effectively helps users to construct GNY statements. All of the statements that were specified with Visual GNY turned out to be syntactically correct. Those individuals who did not score perfect results for the English to Visual GNY test committed semantic errors, specifically the use of incorrect formulae within expressions. The reason why users always constructed syntactically correct structured trees was because the Visual GNY environment did not allow them to exit or change tabs unless all of the trees were complete. The fact that the novice users all got 100% of the Visual GNY statements correct and the educated users only got 98% correct is not very significant. It merely indicates that the novice users concentrated more closely on the formulae which they inserted into the structured tree. With a larger sample of novice users, we would have definitely encountered someone who would have made a substitution error. A noteworthy conclusion that we can draw is that using Visual GNY produces significantly better results than specifying GNY statements by hand. When using Visual GNY individuals scored almost 20% higher. Essentially, what the Visual GNY environment has done is to totally remove the syntactical and notational issues associated with the construction of GNY statements, thus allowing individuals to concentrate on the actual protocol analysis process, which is far more fundamental and important.

The final two tests revealed some interesting results. The scores for reading off mathematical and structured tree-style GNY statements were almost identical for the educated users, and not significantly different in the case of novice users. This seems to indicate that the structured tree representation of a given GNY statement is not any more readable than its corresponding mathematical-style rendering. Difficulties encountered when reading from a structured tree can be attributed to having to jump from node to node, and sometimes having to skip nodes and only return to them later. The fact that mathematical-style GNY is primarily structured in a linear fashion means that it is not that difficult to interpret once the symbols have been understood. Since the novice users had never used GNY before, they did not have any preconceived notions as to how it should be written or structured. This fact might offer a possible explanation as to why they scored better than the educated users when reading from the tree. An interesting point to note is that the test subjects found it easier to read mathematical-style GNY statements than to construct them. This could be because construction of these statements requires recalling the function of each symbol and then stringing these symbols together correctly, while writing out the meaning of mathematical-style GNY statements merely requires one to have an idea of what each symbol represents.

During the course of this brief experimental analysis, we have noticed that many individuals struggle to recall the mathematical-style GNY notation if they have not been using it for some time. As a result of this fact, individuals will not immediately make use of GNY to analyze protocols, since their notational ineptitude would serve as a hindrance to the specification of assumptions and goals. Because of this issue, we chose to develop Visual GNY, empowering those who have used GNY in the past to use it again with ease. The experiments which we have carried out have confirmed that the construction of GNY statements in the Visual GNY environment is a straight-forward and painless operation, producing high-quality syntactically correct statements. However, reading GNY statements from the structured tree is not necessarily a simple task, sometimes confusing individuals. For this reason, the addition of the tooltips and *View as Text* features are exceptionally useful, since they help to create a system which virtually ensures that users construct error free statements — the pop-up menus accelerating and aiding the construction process, and the tooltips and *View as Text* features being used to validate, verify and view the constructed statements.

6.6 Implementation Details

An object-orientated approach was employed when creating the Visual GNY environment in order to facilitate expansion and understanding of the source code. The environment itself was prototyped and written using the Borland C++ Builder package which significantly aids in the creation of event-driven Windows applications. The Visual GNY source code interacts very closely with the GYPSIE API due to the fact that information about the protocol and message formulae can only be obtained in this way. Interaction with the GYNGER analyzer takes place primarily through system calls which invoke external processes, in this case a Prolog analyzer.

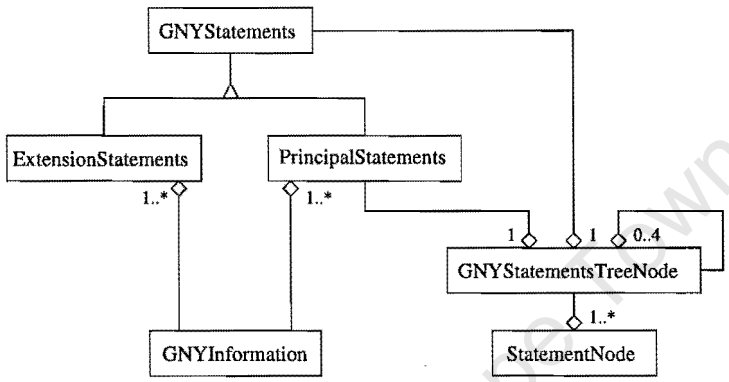


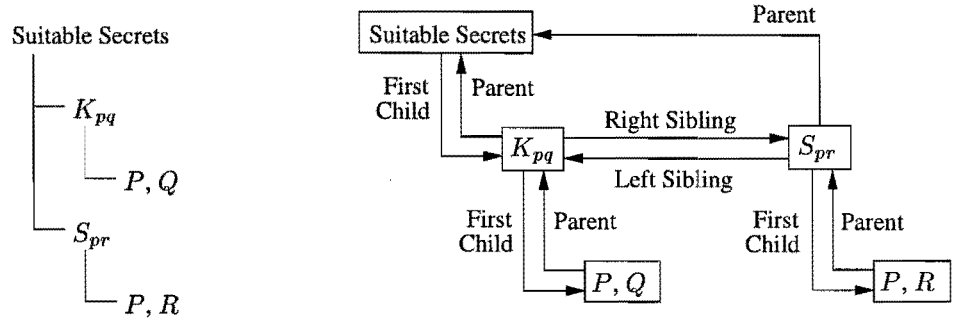
Figure 6.12: Diagram of the classes used in the Visual GNY implementation.

6.6.1 Class Hierarchy

The Visual GNY class hierarchy consists of six classes and is illustrated in Figure 6.12. The `GNYStatements`, `PrincipalStatements` and `ExtensionStatements` classes are used to store GNY statements, while the `GNYStatementsTreeNode` and `StatementNode` classes are used to represent a structured GNY tree. The `GNYInformation` class contains all of the GNY statements applicable to a given protocol.

6.6.1.1 Representing Structured Trees

The `StatementNode` class stores the information that is contained within a structured tree node. This information is either a list of formula references, a GNY statement category or a list of principal names. Only one of these three items is contained in a given `StatementNode` at any one time. The `GNYStatementsTreeNode` provides a mechanism for representing the structure of a GNY statement tree through the use of a sibling-child tree approach (see Figure 6.13). Pointers to other `GNYStatementsTreeNode` objects are stored within a given `GNYStatementsTreeNode` to reference a node's parent, left sibling, right sibling and first child. An object of type `StatementNode` encapsulated within a `GNYStatementsTreeNode` stores the node information. Methods within the `GNYStatementsTreeNode` allow one to write to and initialize from a graphical tree-view component. Obtaining the equivalent English, LaTeX or Prolog-style GNY statements from a structured tree can be accomplished with a simple method call. Tree manipulation, saving, loading and search routines are also provided within the `GNYStatementsTreeNode` class definition.



Standard Tree Representation Sibling–Child Representation

Figure 6.13: Two representations of a structured GNY tree.

6.6.1.2 Storing GNY Statements

The `GNYStatements` class is used to store GNY statements without a believes or possesses prefix. An encapsulated pointer to a structured tree is used to reference the tree storing the GNY statements. The `ExtensionStatements` class inherits from `GNYStatements`. It contains a reference to a formula so that the component to which GNY extension statements are attached can be stored. To store statements with a believes or possesses prefix, the `PrincipalStatements` class is used. This class also inherits from `GNYStatements`. Two pointers to structured trees are stored within the `PrincipalStatements` class. The first is inherited from `GNYStatements` and used to store principal beliefs, while the second has been added to the class and is used to record possession statements. Besides the pointers to these two trees, the name of the principal who is the subject of the stored statements is also recorded so that the believes and possesses prefixes can be correctly constructed. The ability to save, load and access stored GNY statements is provided by appropriate methods within each class.

6.6.1.3 Storing GNY Information

The `GNYInformation` class is used to store all of the belief, possession and extension statements applicable to a given protocol. Three linked lists within the class named `principalPreconditionsList`, `principalGoalsList` and `extensionsList` are used to store initial assumptions, target goals and extensions respectively. Each element of `principalPreconditionsList` and `principalGoalsList` is of type `PrincipalStatements` and contains a principal's initial assumptions and target goals respectively. Similarly, every element of `extensionsList` is of type `ExtensionStatements` and contains the statements attached to a formula. Another two linked lists named `principalValidGoalsList` and `principalFailedGoalsList` are used to store successful and failed goal statements respectively. The elements within these lists are of type `PrincipalStatements`. Other information related to an analysis, such as the name of the results file containing GYNGER output and all generated GNY statements are also stored in this class. Methods that allow one to obtain the structured trees containing a given principal's initial assumptions, target goals, failed goals or valid goals, as well as a formula's extensions, exist within the class definition. The GNY statements contained within the protocol can also be saved, loaded and exported to an appropriate English-style text, LaTeX or Prolog format.

6.6.2 Saving and Loading Structured Trees

All GNY statements created in the Visual GNY environment, as well as any analysis results, are saved along with the GYPSIE specification of a protocol as ASCII text. When loading a GYPSIE specification, the GNY portion of the source file is parsed, resulting in the Visual GNY environment and all of its associated data structures being initialized. Within each of the Visual GNY classes, a constructor exists that will read a subset of the saved specification from a file and then create an appropriate object. Each class also contains methods that allow it to be written to a file.

In the text fragment that follows we illustrate how a structured tree is stored as ASCII text. The code on the left represents the structured tree containing the freshness statements $\sharp(T_b)$ and $\sharp(N_a)$, while the code on the right represents the structured tree containing the shared secret suitability statement $Q \overset{S_i}{\longleftrightarrow} P_i$:

```
GNY_NODE {
    CAPTION = "Fresh Components"
    TYPE = "gtFreshComponents"
    EXPANDED = 1
    GNY_NODE {
        CAPTION = "Tb"
        TYPE = "gtComponent"
        EXPANDED = 0
        PAYLOADS {
            19123592
        }
    }
    GNY_NODE {
        CAPTION = "Na"
        TYPE = "gtComponent"
        EXPANDED = 0
        PAYLOADS {
            13143522
        }
    }
}
```

```
GNY_NODE {
    CAPTION = "Suitable Secrets"
    TYPE = "gtSuitableSecrets"
    EXPANDED = 1
    GNY_NODE {
        CAPTION = "Si"
        TYPE = "gtSuitableSecretComponent"
        EXPANDED = 1
        PAYLOADS {
            14128813
        }
        GNY_NODE {
            CAPTION = "Q, Pi"
            TYPE = "gtSharingPrincipals"
            EXPANDED = 0
            Principal_1 = "Q"
            Principal_2 = "Pi"
        }
}
```

Common to each GNY_NODE are the caption, type and expansion status fields. A 'zero' value for the expansion status indicates that a node is collapsed, while a 'one' indicates that it is expanded. The remaining fields depend on the node's type. When reconstructing the structured tree from saved information, adjacent GNY_NODEs are interpreted as siblings, while nested GNY_NODEs are viewed as children of the container. The numeric codes used within the PAYLOADS section are used to reference formulae defined in GYPSIE. The codes are written to file by GYPSIE prior to saving any Visual GNY information. Obtaining the information applicable to a given formula merely involves performing a look-up on its numeric code in the saved source file.

To save initial assumptions, target goals, successful goals and failed goals, the appropriate structured trees and principal to whom the statements therein apply are written to file. The belief and possession statements are saved separately. When saving successful goals, the GNY_NODE for each terminal node in the corresponding set of structured trees contains a proof in English-style GNY as one of its fields. When saving extensions, the relevant GNY statements as well as the formula to which they are attached is saved. Along with the GNY statements defined in the visual environment, analysis information such as the name of the results file containing GYNGER output and the list of all derived statements is also saved.

In the text fragment that follows we illustrate how GNY statement collections and extensions are saved. The code on the left represents the initial assumptions for principal P, the code in the middle describes the successful goals of principal S, while the code on the right represents the extensions attached to the component with the numeric GYPSIE code of 19135980. In the interests of brevity, we have omitted the information contained in the GNY_NODE structures:

```
PRECONDITIONS {
    PRINCIPAL = "P"
    BELIEFS {
        GNY_NODE {
            ⋮
        }
        ⋮
    }
    POSSESSIONS {
        GNY_NODE {
            ⋮
        }
        ⋮
    }
}
```

```
VALID_GOALS {
    PRINCIPAL = "S"
    BELIEFS {
        GNY_NODE {
            ⋮
        }
        ⋮
    }
    POSSESSIONS {
        GNY_NODE {
            ⋮
        }
        ⋮
    }
}
```

```
EXTENSIONS (19135980) {
    GNY_NODE {
        ⋮
    }
    ⋮
}
```

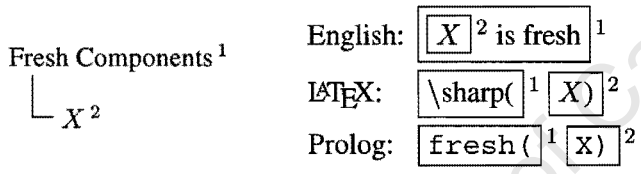
If no beliefs or possessions exist for a principal in a given collection, then the corresponding BELIEFS or POSSESSIONS section is not present. If a given principal has no beliefs and possessions for one of the statement collections, then it will not have a record in that collection. For example, if principal R does not have any target beliefs and possessions, then there is no entry in the `principalGoalsList` of the encapsulating `GNYInformation` object, and hence when saving all GNY statements to file there will be no GOALS record for R.

6.6.3 Exporting Structured Trees

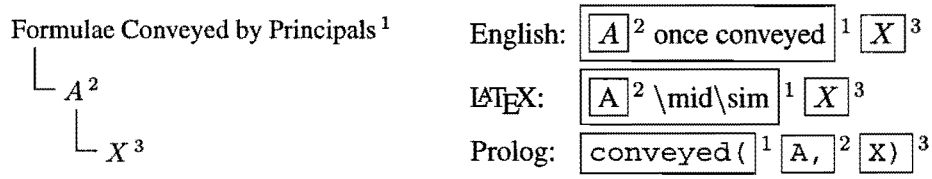
A set of recursive methods belonging to the `GNYStatementsTreeNode` class is used to export structured trees to English, LaTeX or Prolog. These methods effectively navigate a path in a structured tree from the tree's root to the terminal node completing a GNY statement. During this journey the GNY statement to be exported is progressively constructed as each node along the path is visited. Assume that `exportStructuredTree()` is one of these recursive methods. When invoking the `exportStructuredTree()` method of a structured tree node the resultant execution thread first weaves its way to the root of the structured tree by invoking the `exportStructuredTree()` method of each node's parent. Upon reaching the root, these recursive calls terminate and the process of unravelling them commences. Thus, each node along the original execution path is effectively visited twice by the execution thread. The first visit occurs when invoking the `exportStructuredTree()` method of a node's parent, and the second when this method has completed execution. During this second visit the exported statement is updated by appending or inserting text based on the node's type. The code fragment which follows illustrates how the recursive methods which export structured trees appear. Note that three such methods exist — one to export to English text, one to create LaTeX statements and one to create GYNGER-compatible Prolog.

```
AnsiString GNYStatementsTreeNode::exportStructuredTree() {
    AnsiString exportedText;
    GNYTypes nodeType = statementNode->getType();
    if (parent != NULL) exportedText = parent->exportStructuredTree();
    switch(nodeType) {
        ⋮
    }
    return exportedText;
}
```

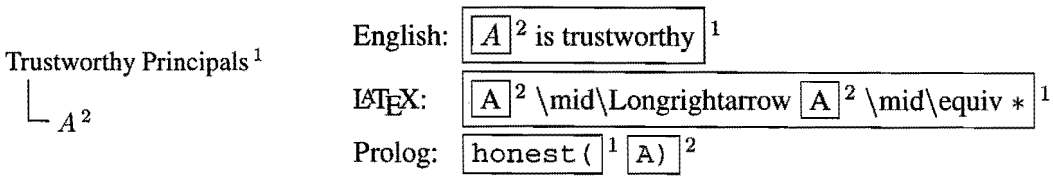
Within the `switch` statement of the above code fragment, text is appended to or inserted into the `exportedText` variable depending on the value of `nodeType`. Recall that in Section 6.2.3 we divided all twelve possible GNY statements into seven groups. In the diagrams that follow, we illustrate how the statements in each of these groups are converted into English text, LaTeX or Prolog statements from a given structured tree. On the left side of each diagram we show a structured tree with each of the nodes labelled sequentially, starting at the root. On the right side we illustrate the text that is appended or inserted when the execution thread passes through a given node during the construction process. Let's begin by considering a freshness statement from Group 1:



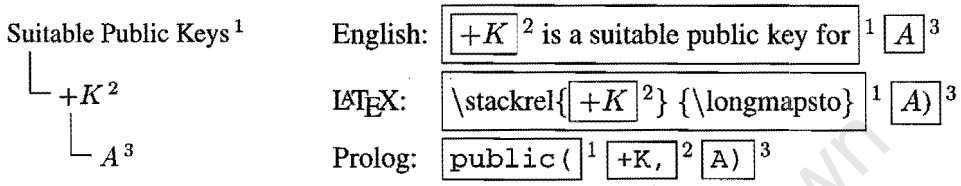
When the execution thread passes through node (1) the output text is initialized with the phrase '\sharp(' or '`fresh(`' for LaTeX or Prolog output respectively. Upon entering node (2), the appropriate textual representation of the fresh formula is appended to the text being constructed — in this case 'X)' for LaTeX output, or '`X)`' for Prolog output. The situation differs with English-style output because the individual components of the English statement are not structured in the same order as their counterparts in the tree. When passing through node (1), the text '$ is fresh' is created and passed on to the next node. Then when entering node (2) the '$' placeholder character is replaced with the textual representation of the correct formula, which in this case is X. We now consider a conveyance statement from Group 2:



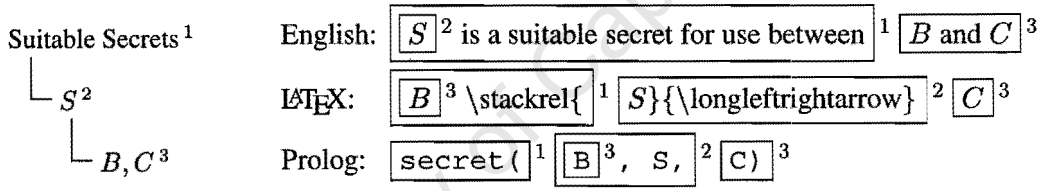
When constructing the conveyance statements in English or LaTeX, we first output the statement with a placeholder for the principal name when evaluating node (1). When passing through node (2), this placeholder is replaced with the actual principal name. The evaluation of node (3) merely requires appending the appropriate textual representation of X to the statement being constructed. When generating Prolog output, all of the respective components making up the statement follow sequentially and thus the appropriate Prolog text merely needs to be appended to the current output when passing through each node. We now examine the trustworthiness statement from Group 3:



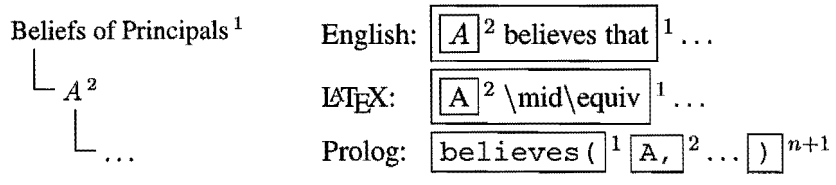
The techniques that we have seen in the past two explanations are again applied to to the generation of trustworthiness statements. However, when generating the corresponding LaTeX output *two* placeholders are inserted into the text output from node (1). As a result, when entering node (2), two substitutions have to be carried out. We now present the suitable public key statement from Group 5:



Generating the English, LaTeX and Prolog forms of the public key suitability statement from a structured tree is carried out using the same principles that were described for the statements in Group 2. We now examine how a shared secret suitability statement from Group 6 is constructed:

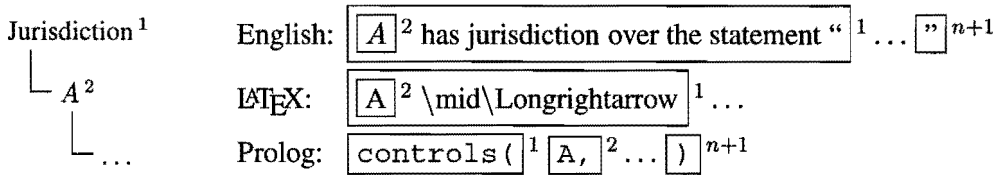


The terminal node in a shared secret suitability tree contains the names of the *two* principals who share the secret. In both the LaTeX and Prolog representations of this statement these two names are separated, thus making substitutions imperative when exporting the statement from a tree. As can be seen in the explanatory diagram above, the substitution of the first principal name (B is this case) takes place when visiting node (3), while the remaining principal name is appended to the end of the statement. When generating English-style output, the two principal names are appended to the statement when the execution thread passes through node (3). We now examine how a believes statement belonging to Group 4 is constructed:



A believes statement essentially consists of a believes prefix concatenated with a GNY statement. When exporting a structured tree containing such a statement, the appropriate believes prefix is generated and then concatenated with the text that results from exporting the remainder of the tree. When generating English and LaTeX GNY statements this technique works correctly. However, when generating Prolog output the closing bracket that terminates the clause must be included. This is done by keeping count of the number of closing brackets required through the use of a counter which is incremented each time

node (1) is encountered. The appropriate number of closing braces are then appended once the terminal node of the believes statement has been visited and processed. Finally, we examine how the jurisdiction statement from Group 7 is exported:



A structured tree representing a jurisdiction statement is exported in the same way as one representing a believes tree. However, because quotes surround the statement over which a principal has jurisdiction, a counter is used to keep track of how many quotes to append at the end of the statement. Once the nested statement is complete, the appropriate number of quotes are appended to the end of the statement.

6.6.4 Conducting an Analysis with GYNGER

Since the executable code for the GYNGER analyzer is not embedded within the SPEAR II application, calls to the analyzer take place by creating, executing and monitoring an external Prolog process. Input for the GYNGER analyzer is generated from structured trees by the Visual GNY environment and then output to file. When GYNGER is finished with the analysis process, the results are dumped to a file which is then parsed by the Visual GNY system. These results are converted into structured trees and displayed in the *Results* pane. The following is a high-level overview of how we have coded this analysis process:

1. *Export structured trees to executable GYNGER-compatible Prolog:* Each initial assumption and target goal is assigned a unique numerical identifier. This identifier ensures that the analyzer output for each target goal can be matched with the appropriate structured tree representation. In this way we can determine which goals were successful and which failed, displaying the appropriate structured trees in the Visual GNY *Results* pane. The structured trees are exported in the following order: initial possession trees, initial belief trees, target possession trees and target belief trees. Within each of these exported collections, the individual statements are sorted by principal, the order of the principals being determined by where the principal appears in the `principalPreconditionsList` and `principalGoalsList` linked lists. Once the structured trees have been exported to Prolog syntax, the resulting text is extended by attaching the `go/0` predicate described in Section 5.2.5. This generated Prolog output is then dumped to file.

2. *Execute GYNGER and monitor the analysis process:* Once the Prolog output has been generated and dumped to file, the GYNGER analyzer can be invoked by executing the SWI-Prolog interpreter with the command-line parameters described in Section 5.2.5. The Windows `CreateProcess()` API call is used to start the interpreter, all of the command-line parameters being passed as an argument to `CreateProcess()`. The `CreateProcess()` function call returns a handle to the executing process which it spawns. At this point a modal dialog box that monitors the execution handle using the `GetExitCodeProcess()` API call is displayed. The dialog box contains a progress bar which is continuously incremented by a background thread. Once the execution of the Prolog process is complete, the modal dialog is closed and the Visual GNY system is then free to examine the analysis results that GYNGER dumped to file.

3. *Parse and process the results of the GNY analysis:* Once GYNGER has completed an analysis, the results are dumped to file and then read in by the Visual GNY system. The format of the analysis results returned by GYNGER is described in Section 5.2.5. A proof is generated for each successful goal statement. If the goal failed, then the text 'FAILED!' appears instead of a proof. The parser that we have constructed reads this GYNGER output and then for each goal statement converts the corresponding proof into a linked list of strings, which is in turn placed into a linked list of goal results. If the goal failed, then the text 'FAILED!' is inserted as the proof text in the linked list of goal results. After this parsing process is complete, the linked lists of strings associated with a failed goal are each replaced by a `NULL` value. Thus, we are left with a linked list of proofs for each specified goal, a proof being `NULL` if the goal failed, and containing a linked list of strings otherwise. The index of the proof in the linked list corresponds to the numerical identifier that the corresponding goal statement was assigned during the export phase of the analysis. All of the derived statements are also extracted from the GYNGER output file and stored.

4. *Display the analysis results in the Visual GNY environment:* Once the analysis results returned from GYNGER have been parsed, recorded and appropriately structured, we can initialize the *Results* pane with this information. We will now briefly describe how the structured trees which store these analysis results are constructed. The steps performed to initialize the trees containing the successful and failed possession goals are listed below:

   for $i = 1$ **to** (number of principal goal records stored in `principalGoalsList`) {
   Extract the possession statements in `principalGoalsList`[i].
   Create two structured trees named `validTree` and `failedTree`.
   Initialize `validTree` and `failedTree` with the extracted possession statements.
   for $j = 1$ **to** (number of possession statements in `principalGoalsList`[i]) {
   if (proof for possession goal j is `NULL`) {
   Remove possession goal j from `validTree`.
   } **else if** (proof for possession goal j exists) {
   Remove possession goal j from `failedTree`.
   Add proof to terminal node of possession goal j in `validTree`.
   }
   Save the `validTree` and `failedTree` structured trees.
   }
   Use the appropriate methods to initialize the *Results* tree-views and combo-box.

   In a similar way, we initialize the trees that store the successful and failed target beliefs. Once all of the results trees have been initialized and saved in the Visual GNY data structures, the linked list of proofs created in step (3) is discarded as the proof for each goal is stored in the terminal node associated with the structured tree representation of a given goal.

Once the structured trees are displayed in the *Results* pane tree-views, the associated proofs can be viewed by right-clicking on the terminal node of a successful goal and then selecting the *View GNY Proof* option. If a failed goal is right-clicked, no pop-up menu appears. The proof for a successful goal that is stored within the `StatementNode` object associated with the terminal node is extracted by calling the `getProofStringList()` method. All of the derived statements are stored within the associated `GNYInformation` object and are extracted by using the `getAllGeneratedGNYStatements()` method. After a successful analysis has been conducted, the results are saved if the user presses the *OK* button, and discarded if *Cancel* is pressed.

6.6.5 Interaction with GYPSIE

The most obvious interaction that takes place between the GYPSIE and Visual GNY environments is the exchange of formulae and principal names that are involved in a protocol. When inserting formulae into the Visual GNY pop-up menus, the `getPayloadsUsedInMessages()` method belonging to the `DesignController` class is employed. This method returns a linked list containing references to each of the formulae defined within a protocol's message passing specification. This list of formulae is also used to initialize the *Extensions* pane combo-box. To obtain individual principal names, the `getPrincipalName()` method belonging to the `DesignController` class is used to iterate through all of the principal names specified in the protocol so that the relevant Visual GNY combo-boxes can be initialized. An important point to note is that no principal names or formulae defined within nested subprotocols are exchanged between GYPSIE and Visual GNY.

Generating the text, LaTeX and GYNGER-compatible Prolog versions of the being-told GNY statements associated with a given protocol is the responsibility of the GYPSIE environment. To determine whether a formula within a given being-told statement is eligible for a star, the `payloadHasPrefixedStar()` method belonging to the `DesignController` class is used. This method takes the objects representing a formula and the encapsulating message as parameters, returning a boolean value of *true* if a star is allowed, and *false* otherwise. To generate being-told statements for a given message, four methods belonging to the `Message` class are used to export the message contents. The `getPrologWithExtensions()` method is used to generate Prolog-style being-told statements. This method does not attach stars to formulae, but it does include their extensions in the output. The methods `getTextWithExtensionsAndStars()`, `getLatexWithExtensionsAndStars()` and `getPrologWithExtensionsAndStars()` all generate their respective output with stars and extensions. Each of the four preceding methods parses the component tree stored within a message object, using the `payloadHasPrefixedStar()` method to determine whether a given formula is eligible for a star. Thereafter, the appropriate being-told prefix is added to the output.

To determine whether a formula used in a structured tree still exists within the GYPSIE specification, the `DesignController` method `componentIDExistsAmongMessages()` is used. If the formula exists, a boolean value of *true* is returned, otherwise a value of *false* results. A question mark icon is displayed in a structured tree if the associated formula no longer exists in the protocol specification to which the GNY statement tree is related. Methods within the GYPSIE `DesignController` class are also employed to populate the *Suggestions* pop-up menu items. To obtain the list of formulae which are initially possessed by a principal, the `getPossibleInitialPossessions()` method is used. This method takes a principal name as a parameter, and returns a linked list of formula references. Similarly, to determine the nonces a principal has originated and the timestamps he has received, we can use the `getNoncesOriginatedByPrincipal()` and `getTimestampsReceivedByPrincipal()` methods respectively.

The GYPSIE environment makes use of embedded `GNYInformation` objects to store the GNY statements related to a given protocol. Every `DesignController` contains a `GNYInformation` object that holds the GNY statements applicable to the protocol specification which it manages. When the Visual GNY environment is launched, a backup copy of this `GNYInformation` object is stored alongside the data structures associated with the Visual GNY dialog. Thereafter, any changes made to the structured GNY trees only result in the `GNYInformation` object embedded within the `DesignController` container being updated. The GNY statements stored within the saved `GNYInformation` object are copied back into the the `GNYInformation` object contained within the `DesignController` only if the *Cancel* button is pressed when exiting from the Visual GNY environment.

When executing the SPEAR II application, the user preferences and analysis settings are read from an initialization file named *spear2.ini*. These settings are stored in a preferences object and every `DesignController` in the system has a pointer attribute that references these preferences. The location of the SWI-Prolog executable, working directory and GYNGER source code are all retrieved from this preferences object and used to initialize the `GNYInformation` object contained within the `DesignController`. Whenever the settings are changed in the Visual GNY *Analysis* pane, they are resaved if the *OK* button is pressed when exiting from the Visual GNY environment.

6.7 Closing Remarks

In this chapter we have described a method for graphically representing all possible GNY statements. The approach which we have devised represents a given GNY statement as a collection of successive nodes in a tree-like structure, each node being a child of its predecessor. All statements of the same type form part of the same tree, a heterogeneous set of GNY statements being represented by a collection of separate trees. Each node within a GNY tree has an assigned type and its position within the tree is determined by that type. Hence we use the term 'structured tree' to refer to this tree-like representation, since the tree nodes are ordered according to a predefined pattern in order to produce a meaningful representation of GNY information. The Visual GNY environment is the interface which a protocol engineer uses to conduct a GNY analysis within the SPEAR II Framework. Collections of structured trees are used within this environment to represent initial assumptions, extensions, goals and analysis results. To view and modify a set of structured trees, a number of common GUI components such as tree-views, tabbed panes and pop-up menus are used. When an analysis needs to be conducted, the GYNGER analyzer is invoked by the Visual GNY system with the GNY statements constructed therein being used as input. Once the analysis is complete, the results are displayed in the Visual GNY environment using structured trees and English-style GNY syntax.

The most significant contribution of the Visual GNY environment to security protocol engineering is that it removes the tedium and syntactical issues associated with GNY-based analysis, making a given analysis session easier to manage, conduct and conclude. Within the confines of the Visual GNY environment, it is impossible to specify a syntactically incorrect GNY statement as the structured trees are constructed using dynamic pop-up menus which limit a user's choice regarding the set of nodes which can be appended or attached to an existing node. The set of formulae and principal names which can be used in a GNY statement are extracted from the GYPSIE specification of a protocol so that no irrelevant information is used. However, we cannot guarantee that statements which use the correct formulae or principal names will always be constructed. In effect, it is still the responsibility of a protocol engineer to intelligently and purposefully carry out a GNY analysis. Another major benefit of the Visual GNY environment is that it appears simple, clean and concise and does not intimidate users with any apparent complexity. During our experiments which focused on the structured tree dimension of the environment, we found that users were able to construct a set of GNY statements with 98% accuracy using when the Visual GNY environment. When asked to construct a similar set of statements using mathematical-style GNY, their accuracy rate was only 78%.

To manage and organize an analysis, the Visual GNY environment makes use of tabbed panes to group related sets of GNY statements. Within each of these panes, the set of structured trees applicable to a given principal or formula is selected by using a combination of radio buttons and combo-boxes. Within each tree-view that contains the structured trees, statements are grouped according to their type. Nodes within a structured tree representation can be expanded or collapsed to vary the level of detail. Expanding

a node reveals its direct descendants, while collapsing a node hides all of its descendants. An important issue that we noticed during our experimentation was that users did not always find it easy to read a statement represented as a structured tree. In fact, most users felt that reading a linear statement in English or Mathematical-style GNY was a lot easier. Thus, in order to facilitate reading structured tree-style GNY statements, we implemented a feature that reveals the GNY statement represented by a collection of nodes through the use of tooltips — when a user's mouse pointer hovers over a terminal node in a structured tree, English-style text describing the statement to which the node belongs is displayed. In this way we have the best of both worlds, with the structured trees aiding in efficient statement construction, and the tooltips helping to read these constructed statements.

A limitation of the current Visual GNY interface is that it does not present a protocol engineer with a central, easily accessible view of all the GNY statements that have been constructed. Instead, GNY statements are rigidly organized according to their type and the principal or formula with which they are associated. This approach allows one to quickly locate a given statement or find and modify the set of statements applicable to a given principal or formula. However, obtaining an overall impression of what has been specified is not that straight-forward. For example, determining which formulae have extensions involves having to cycle through all of the formulae listed in the *Extensions* tabbed pane of the Visual GNY environment. So, to address this situation we have added a feature to the SPEAR II Framework which displays all of the constructed GNY statements in English-style GNY text within a dialog box window. These statements are displayed in the following order: parsed protocol specification including extensions, initial beliefs and possessions, target beliefs and possessions, and analysis results showing the failed and successful beliefs and possessions. Besides allowing users to view all the GNY statements in English text, statements can also be exported to file in two more additional formats, namely LaTeX and GYNGER-compatible Prolog, using commands accessible from the SPEAR II application.

When conducting experiments that involved users and the Visual GNY environment, our main concern was to examine the feasibility of the structured tree approach for representing GNY statements. The experiments that we conducted pitted hand-written GNY statement construction against construction in the Visual GNY environment. In many respects it was clear that the structured tree approach would dominate, however we were interested in how much better it would be and how easily individuals would adapt to it. Our findings showed that the structured tree approach aids significantly in statement construction due to the rigorous syntactic and partial semantic guidance that it offers through pop-up menus. In fact, many of the people who used the system were particularly enthusiastic about it and the students that we employed were particularly keen to use it for future assignments. To examine the functionality of the SPEAR II environment, we used it within four fully fledged protocol analyses. The protocols we used are the same that were used when testing GYNGER in Section 5.3. For each protocol the messages were defined in GYPSIE and the related GNY statements in Visual GNY. The GYNGER analyzer was invoked by the Visual GNY environment and the appropriate goals were all found to have succeeded, a correct proof being generated for each. These results were saved along with the GYPSIE protocol specification.

Visual GNY in combination with the SPEAR II Framework can play a significant enabling role within the context of security protocol engineering and analysis. By minimizing the effort that protocol engineers have to expend on the syntactic and repetitive portions of an analysis, the SPEAR II Framework frees these engineers to concentrate more on the essential portions an analysis. Through the course of this chapter we have tried to show that the Visual GNY environment is a more than viable system to use when conducting GNY-based protocol analysis. By effectively partitioning GNY statements and aiding in their construction, the system helps to produce an engineering environment that is more robust and less daunting to use than others. All of the textual logic-based analysis environments that we have examined entail the user having to learn the specification syntax of the system [14, 54, 27, 61, 47], while the

graphical techniques that we examined in Section 6.2.2 have weaknesses that limit their expressiveness or usability. In comparison, the Visual GNY environment provides an interface that is a clean, concise, consistent and uncluttered. However, it is still expressive, easy to learn and able to represent all GNY statements. Based on the comments from users and our experimental results, we feel that the Visual GNY approach to protocol analysis is headed in the right direction and is a step forwards towards implementing more wide-scale and usable protocol analysis and modelling tools.

Chapter 7

Conclusion

"Most of the application of formal methods to cryptographic protocols has concentrated on applying formal methods to existing protocols. However, it would be cheaper and more effective to use formal methods in the design of a protocol, and so save on the expense of redesign."

— *Catherine Meadows, "Formal Verification of Cryptographic Protocols: A Survey"*

Protocol engineering is not an easy task and the development of tools and techniques that make it less burdensome should be encouraged. A multi-dimensional approach to protocol engineering helps to increase the confidence we place in protocols, since it achieves a higher level of assurance than individual engineering techniques can attain when used in isolation. The SPEAR II application that we have developed implements a multi-dimensional approach to security protocol engineering by incorporating modelling and analysis functionality in a single application. Besides the current set of implemented features, the SPEAR II project is ongoing and extensible, and we envisage the implementation of a number of additional engineering dimensions, such as code generation and attack analysis.

In this dissertation, we have refined the original SPEAR I Multi-Dimensional Protocol Engineering Framework, resulting in the development and partial implementation of the SPEAR II Framework. As it currently stands, the SPEAR II application, which realizes the SPEAR II Framework, contains a protocol modelling environment (GYPSIE), a GNY analysis environment which helps to collate and construct GNY statements (Visual GNY), and a Prolog-based GNY analyzer (GYNGER). A rounds calculator that determines both synchronous and optimal rounds has also been included to aid in simple performance analysis. To accommodate the addition of further protocol engineering dimensions and techniques, SPEAR II has been written in a modular fashion so that expansion can take place by embedding the source code for the additional modules directly within SPEAR II, or by manipulating the execution of an external application using system calls and then retrieving the results. In both cases, the SPEAR II user interface will have to be upgraded to facilitate interaction with the new engineering modules.

To a large extent, the majority of this dissertation has focused on the facilitation of GNY-based protocol analysis. We have aimed to make GNY analysis accessible to a wider range of individuals by freeing them to focus more on the semantic issues related to analysis, instead of getting bogged down in the associated syntax. Through the creation of GYPSIE, GYNGER and Visual GNY, we have produced an application that assists in the entire GNY analysis process, aiding in the specification of a protocol, construction of its associated GNY preconditions and goals, application of the GNY inference rules and the retrieval and presentation of the final analysis results. As a result, we have focused primarily on the *use* of GNY logic and have not concentrated on its suitability from an academic viewpoint.

7.1 Summary of Results

In this section we will present the highlights of this dissertation, summarizing key features and conclusions that we have developed during the implementation of the GYPSIE, GYNGER and Visual GNY modules of the SPEAR II Framework. To illustrate the potential and value of the SPEAR II application we have used it to conduct GNY analyses of fifteen published protocols. The source files for all of these analyses are included in the SPEAR II distribution, the results of twelve of these analyses being shown in Appendix B and Appendix E. The analyses described in Appendix E are all taken from [1] and [35] and include well-known protocols such as the Yahalom, Wide-Mouth Frog, Needham-Schroeder Public-Key, Otway-Rees, Kerberos and Andrew Secure RPC Handshake Protocols. The protocols in Appendix B were used to initially test the GYNGER analyzer. Testing of the rounds analyzer was conducted using protocols from [35].

7.1.1 Security Protocol Modelling with GYPSIE

The GYPSIE environment facilitates the modelling of cryptographic protocols through the use of a graphical user interface divided into three levels of abstraction. GYPSIE currently functions as the core interface to the SPEAR II Framework and a given protocol must be specified therein before any further engineering operations can take place. The following list summarizes the salient features of GYPSIE:

- Divided into three views to facilitate abstraction and help protocol engineers distil the critical issues in a specification:
  - The *High-Level View* describes the overall flow of messages, using a formalism based on MSCs and SDL to represent the message passing specification.
  - The *Navigator View* appears adjacent to the High-Level View and summarizes the contents and structure of a protocol using a tree-view with expandable and collapsible nodes.
  - The *Component View* is invoked from the High-Level View and allows one to view and edit the contents of protocol messages, each message being displayed as a hierarchical tree.
- Includes support for embedding subprotocols in a specification. The use of subprotocols allows one to construct a subprotocol hierarchy, with the subprotocols in this hierarchy being executed sequentially or conditionally.
- Supports drag-and-drop operations in all three views to facilitate the reordering, copying and moving of messages, principals and components. The Navigator and High-Level Views are tightly integrated so that messages and principals can be dragged from the Navigator View into the High-Level View to facilitate copying between subprotocols.
- Protocol engineering operations can be performed on objects in all three views using pop-up or pull-down menus. Using pop-up menus or cut, copy and paste facilities, message components can be duplicated among messages, even if these messages reside in different subprotocols.
- Reference count-based garbage collection is used to ensure the consistency and uniformity of the memory used to store protocol specifications.
- Undo and redo functionality has been provided to facilitate experimentation and to allow users to easily recover from accidental engineering operations, such as deletions and edits.

- Includes a Component Tracker to help protocol engineers deal with the complexity that results from having a large number of components defined in a protocol. The Component Tracker allows users to highlight all of the locations where a given component appears on the design canvas in the High-Level View. Using this feature, components can be easily located and their movement through the protocol message passing specification can be clearly followed.

- Every component embedded within a message has an associated type. These types range from terminal types, such as nonces, timestamps and symmetric keys, to non-terminal types, such as hashes and public key encryptions. Non-terminal components serve as anchor points to which other components are attached. This approach leads to a hierarchical message representation which is displayed using a tree-view with expandable and collapsible nodes in the Component View. Different properties, behaviour and iconic representations are assigned to each individual type.

- Specifications can be saved for later use and exported to other formats. Once a protocol specification has been created, a user can generate plain text, LaTeX or Prolog output thereof. The Prolog output is used in conjunction with the GYNGER GNY-based protocol analyzer.

- Supports the incorporation of code generation routines. The message formalism allows one to specify actions to be executed and conditions to be examined before or after a message is sent or received. A properties tab has been added to components whose structure can be defined through the use of an ASN.1 specification. Finally, communications settings for principals and messages can be specified. These settings currently include details such as transport protocols, IP addresses and communications ports.

- A specification can be queried and manipulated by using the associated GYPSIE API. The API facilitates the expansion of the SPEAR II Framework by allowing engineering modules to easily retrieve information about a protocol that has been specified. The Visual GNY environment and message rounds calculator both use the API extensively.

Usability experiments were conducted with twenty individuals, each of whom had a security protocol engineering background. Each of them was asked to specify three cryptographic protocols in the GYPSIE environment and every one of them was able to construct the specifications in a reasonable amount of time, each specification being completed in under ten minutes on average. The error level was also reasonably low, with 75% of the resultant sixty specifications being error free.

7.1.2 Calculating Message Rounds

The message rounds calculator receives a message passing specification from GYPSIE and then returns the messages that can be sent together in parallel. This information helps to ensure that the most efficient protocol design in terms of message rounds can be deployed at the implementation stage, since the number of rounds resulting from the protocol model can be compared to the optimal number required for the protocol class into which the specification falls [35]. The SPEAR I tool carried out synchronous message rounds calculations. Synchronous rounds calculations assume that a principal can only transmit a given message once he has received all of the previous messages in the specification which were destined for him. Optimal rounds calculations assume that message transmission can take place asynchronously. SPEAR II implements both synchronous and optimal message rounds calculations and is thus able to assist protocol engineers in bridging the gap from design to actual implementation.

7.1.3 Automated GNY Analysis with GYNGER

GYNGER is a Prolog-based GNY analyzer that can be used to analyze cryptographic protocols. It is based on the analyzer described in [54] and allows one to represent all possible GNY statements, except for those involving eligibility. The most significant feature of GYNGER is that it uses a forward-chaining approach to automate the tedious application of GNY inference rules, allowing all derivable GNY statements to be generated quickly, accurately and efficiently. GYNGER includes some noteworthy improvements when compared to the analyzer in [54]:

- Implements seventy-two of the eighty-eight GNY inference rules, as opposed to the forty-eight implemented by the analyzer in [54].
- Incorporates syntax to represent shared secrets that are being used as identifiers in hashes, public key encryptions and symmetric encryptions.
- Includes support for the 'never-originated-here' binary operator.
- Features a vastly improved syntax for representing functions. This new syntax ensures that the GNY inference rules do not need to be reformulated for every function included in a protocol specification.

GYNGER does not implement all of the GNY inference rules since some of these rules are not suitable for forward-chaining as they give rise to infinite loops. However, the exclusion of these rules does not affect the useful inferences that can be derived. To conduct an analysis with GYNGER a protocol engineer needs to specify a protocol's messages, initial assumptions and target goals in a Prolog-style GNY syntax. The GNY rule set is then imported and employed in the analysis, after which a proof is generated in an *English-style* GNY syntax for every successful goal that was specified. This English-style proof lists all of the statements involved in the derivation of the successful goal, indicating the postulates that were used and the premises which were employed in the postulate's application. The fact that the proof is in an English-style syntax makes it more readable and comprehensible.

7.1.4 Visual GNY

The Visual GNY environment was created to facilitate GNY-based protocol analysis and works in close conjunction with the GYPSIE design environment. All of the information related to a protocol specification, specifically the message components and principals involved, are obtained by querying the GYPSIE environment. In essence, the Visual GNY environment functions as a user-friendly interface to the GYNGER analyzer. GNY statements necessary for an analysis are constructed in the Visual GNY interface and then passed on to GYNGER. Results from GYNGER are returned to the Visual GNY environment. The following list summarizes the salient features of the Visual GNY interface:

- GNY statements are represented as a collection of successive nodes in a tree-like structure, each node being a child of its predecessor. All statements of the same type form part of the same tree, a heterogeneous set of GNY statements being represented by a collection of separate trees. Each node within a GNY tree has an assigned type and its position within the tree is determined by that type. The term 'structured tree' is used to refer to this tree-like representation, since the tree nodes are ordered according to a predefined pattern in order to produce a meaningful representation of GNY information.

- GNY statements are grouped and collated in the environment using tabbed panes. There are four tabbed panes within the Visual GNY interface that store GNY statements. The *Assumptions* pane holds the structured trees representing a principal's initial belief and possession sets. Within this tabbed pane, the principal whose belief and possession sets are being examined is selected from a drop-down combo-box. The *Goals* pane contains the target belief and possession sets for each principal, while the *Results* pane contains the structured trees which describe the failed and successful target goals for each principal. A drop-down combo-box embedded in the *Extensions* tabbed pane is used to select the formula whose extensions appear in the tree-view associated with this pane.

- Users are guided through the use of pop-up menus when constructing structured trees that represent a collection of GNY statements. The pop-up menu which is displayed depends on the type of node which is currently selected. In this way, a user is given a limited, yet appropriate choice as to the nodes that can be attached to the one which is currently selected. As a result, the Visual GNY environment is able to guarantee that only syntactically correct structured trees are created. All formulae and principal names presented for insertion in a structured tree are extracted from the GYPSIE specification of the protocol, ensuring that no irrelevant information is used.

- Assistance is provided when reading a collection of structured trees. When hovering a mouse pointer over a terminal node in a given structured tree, the English-style GNY statement represented by that node is displayed. To view the GNY statements represented by a collection of structured trees in a given tree-view, the *View as Text* command, accessible from the pop-up menus, is used. The execution of this command causes a text dialog to be displayed which lists the English-style statements represented by each terminal node within the given tree-view.

- To aid in the construction of structured trees, selected pop-up menus suggest components that could be inserted at a given point in a structured tree. For example, if a principal originates a nonce, then he can believe that nonce to be fresh and recognizable.

- To avoid the duplication of statements within a given collection of structured trees, the contents of the pop-up menus are dynamically updated depending on the current state of the Visual GNY environment and the structured trees specified therein.

- The layout of the Visual GNY environment provides direction during the analysis process. The presence of the tabbed panes and the order in which they appear helps to remind users what needs to be specified before an analysis can proceed. Once all of the preconditions for an analysis have been specified, the postulate application process can be invoked by using the *Analyze* command which is available in the *Analysis* tabbed pane. This action initiates protocol parsing and then executes the GYNGER analyzer, passing all the information needed for analysis on to the GYNGER engine in the appropriate GYNGER syntax. Once an analysis is complete, the resulting output is retrieved, parsed and then displayed in the *Results* tabbed pane.

- The *Results* pane allows a user to easily access the GNY statements derived during a GNY analysis. The successful and failed possession and belief goals for each principal are displayed using structured tree sets. If a given set for a principal is empty, then that set is not accessible and the radio button used to switch to the set is dimmed. This allows one to quickly determine whether there are any failed or successful goals. All of the GNY statements generated by GYNGER can be viewed as English-style text, and the proof for a successful target goal can be viewed by right-clicking on the terminal node of its structured tree representation and selecting the *View Proof* command from the resulting pop-up menu.

Experiments were carried out to examine how well the structured tree metaphor facilitates GNY statement construction. Each of the twenty individuals involved in the experiments had previously been instructed in GNY analysis techniques and they had all carried out GNY analyses in the past. The experiments that we conducted required that they construct GNY statements by hand using mathematical notation, and in the Visual GNY environment using structured trees. We found that when using Visual GNY, they were able to correctly transcribe 98% of the GNY statements we presented to them. When transcribing a similar set of statements by hand into mathematical notation, their accuracy rate was only 78%. Errors made in the Visual GNY construction resulted from using incorrect components of the same type, for example using the symmetric key K_{as} instead of K_{ab}.

7.2 Future Work

The implementation of the SPEAR II Framework has been brought to the point where key areas of protocol specification and analysis have been realized. We hope to witness the expansion of the SPEAR II application through the implementation of additional engineering dimensions and techniques as per the diagram in Figure 1.2. In the list that follows, we suggest some projects that can be tackled to expand the current feature set of the SPEAR II tool:

- The GYNGER analyzer does not yet implement eligibility checking. In the context of this dissertation this feature was not that important as GYNGER is still able to perform high-quality analysis without having to concern itself with the feasibility of a protocol specification. However, the addition of eligibility checking would add a further level of completeness to the *Security Logics* dimension. Eligibility checking would probably have to be conducted apart from GYNGER, as it would cause the forward-chaining based analyzer to loop infinitely.

- The Visual GNY environment has presented a mechanism that can be used to automate protocol analysis using modal logics. We hope to see this approach expanded to incorporate other logics, such as BAN and SVO. Initially, BAN and SVO analyzers will have to be created, followed by suitable statement specification environments that integrate within the SPEAR II Framework. An interesting investigation would be to examine the feasibility of combining all of these specification environments into one unified interface that can cater for a wide array of logics.

- The goal of a GNY analysis is to determine whether a protocol achieves its design goals. However, individuals often struggle to determine what these goals should be or what initial conditions should exist in an analysis. We have tried to provide some guidance in this respect by offering initial belief and possession suggestions within the Visual GNY environment. However, the current level of guidance is minimal. We envisage the creation of a more advanced statement suggestion engine that would examine a protocol or use information supplied about a protocol to build a list of possible statements that would describe the initial and target states of a protocol.

- When carrying out an analysis, some of the initial beliefs, possessions or formula extensions specified in the Visual GNY environment do not get used in any of the proofs generated by GYNGER. Instead of being content with this situation, it would be convenient to be informed of these redundant statements. Such a feature would allow protocol engineers to add a number of initial beliefs, possessions and extensions to a protocol, conduct an analysis, and then afterwards examine which of these are redundant and not needed for a given collection of goals to succeed. In this way, we can work towards determining the optimal precondition sets.

- The Interrogator [24] and NRL Protocol Analyzer [57] are both powerful applications that can be used to reveal replay attacks to which a given protocol is susceptible. The GYPSIE design environment does not require any significant modifications to accommodate an attack analysis module of a similar nature, since most attack analyses only requires a message passing specification to proceed. However, an environment that can be used to specify penetration objectives and view the analysis results would have to be created to facilitate user interaction with the module.

- The addition of multiclass queuing network analysis to the *Performance Analysis* dimension would aid in building a protocol model that can help to predict the performance, likely bottlenecks and points of optimization in a given protocol specification. Such an analysis could be carried out as described in [8]. First performance measurements can be obtained through simulation or meta-execution of a number of protocol sessions. Then, this information can then be used to construct a queuing model in a stochastic analysis programming language such as *SnapL* [11]. Finally, this model can be analyzed for performance using a tool such as *MicroSnap* [26].

- The development of a source code generation module will significantly boost the practical value of the SPEAR II application. This module should have the ability to generate the necessary source code that will encode, send, receive and decode the messages defined in the message passing specification. Source code would effectively have to be produced for each of the principals taking part in a protocol. Target languages could include Java, C++ and SDL/PR. Implementations produced with the code generation module would have a higher confidence level, as the specification can be tested in the SPEAR II Framework by using the tools and techniques that exist therein before it is deployed. Furthermore, the chances of random bugs affecting the generated source code are smaller, since the code generation engine will remove the need for error-prone, repetitious and mundane programming tasks often carried out by programmers.

- The GYPSIE formalism can be upgraded to enable the modelling of multi-cast protocols. What is needed is the ability to add more receiver boxes to a given message so that there can be multiple recipients, but only one sender. These boxes would be linked with a solid line in the same way that current sender and receiver boxes are joined together. Multi-cast protocol compatibility would be complimented by a code generation dimension, allowing protocol engineers to easily create multi-cast protocols. However, the creation of multi-cast mechanisms could complicate some analysis techniques which are not able to accommodate multi-cast formalisms.

This list is by no means exhaustive and merely serves to illustrate how we envisage the development of the SPEAR II platform to progress. Our aim is to see the current SPEAR II application grow into a suite of protocol analysis tools, with GYPSIE or a similar environment acting as a unifying modelling environment. The implementation of the Visual GNY and GYNGER tools have shown the viability of the SPEAR II multi-dimensional approach by illustrating how an environment of this nature can be used to facilitate and enable techniques that are deemed complex and tedious to conduct by hand.

7.3 Contributions of This Dissertation

Through the course of this dissertation we have described an application that we developed to realize portions of the SPEAR II Multi-Dimensional Security Protocol Engineering Framework. The details of this work have primarily focused on implementation issues, graphical components, user interfaces, GNY syntax and Prolog source code. However, despite all of these intricate details, the motivating factor

behind this work has always been to *aid protocol engineers in distilling the critical issues during a protocol engineering session, presenting them with an appropriate level of detail and guiding them as much as possible in analysis and engineering processes.*

A novel outcome of this work has been the development of the Visual GNY environment. Research in the past has focused on the suitability and correctness of the GNY logic. Instead of delving into these issues, we have focused on developing a system that will facilitate rapid, accurate and high-quality GNY analyses. The Visual GNY environment frees protocol engineers to concentrate on an analysis and shields them from being burdened by the mechanics associated with inference rule application. Users of the Visual GNY environment must still be familiar with GNY logic and its underlying principles. However, they do not need to memorize the GNY notation, syntax and inference rules since Visual GNY and GYNGER handle all of these issues by using appropriate guidance and automation.

The GYNGER analyzer delivers on the functionality that Visual GNY and GYPSIE enable by actually implementing the mechanics of the analyses which they initially specify. When put into practice, GYNGER works solidly, delivering the expected results for every one of the GNY analyses it has been used to conduct. The fact that GYNGER is able to generate proofs in English-style syntax is a huge advantage to protocol engineers who want to be able to validate and closely examine the analysis process. The extensive syntax also helps to ensure that GYNGER can carry out a broad range of GNY analyses, catering for a number of diverse protocol types and GNY statements.

Steve Brackin has developed a graphically-based analysis system based on the BGNY modal logic [16]. His Convince toolkit includes a graphical interface for protocol design, and a HOL-based protocol analyzer. In essence, this dissertation is similar to the work Brackin has carried out. However, our aims are slightly divergent. Convince focuses primarily on protocol analysis. The associated analyzer, known as the AAPA2, is powerful, robust and well-known, however, the interface is textually-based, resulting in users having to familiarize themselves with its syntax. On the other hand, our focus has been on the user interface aspect of GNY analysis and we have attempted to create a usable and effective framework which can be employed to facilitate GNY analyses. Our system has a powerful and intuitive user interface, and a Prolog-based analyzer which has been tested on fifteen published cryptographic protocols.

Besides the creation of the Visual GNY environment, we have also designed and implemented a protocol design environment that is completely graphically-based. The concept of abstracting the protocol design into three complementary views helps engineers to visualize and manipulate a given protocol more easily. The GYPSIE environment essentially functions as the core enabler in the SPEAR II Framework and for this reason we have spent time making it as rich, robust and usable as possible. The modular implementation of the SPEAR II Framework, combined with the GYPSIE API, makes it viable to upgrade and extend the SPEAR II application to include more powerful cryptographic routines, such as code generation and attack analysis.

The SPEAR II tool has applications in both academic and commercial disciplines. In an academic sense it can be used when teaching and illustrating GNY analysis techniques [74], since it helps students to focus more on the important issues in an analysis, instead of burdening them with sideline issues such as inference rule application, syntactical and notational issues. In both the commercial and academic spheres, SPEAR II can be used to model and analyze cryptographic protocols that are to be or have already been implemented. The current implementation of the SPEAR II Framework supports GNY analysis and message rounds calculations. However, as more dimensions and techniques are added to the SPEAR II application, we feel that it will become a very valuable and effective tool to drive the concept and adoption of multi-dimensional protocol engineering.

APPENDICES

Appendix A

GNY Inference Rules

This appendix contains all eighty-eight of the GNY inference rules we have referenced through the course of this dissertation. The rules are categorized by type and each has an associated identification code that can be cited in proofs.

A.1 Being Told Rules

T1. $\dfrac{P \rightarrow Q: X, P \propto X}{Q \lhd X}$

T2. $\dfrac{P \lhd *X}{P \lhd X}$

T3. $\dfrac{P \lhd X \rightsquigarrow C}{P \lhd X}$

T4. $\dfrac{P \lhd (X, Y)}{P \lhd X}$

T5. $\dfrac{P \lhd F(X_0, X_1, ..., X_{n-1}), P \ni (X_0, ..., X_{i-1}, X_{i+1}, ..., X_{n-1})}{P \lhd X_i}$

T6. $\dfrac{P \lhd \{X\}_K, P \ni K}{P \lhd X}$

T7. $\dfrac{P \lhd \{X\}_{+K}, P \ni -K}{P \lhd X}$

T8. $\dfrac{P \lhd \{X\}_{-K}, P \ni +K}{P \lhd X}$

T9. $\dfrac{P \lhd <S>}{P \lhd S}$

A.2 Possession Rules

P1. $\dfrac{P \lhd X}{P \ni X}$

P2. $\frac{P \ni X \rightsquigarrow C}{P \ni X}$

P3. $\frac{P \ni X, P \ni Y}{P \ni (X,Y), P \ni F(X,Y)}$

P4. $\frac{P \ni (X, Y)}{P \ni X}$

P5. $\frac{P \ni F(X_0, X_1, ..., X_{n-1}), P \ni (X_0, ..., X_{i-1}, X_{i+1}, ..., X_{n-1})}{P \ni X_i}$

P6. $\frac{P \ni X}{P \ni H(X)}$

P7. $\frac{P \ni X, P \ni K}{P \ni \{X\}_K, P \ni \{X\}_K^{-1}}$

P8. $\frac{P \ni X, P \ni +K}{P \ni \{X\}_{+K}}$

P9. $\frac{P \ni X, P \ni -K}{P \ni \{X\}_{-K}}$

P10. $\frac{P \ni S, P \mid\equiv P \overset{S}{\longleftrightarrow} Q}{P \ni <S>}, S \notin \{K, +K, -K\}$

A.3 Eligibility Rules

E1. $\frac{P \ni X}{P \propto X}$

E2. $\frac{P \propto X}{P \propto *X}$

E3. $\frac{P \propto X, P \propto Y}{P \propto (X, Y), P \propto F(X, Y)}$

E4. $\frac{P \propto X, P \ni S, P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv C}{P \propto H(X, <S>) \rightsquigarrow C}$

E5. $\frac{P \propto X, P \ni K, P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv C}{P \propto \{X\}_K \rightsquigarrow C, P \propto \{X\}_K^{-1} \rightsquigarrow C}$

E6. $\frac{P \propto X, P \ni (S, +K), P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \overset{+K}{\mapsto} Q, P \mid\equiv C}{P \propto \{X, <S>\}_{+K} \rightsquigarrow C}$

E7. $\frac{P \propto X, P \ni -K, P \mid\equiv \overset{+K}{\mapsto} P, P \mid\equiv C}{P \propto \{X\}_{-K} \rightsquigarrow C}$

E8. $\frac{P \propto X}{P \propto H(X)}$

E9. $\dfrac{P \propto X, P \ni +K, P \mid\equiv \overset{+K}{\mapsto} Q}{P \propto \{X\}_{+K}}$

E10. $\dfrac{P \propto X, P \ni (S,K), P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv C}{P \propto \{X,<S>\}_K}$

A.4 Recognizability Rules

R1. $\dfrac{P \mid\equiv \phi(X), P \ni X \rightsquigarrow C}{P \mid\equiv \phi(X \rightsquigarrow C)}$

R2. $\dfrac{P \mid\equiv \phi(X \rightsquigarrow C), P \ni X}{P \mid\equiv \phi(X)}$

R3′. $\dfrac{P \mid\equiv \phi(X), P \ni (X, Y)}{P \mid\equiv \phi(X, Y)}$

R3″. $\dfrac{P \mid\equiv \phi(X), P \ni F(X)}{P \mid\equiv \phi(F(X))}$

R4. $\dfrac{P \mid\equiv \phi(X), P \ni K, P \ni \{X\}_K}{P \mid\equiv \phi(\{X\}_K)}, X \neq \{Y\}_K^{-1}$

R5. $\dfrac{P \mid\equiv \phi(X), P \ni (X, K), P \ni \{X\}_K^{-1}}{P \mid\equiv \phi(\{X\}_K^{-1})}, X \neq \{Y\}_K$

R6. $\dfrac{P \mid\equiv \phi(X), P \ni (X, +K), P \ni \{X\}_{+K}}{P \mid\equiv \phi(\{X\}_{+K})}, X \neq \{Y\}_{-K}$

R7. $\dfrac{P \mid\equiv \phi(X), P \ni (X, -K), P \ni \{X\}_{-K}}{P \mid\equiv \phi(\{X\}_{-K})}, X \neq \{Y\}_{+K}$

R8. $\dfrac{P \mid\equiv \phi(X), P \ni +K, P \ni \{X\}_{-K}}{P \mid\equiv \phi(\{X\}_{-K})}, X \neq \{Y\}_{+K}$

R9. $\dfrac{P \mid\equiv \phi(X), P \ni -K, P \ni \{X\}_{+K}}{P \mid\equiv \phi(\{X\}_{+K})}, X \neq \{Y\}_{-K}$

R10. $\dfrac{P \mid\equiv \phi(X), P \ni X, P \ni H(X)}{P \mid\equiv \phi(H(X))}$

R11. $\dfrac{P \ni H(X), P \mid\equiv \phi(H(X)), P \ni X}{P \mid\equiv \phi(X)}$

R12. $\dfrac{P \mid\equiv \phi(S); P \ni <S>}{P \mid\equiv \phi(<S>)}, S \notin \{K, +K, -K\}$

A.5 Freshness Rules

F1. $\dfrac{P \mid\equiv \sharp(X), P \ni X \rightsquigarrow C}{P \mid\equiv \sharp(X \rightsquigarrow C)}$

F2. $\dfrac{P \mid\equiv \sharp(X \rightsquigarrow C),\ P \ni X}{P \mid\equiv \sharp(X)}$

F3′. $\dfrac{P \mid\equiv \sharp(X),\ P \ni (X, Y)}{P \mid\equiv \sharp(X, Y)}$

F3″. $\dfrac{P \mid\equiv \sharp(X),\ P \ni F(X)}{P \mid\equiv \sharp(F(X))}$

F4. $\dfrac{P \mid\equiv \sharp(X),\ P \ni X,\ P \ni H(X)}{P \mid\equiv \sharp(H(X))}$

F5. $\dfrac{P \mid\equiv \sharp(X),\ P \ni K,\ P \ni \{X\}_K}{P \mid\equiv \sharp(\{X\}_K)},\ X \neq \{Y\}_K^{-1}$

F6. $\dfrac{P \mid\equiv \sharp(X),\ P \ni (X, K),\ P \ni \{X\}_K^{-1}}{P \mid\equiv \sharp(\{X\}_K^{-1})},\ X \neq \{Y\}_K$

F7. $\dfrac{P \mid\equiv \sharp(X),\ P \ni (X, +K),\ P \ni \{X\}_{+K}}{P \mid\equiv \sharp(\{X\}_{+K})},\ X \neq \{Y\}_{-K}$

F8. $\dfrac{P \mid\equiv \sharp(X),\ P \ni (X, -K),\ P \ni \{X\}_{-K}}{P \mid\equiv \sharp(\{X\}_{-K})},\ X \neq \{Y\}_{+K}$

F9. $\dfrac{P \mid\equiv \sharp(X),\ P \ni +K,\ P \ni \{X\}_{-K}}{P \mid\equiv \sharp(\{X\}_{-K})},\ X \neq \{Y\}_{+K}$

F10. $\dfrac{P \mid\equiv \sharp(X),\ P \ni -K,\ P \ni \{X\}_{+K}}{P \mid\equiv \sharp(\{X\}_{+K})},\ X \neq \{Y\}_{-K}$

F11. $\dfrac{P \mid\equiv \sharp(+K),\ P \ni -K}{P \mid\equiv \sharp(-K)}$

F12. $\dfrac{P \mid\equiv \sharp(-K),\ P \ni +K}{P \mid\equiv \sharp(+K)}$

F13. $\dfrac{P \mid\equiv \phi(X),\ P \mid\equiv \sharp(K),\ P \ni K,\ P \ni \{X\}_K}{P \mid\equiv \sharp(\{X\}_K)},\ X \neq \{Y\}_K^{-1}$

F14. $\dfrac{P \mid\equiv \phi(X),\ P \mid\equiv \sharp(K),\ P \ni (X, K),\ P \ni \{X\}_K^{-1}}{P \mid\equiv \sharp(\{X\}_K^{-1})},\ X \neq \{Y\}_K$

F15. $\dfrac{P \mid\equiv \phi(X),\ P \mid\equiv \sharp(+K),\ P \ni (X, +K),\ P \ni \{X\}_{+K}}{P \mid\equiv \sharp(\{X\}_{+K})},\ X \neq \{Y\}_{-K}$

F16. $\dfrac{P \mid\equiv \phi(X),\ P \mid\equiv \sharp(-K),\ P \ni (X, -K),\ P \ni \{X\}_{-K}}{P \mid\equiv \sharp(\{X\}_{-K})},\ X \neq \{Y\}_{+K}$

F17. $\dfrac{P \mid\equiv \phi(X),\ P \mid\equiv \sharp(-K),\ P \ni -K,\ P \ni \{X\}_{+K}}{P \mid\equiv \sharp(\{X\}_{+K})},\ X \neq \{Y\}_{-K}$

F18. $\dfrac{P \mid\equiv \phi(X),\ P \mid\equiv \sharp(+K),\ P \ni +K,\ P \ni \{X\}_{-K}}{P \mid\equiv \sharp(\{X\}_{-K})},\ X \neq \{Y\}_{+K}$

F19. $\dfrac{P \mid\equiv \sharp(H(X)), P \ni H(X), P \ni X}{P \mid\equiv \sharp(X)}$

F20. $\dfrac{P \mid\equiv \sharp(S); P \ni <S>}{P \mid\equiv \sharp(<S>)}, S \notin \{K, +K, -K\}$

A.6 Conveyance Rules

C1. $\dfrac{P \lhd *\{X\}_K \rightsquigarrow C, P \ni K, P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \phi(X), P \mid\equiv \sharp(X, K)}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K \rightsquigarrow C, P \mid\equiv Q \ni (X, K)}$

C2. $\dfrac{P \lhd *\{X\}_K^{-1} \rightsquigarrow C, P \ni (X, K), P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \phi(X), P \mid\equiv \sharp(X, K)}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K^{-1} \rightsquigarrow C, P \mid\equiv Q \ni (X, K)}$

C3. $\dfrac{P \lhd *H(X, <S>) \rightsquigarrow C, P \ni (X, S), P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \sharp(X, S)}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim H(X, <S>) \rightsquigarrow C, P \mid\equiv Q \ni (X, S)}$

C4. $\dfrac{P \lhd *\{X, <S>\}_{+K} \rightsquigarrow C, P \ni -K, P \mid\equiv \overset{+K}{\mapsto} P, P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \phi(X, S), P \mid\equiv \sharp(X, S, +K)}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim \{X, <S>\}_{+K} \rightsquigarrow C, P \mid\equiv Q \ni (X, S, +K)}$

C5. $\dfrac{P \lhd *\{X, <S>\}_{+K} \rightsquigarrow C, P \ni (X, +K, S), P \mid\equiv \overset{+K}{\mapsto} Q, P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \phi(X, S), P \mid\equiv \sharp(X, S, +K)}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim \{X, <S>\}_{+K} \rightsquigarrow C, P \mid\equiv Q \ni (X, S, +K)}$

C6. $\dfrac{P \lhd *\{X\}_{-K} \rightsquigarrow C, P \ni +K, P \mid\equiv \overset{+K}{\mapsto} Q, P \mid\equiv \phi(X)}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_{-K} \rightsquigarrow C}$

C7. $\dfrac{P \lhd *\{X\}_{-K}, P \ni +K, P \mid\equiv \overset{+K}{\mapsto} Q, P \mid\equiv \phi(X), P \mid\equiv \sharp(X, -K)}{P \mid\equiv Q \ni (X, -K)}$

C8. $\dfrac{P \mid\equiv Q \mid\sim X \rightsquigarrow C}{P \mid\equiv Q \mid\sim X}$

C9. $\dfrac{P \mid\equiv Q \mid\sim (X, Y)}{P \mid\equiv Q \mid\sim X}$

C10. $\dfrac{P \lhd *\{X\}_K \rightsquigarrow C, P \ni K, P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \phi(X), P \mid\equiv P \dashv (\{X\}_K)}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K \rightsquigarrow C}$

C11. $\dfrac{P \lhd *\{X\}_K^{-1} \rightsquigarrow C, P \ni (X, K), P \mid\equiv P \overset{K}{\longleftrightarrow} Q, P \mid\equiv \phi(X), P \mid\equiv P \dashv (\{X\}_K^{-1})}{P \mid\equiv Q \mid\sim X, P \mid\equiv Q \mid\sim \{X\}_K^{-1} \rightsquigarrow C}$

C12. $\dfrac{P \lhd *H(X, <S>) \rightsquigarrow C, P \ni (X, S), P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv P \dashv (H(X, <S>))}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim H(X, <S>) \rightsquigarrow C}$

C13. $\dfrac{P \lhd *\{X, <S>\}_{+K} \rightsquigarrow C, P \ni -K, P \mid\equiv \overset{+K}{\mapsto} P, P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \phi(X, S), P \mid\equiv P \dashv (\{X, <S>\}_{+K})}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim \{X, <S>\}_{+K} \rightsquigarrow C}$

C14. $\dfrac{P \lhd *\{X, <S>\}_{+K} \rightsquigarrow C, P \ni (X, S, +K), P \mid\equiv \overset{+K}{\mapsto} Q, P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \phi(X, S), P \mid\equiv P \dashv (\{X, <S>\}_{+K})}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim \{X, <S>\}_{+K} \rightsquigarrow C}$

C15. $$\frac{P \lhd *\{X, <S>\}_K \rightsquigarrow C, P \ni K, P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \phi(X, S), P \mid\equiv \sharp(X, S, K)}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim \{X, <S>\}_K \rightsquigarrow C, P \mid\equiv Q \ni (X, S, K)}$$

C16. $$\frac{P \lhd *\{X, <S>\}_K \rightsquigarrow C, P \ni K, P \mid\equiv P \overset{S}{\longleftrightarrow} Q, P \mid\equiv \phi(X, S), P \mid\equiv \dashv (\{X, <S>\}_K)}{P \mid\equiv Q \mid\sim (X, S), P \mid\equiv Q \mid\sim \{X, <S>\}_K \rightsquigarrow C}$$

C17. $$\frac{P \mid\equiv Q \mid\sim X \rightsquigarrow (C, C')}{P \mid\equiv Q \mid\sim X \rightsquigarrow C}$$

C18. $$\frac{P \mid\equiv Q \mid\sim X, P \mid\equiv \sharp(X)}{P \mid\equiv Q \ni X}$$

C19. $$\frac{P \mid\equiv Q \ni (X, Y)}{P \mid\equiv Q \ni X}$$

C20. $$\frac{P \mid\equiv Q \overset{S}{\longleftrightarrow} R}{P \mid\equiv R \overset{S}{\longleftrightarrow} Q}$$

C21. $$\frac{P \mid\equiv Q \dashv (X \rightsquigarrow C)}{P \mid\equiv Q \dashv (X)}$$

C22. $$\frac{P \mid\equiv Q \overset{S \rightsquigarrow C}{\longleftrightarrow} R}{P \mid\equiv Q \overset{S}{\longleftrightarrow} R}$$

A.7 Jurisdiction Rules

J1. $$\frac{P \mid\equiv Q \mid\Longrightarrow C, P \mid\equiv Q \mid\equiv C}{P \mid\equiv C}$$

J2. $$\frac{P \mid\equiv Q \mid\Longrightarrow Q \mid\equiv *, P \mid\equiv Q \mid\sim (X \rightsquigarrow C), P \mid\equiv \sharp(X)}{P \mid\equiv Q \mid\equiv C}$$

J3. $$\frac{P \mid\equiv Q \mid\Longrightarrow Q \mid\equiv *, P \mid\equiv Q \mid\equiv Q \mid\equiv C}{P \mid\equiv Q \mid\equiv C}$$

Appendix B

Proofs Generated by GYNGER

The proofs of the protocol goals in this appendix were all automatically generated by the GYPSIE Prolog-based GNY analyzer. To save space we have omitted mentioning the initial belief and possession sets associated with each protocol. However, statements that are part of the initial belief or possession sets are indicated by the text "Assumption" in curly brackets within the various proofs. Statements that are part of the protocol specification are tagged with the text "Step".

B.1 Voting Protocol

Idealized Protocol Message Flows:

(1) $Q \longrightarrow P_i : N_q$
(2) $P_i \longrightarrow Q : P_i, N_i, v_i, *H(N_q, < S_i >, v_i)$
(3) $Q \longrightarrow P_i : result, *H(N_i, < S_i >, result)$

Proof for Q believes that Pi once conveyed vi:

```
1.  Q was told (Pi, Ni, vi, *H(Nq, <Si>, vi)).  {Step}
2.  Q possesses Si.  {Assumption}
3.  Q believes that Si is a suitable secret for use between Q and Pi.  {Assumption}
4.  Q possesses Nq.  {Assumption}
5.  Q believes that Nq is fresh.  {Assumption}
6.  Q was told (Ni, vi, *H(Nq, <Si>, vi)).  {1, T4}
7.  Q possesses (Ni, vi, *H(Nq, <Si>, vi)).  {6, P1}
8.  Q possesses (vi, *H(Nq, <Si>, vi)).  {7, P4}
9.  Q was told (vi, *H(Nq, <Si>, vi)).  {6, T4}
10. Q possesses vi.  {8, P4}
11. Q was told *H(Nq, <Si>, vi).  {9, T4}
12. Q believes that Pi once conveyed ((Nq, vi), Si).  {11, 10, 4, 2, 3, 5, C3}
13. Q believes that Pi once conveyed (Nq, vi).  {12, C9}
14. Q believes that Pi once conveyed vi.  {13, C9}
```

Proof for Pi believes that Q once conveyed result:

```
1.  Pi was told (result, *H(Ni, <Si>, result)).  {Step}
```

```
2.  Pi possesses Si.  {Assumption}
3.  Pi believes that Si is a suitable secret for use between Q and Pi.  {Assumption}
4.  Pi possesses Ni.  {Assumption}
5.  Pi believes that Ni is fresh.  {Assumption}
6.  Pi was told result.  {1, T4}
7.  Pi was told *H(Ni, <Si>, result).  {1, T4}
8.  Pi possesses result.  {6, P1}
9.  Pi believes that Si is a suitable secret for use between Pi and Q.  {3, C20}
10. Pi believes that Q once conveyed ((Ni, result), Si).  {7, 8, 4, 2, 9, 5, C3}
11. Pi believes that Q once conveyed (Ni, result).  {10, C9}
12. Pi believes that Q once conveyed result.  {11, C9}
```

Proof for Q believes that H(Nq, <Si>, vi) is fresh:

```
1.  Q was told (Pi, Ni, vi, *H(Nq, <Si>, vi)).  {Step}
2.  Q possesses Si.  {Assumption}
3.  Q believes that Si is a suitable secret for use between Q and Pi.  {Assumption}
4.  Q possesses Nq.  {Assumption}
5.  Q believes that Nq is fresh.  {Assumption}
6.  Q was told (Ni, vi, *H(Nq, <Si>, vi)).  {1, T4}
7.  Q possesses (Ni, vi, *H(Nq, <Si>, vi)).  {6, P1}
8.  Q possesses (vi, *H(Nq, <Si>, vi)).  {7, P4}
9.  Q possesses <Si>.  {2, 3, P10}
10. Q was told (vi, *H(Nq, <Si>, vi)).  {6, T4}
11. Q possesses vi.  {8, P4}
12. Q was told *H(Nq, <Si>, vi).  {10, T4}
13. Q was told H(Nq, <Si>, vi).  {12, T2}
14. Q possesses H(Nq, <Si>, vi).  {13, P1}
15. Q believes that H(Nq, <Si>, vi) is fresh.  {5, 11, 9, 4, 14, F4}
```

Proof for Pi believes that H(Ni, <Si>, result) is fresh:

```
1.  Pi was told (result, *H(Ni, <Si>, result)).  {Step}
2.  Pi possesses Si.  {Assumption}
3.  Pi believes that Si is a suitable secret for use between Q and Pi.  {Assumption}
4.  Pi possesses Ni.  {Assumption}
5.  Pi believes that Ni is fresh.  {Assumption}
6.  Pi was told result.  {1, T4}
7.  Pi was told *H(Ni, <Si>, result).  {1, T4}
8.  Pi possesses result.  {6, P1}
9.  Pi believes that Si is a suitable secret for use between Pi and Q.  {3, C20}
10. Pi was told H(Ni, <Si>, result).  {7, T2}
11. Pi possesses H(Ni, <Si>, result).  {10, P1}
12. Pi possesses <Si>.  {2, 9, P10}
13. Pi believes that H(Ni, <Si>, result) is fresh.  {5, 8, 12, 4, 11, F4}
```

B.2 Information Exchange Protocol

Idealized Protocol Message Flows:

(1) $A \longrightarrow B : *\{T_a, B, X, S\}_{K_{ab}} \rightsquigarrow A \overset{S}{\longleftrightarrow} B$
(2) $B \longrightarrow A : *H(X, < S >) \rightsquigarrow A \mid\sim X$

Proof for B possesses X:

1. B was told *E(Kab : Ta, B, X, S) -> (S is a suitable secret for use between A and B). {Step}
2. B possesses Kab. {Assumption}
3. B was told *E(Kab : Ta, B, X, S). {1, T3}
4. B was told E(Kab : Ta, B, X, S). {3, T2}
5. B was told (Ta, B, X, S). {4, 2, T6}
6. B possesses (Ta, B, X, S). {5, P1}
7. B possesses (B, X, S). {6, P4}
8. B possesses (X, S). {7, P4}
9. B possesses X. {8, P4}

Proof for B believes that A once conveyed X:

1. B was told *E(Kab : Ta, B, X, S) -> (S is a suitable secret for use between A and B). {Step}
2. B possesses Kab. {Assumption}
3. B believes that B is recognizable. {Assumption}
4. B believes that Ta is fresh. {Assumption}
5. B believes that Kab is a suitable secret for use between A and B. {Assumption}
6. B was told *E(Kab : Ta, B, X, S). {1, T3}
7. B believes that Kab is a suitable secret for use between B and A. {5, C20}
8. B was told E(Kab : Ta, B, X, S). {6, T2}
9. B was told (Ta, B, X, S). {8, 2, T6}
10. B possesses (Ta, B, X, S). {9, P1}
11. B believes that (Ta, B, X, S) is recognizable. {3, 10, R3'}
12. B believes that (Ta, B, X, S) is fresh. {4, 10, F3'}
13. B believes that A once conveyed (Ta, B, X, S). {6, 2, 7, 11, 12, C1}
14. B believes that A once conveyed (B, X, S). {13, C9}
15. B believes that A once conveyed (X, S). {14, C9}
16. B believes that A once conveyed X. {15, C9}

Proof for A believes that B believes that A once conveyed X:

1. A was told *H(X, <S>) -> (A once conveyed X). {Step}
2. A possesses (Ta, B, X, S, Kab). {Assumption}
3. A believes that S is fresh. {Assumption}
4. A believes that S is a suitable secret for use between A and B. {Assumption}
5. A believes that B is trustworthy. {Assumption}
6. A was told *H(X, <S>). {1, T3}
7. A possesses (B, X, S, Kab). {2, P4}
8. A was told H(X, <S>). {6, T2}
9. A possesses H(X, <S>). {8, P1}
10. A possesses (X, S, Kab). {7, P4}
11. A possesses X. {10, P4}
12. A possesses (S, Kab). {10, P4}
13. A possesses S. {12, P4}
14. A possesses <S>. {13, 4, P10}
15. A believes that <S> is fresh. {3, 14, F20}
16. A believes that B once conveyed H(X, <S>) -> (A once conveyed X). {1, 11, 13, 4, 3, C3}
17. A believes that H(X, <S>) is fresh. {15, 14, 11, 9, F4}
18. A believes that B believes that A once conveyed X. {5, 16, 17, J2}

Proof for A believes that B possesses X:

```
1. A was told *H(X, <S>) -> (A once conveyed X).  {Step}
2. A possesses (Ta, B, X, S, Kab).  {Assumption}
3. A believes that S is fresh.  {Assumption}
4. A believes that S is a suitable secret for use between A and B.  {Assumption}
5. A was told *H(X, <S>).  {1, T3}
6. A possesses (B, X, S, Kab).  {2, P4}
7. A possesses (X, S, Kab).  {6, P4}
8. A possesses X.  {7, P4}
9. A possesses (S, Kab).  {7, P4}
10. A possesses S.  {9, P4}
11. A believes that B possesses (X, S).  {5, 8, 10, 4, 3, C3}
12. A believes that B possesses X.  {11, C19}
```

B.3 Authentication Protocol

Idealized Protocol Message Flows:

(1) $A \longrightarrow B : A, N_a$

(2) $B \longrightarrow A : \{B, *\{N_a\}_{-K_b} \rightsquigarrow A \overset{K}{\longleftrightarrow} B, K, N_b\}_{+K_a}, *\{N_a\}_K$

(3) $A \longrightarrow B : *\{N_b\}_K \rightsquigarrow A \overset{K}{\longleftrightarrow} B$

Proof for A possesses K:

```
1.  A was told (E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A
    and B), K, Nb), *E(K : Na)).  {Step}
2.  A possesses (A, Na, -Ka, +Kb).  {Assumption}
3.  A was told E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A
    and B), K, Nb).  {1, T4}
4.  A possesses (Na, -Ka, +Kb).  {2, P4}
5.  A possesses (-Ka, +Kb).  {4, P4}
6.  A possesses -Ka.  {5, P4}
7.  A was told (B, *E(-Kb : Na) -> (K is a suitable secret for use between A and B),
    K, Nb).  {3, 6, T7}
8.  A possesses (B, *E(-Kb : Na) -> (K is a suitable secret for use between A
    and B), K, Nb).  {7, P1}
9.  A possesses (*E(-Kb : Na) -> (K is a suitable secret for use between A and B),
    K, Nb).  {8, P4}
10. A possesses (K, Nb).  {9, P4}
11. A possesses K.  {10, P4}
```

Proof for A believes that B possesses K:

```
1.  A was told (E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A
    and B), K, Nb), *E(K : Na)).  {Step}
2.  A possesses (A, Na, -Ka, +Kb).  {Assumption}
3.  A believes that Na is recognizable.  {Assumption}
4.  A believes that Na is fresh.  {Assumption}
5.  A believes that +Kb is a suitable public key for B.  {Assumption}
6.  A believes that B is trustworthy.  {Assumption}
7.  A believes that B has jurisdiction over the statement "K is a suitable secret
```

```
    for use between A and B".  {Assumption}
8.  A was told E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A
    and B), K, Nb).  {1, T4}
9.  A was told *E(K : Na).  {1, T4}
10. A possesses (Na, -Ka, +Kb).  {2, P4}
11. A possesses (-Ka, +Kb).  {10, P4}
12. A possesses -Ka.  {11, P4}
13. A possesses +Kb.  {11, P4}
14. A was told (B, *E(-Kb : Na) -> (K is a suitable secret for use between A and B),
    K, Nb).  {8, 12, T7}
15. A possesses (B, *E(-Kb : Na) -> (K is a suitable secret for use between A
    and B), K, Nb).  {14, P1}
16. A possesses (*E(-Kb : Na) -> (K is a suitable secret for use between A and B),
    K, Nb).  {15, P4}
17. A was told (*E(-Kb : Na) -> (K is a suitable secret for use between A and B),
    K, Nb).  {14, T4}
18. A possesses (K, Nb).  {16, P4}
19. A was told *E(-Kb : Na) -> (K is a suitable secret for use between A and B).
    {17, T4}
20. A possesses K.  {18, P4}
21. A believes that B once conveyed E(-Kb : Na) -> (K is a suitable secret for use
    between A and B).  {19, 13, 5, 3, C6}
22. A was told *E(-Kb : Na).  {19, T3}
23. A was told E(-Kb : Na).  {22, T2}
24. A possesses E(-Kb : Na).  {23, P1}
25. A believes that E(-Kb : Na) is fresh.  {4, 13, 24, F9}
26. A believes that B believes that K is a suitable secret for use between A and B.
    {6, 21, 25, J2}
27. A believes that K is a suitable secret for use between A and B.  {7, 26, J1}
28. A believes that B possesses (Na, K).  {9, 20, 27, 3, 4, C1}
29. A believes that B possesses K.  {28, C19}
```

Proof for B believes that A possesses K:

```
1.  B was told *E(K : Nb) -> (K is a suitable secret for use between A and B).
    {Step}
2.  B possesses (B, Nb, +Ka, -Kb, K).  {Assumption}
3.  B believes that Nb is recognizable.  {Assumption}
4.  B believes that Nb is fresh.  {Assumption}
5.  B believes that K is a suitable secret for use between A and B.  {Assumption}
6.  B was told *E(K : Nb).  {1, T3}
7.  B possesses (Nb, +Ka, -Kb, K).  {2, P4}
8.  B believes that K is a suitable secret for use between B and A.  {5, C20}
9.  B possesses (+Ka, -Kb, K).  {7, P4}
10. B possesses (-Kb, K).  {9, P4}
11. B possesses K.  {10, P4}
12. B believes that A possesses (Nb, K).  {6, 11, 8, 3, 4, C1}
13. B believes that A possesses K.  {12, C19}
```

Proof for A believes that K is a suitable secret for use between A and B:

```
1. A was told (E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A
   and B), K, Nb), *E(K : Na)).  {Step}
2. A possesses (A, Na, -Ka, +Kb).  {Assumption}
3. A believes that Na is recognizable.  {Assumption}
```

4. A believes that Na is fresh. {Assumption}
5. A believes that +Kb is a suitable public key for B. {Assumption}
6. A believes that B is trustworthy. {Assumption}
7. A believes that B has jurisdiction over the statement "K is a suitable secret for use between A and B". {Assumption}
8. A was told E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A and B), K, Nb). {1, T4}
9. A possesses (Na, -Ka, +Kb). {2, P4}
10. A possesses (-Ka, +Kb). {9, P4}
11. A possesses -Ka. {10, P4}
12. A possesses +Kb. {10, P4}
13. A was told (B, *E(-Kb : Na) -> (K is a suitable secret for use between A and B), K, Nb). {8, 11, T7}
14. A was told (*E(-Kb : Na) -> (K is a suitable secret for use between A and B), K, Nb). {13, T4}
15. A was told *E(-Kb : Na) -> (K is a suitable secret for use between A and B). {14, T4}
16. A believes that B once conveyed E(-Kb : Na) -> (K is a suitable secret for use between A and B). {15, 12, 5, 3, C6}
17. A was told *E(-Kb : Na). {15, T3}
18. A was told E(-Kb : Na). {17, T2}
19. A possesses E(-Kb : Na). {18, P1}
20. A believes that E(-Kb : Na) is fresh. {4, 12, 19, F9}
21. A believes that B believes that K is a suitable secret for use between A and B. {6, 16, 20, J2}
22. A believes that K is a suitable secret for use between A and B. {7, 21, J1}

Proof for A believes that B believes that K is a suitable secret for use between A and B:

1. A was told (E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A and B), K, Nb), *E(K : Na)). {Step}
2. A possesses (A, Na, -Ka, +Kb). {Assumption}
3. A believes that Na is recognizable. {Assumption}
4. A believes that Na is fresh. {Assumption}
5. A believes that +Kb is a suitable public key for B. {Assumption}
6. A believes that B is trustworthy. {Assumption}
7. A was told E(+Ka : B, *E(-Kb : Na) -> (K is a suitable secret for use between A and B), K, Nb). {1, T4}
8. A possesses (Na, -Ka, +Kb). {2, P4}
9. A possesses (-Ka, +Kb). {8, P4}
10. A possesses -Ka. {9, P4}
11. A possesses +Kb. {9, P4}
12. A was told (B, *E(-Kb : Na) -> (K is a suitable secret for use between A and B), K, Nb). {7, 10, T7}
13. A was told (*E(-Kb : Na) -> (K is a suitable secret for use between A and B), K, Nb). {12, T4}
14. A was told *E(-Kb : Na) -> (K is a suitable secret for use between A and B). {13, T4}
15. A believes that B once conveyed E(-Kb : Na) -> (K is a suitable secret for use between A and B). {14, 11, 5, 3, C6}
16. A was told *E(-Kb : Na). {14, T3}
17. A was told E(-Kb : Na). {16, T2}
18. A possesses E(-Kb : Na). {17, P1}
19. A believes that E(-Kb : Na) is fresh. {4, 11, 18, F9}
20. A believes that B believes that K is a suitable secret for use between A and B. {6, 15, 19, J2}

Proof for B believes that A believes that K is a suitable secret for use between A and B:

```
1.  B was told *E(K : Nb) -> (K is a suitable secret for use between A and B).
    {Step}
2.  B possesses (B, Nb, +Ka, -Kb, K).  {Assumption}
3.  B believes that Nb is recognizable.  {Assumption}
4.  B believes that Nb is fresh.  {Assumption}
5.  B believes that K is a suitable secret for use between A and B.  {Assumption}
6.  B believes that A is trustworthy.  {Assumption}
7.  B was told *E(K : Nb).  {1, T3}
8.  B possesses (Nb, +Ka, -Kb, K).  {2, P4}
9.  B believes that K is a suitable secret for use between B and A.  {5, C20}
10. B was told E(K : Nb).  {7, T2}
11. B possesses E(K : Nb).  {10, P1}
12. B possesses (+Ka, -Kb, K).  {8, P4}
13. B possesses (-Kb, K).  {12, P4}
14. B possesses K.  {13, P4}
15. B believes that E(K : Nb) is fresh.  {4, 14, 11, F5}
16. B believes that A once conveyed E(K : Nb) -> (K is a suitable secret for use
    between A and B).  {1, 14, 9, 3, 4, C1}
17. B believes that A believes that K is a suitable secret for use between A
    and B.  {6, 16, 15, J2}
```

B.4 Needham-Schroeder Protocol

Idealized Protocol Message Flows:

(1) $P \longrightarrow Q : P$

(2) $Q \longrightarrow P : *\{P, N_{q_1}\}_{K_{qs}}$

(3) $P \longrightarrow S : P, Q, N_p, *\{P, N_{q_1}\}_{K_{qs}}$

(4) $S \longrightarrow P : *\{N_p, Q, K, *\{K, N_{q_1}, P\}_{K_{qs}} \rightsquigarrow P \overset{K}{\longleftrightarrow} Q\}_{K_{ps}} \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$

(5) $P \longrightarrow Q : N_{p_1}, *\{K, N_{q_1}, P\}_{K_{qs}} \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$

(6) $Q \longrightarrow P : *\{N_q, N_{p_1}\}_K \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$

(7) $P \longrightarrow Q : *\{F(N_q)\}_K \rightsquigarrow P \overset{K}{\longleftrightarrow} Q$

Proof for P possesses K:

```
1.  P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret
    for use between P and Q)) -> (K is a suitable secret for use between P and Q).
    {Step}
2.  P possesses (P, Q, Np, Np1, Kps).  {Assumption}
3.  P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret
    for use between P and Q)).  {1, T3}
4.  P possesses (Q, Np, Np1, Kps).  {2, P4}
5.  P was told E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret
    for use between P and Q)).  {3, T2}
6.  P possesses (Np, Np1, Kps).  {4, P4}
7.  P possesses (Np1, Kps).  {6, P4}
8.  P possesses Kps.  {7, P4}
9.  P was told (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use
    between P and Q)).  {5, 8, T6}
10. P possesses (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use
```

between P and Q)). {9, P1}
11. P possesses (Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {10, P4}
12. P possesses (K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {11, P4}
13. P possesses K. {12, P4}

Proof for Q possesses K:

1. Q was told (*E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q), Np1). {Step}
2. Q possesses (Nq, Nq1, Kqs). {Assumption}
3. Q was told *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {1, T4}
4. Q possesses (Nq1, Kqs). {2, P4}
5. Q was told *E(Kqs : K, Nq1, P). {3, T3}
6. Q possesses Kqs. {4, P4}
7. Q was told E(Kqs : K, Nq1, P). {5, T2}
8. Q was told (K, Nq1, P). {7, 6, T6}
9. Q possesses (K, Nq1, P). {8, P1}
10. Q possesses K. {9, P4}

Proof for P believes that K is a suitable secret for use between P and Q:

1. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) -> (K is a suitable secret for use between P and Q). {Step}
2. P possesses (P, Q, Np, Np1, Kps). {Assumption}
3. P believes that Np is fresh. {Assumption}
4. P believes that Np is recognizable. {Assumption}
5. P believes that Kps is a suitable secret for use between P and S. {Assumption}
6. P believes that S has jurisdiction over the statement "K is a suitable secret for use between P and Q". {Assumption}
7. P believes that S is trustworthy. {Assumption}
8. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {1, T3}
9. P possesses (Q, Np, Np1, Kps). {2, P4}
10. P was told E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {8, T2}
11. P possesses E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {10, P1}
12. P possesses (Np, Np1, Kps). {9, P4}
13. P possesses (Np1, Kps). {12, P4}
14. P possesses Kps. {13, P4}
15. P was told (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {10, 14, T6}
16. P possesses (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {15, P1}
17. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is recognizable. {4, 16, R3'}
18. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is fresh. {3, 16, F3'}
19. P believes that E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is fresh. {18, 14, 11, F5}
20. P believes that S once conveyed E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is

a suitable secret for use between P and Q)) -> (K is a suitable secret for use between P and Q). {1, 14, 5, 17, 18, C1}
21. P believes that S believes that K is a suitable secret for use between P and Q. {7, 20, 19, J2}
22. P believes that K is a suitable secret for use between P and Q. {6, 21, J1}

Proof for Q believes that K is a suitable secret for use between P and Q:

1. Q was told (*E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q), Np1). {Step}
2. Q possesses (Nq, Nq1, Kqs). {Assumption}
3. Q believes that Nq1 is fresh. {Assumption}
4. Q believes that Nq1 is recognizable. {Assumption}
5. Q believes that Kqs is a suitable secret for use between Q and S. {Assumption}
6. Q believes that S has jurisdiction over the statement "K is a suitable secret for use between P and Q". {Assumption}
7. Q believes that S is trustworthy. {Assumption}
8. Q was told *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {1, T4}
9. Q possesses (Nq1, Kqs). {2, P4}
10. Q was told *E(Kqs : K, Nq1, P). {8, T3}
11. Q possesses Kqs. {9, P4}
12. Q was told E(Kqs : K, Nq1, P). {10, T2}
13. Q was told (K, Nq1, P). {12, 11, T6}
14. Q possesses (K, Nq1, P). {13, P1}
15. Q possesses E(Kqs : K, Nq1, P). {12, P1}
16. Q believes that (K, Nq1, P) is recognizable. {4, 14, R3'}
17. Q believes that (K, Nq1, P) is fresh. {3, 14, F3'}
18. Q believes that E(Kqs : K, Nq1, P) is fresh. {17, 11, 15, F5}
19. Q believes that S once conveyed E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {8, 11, 5, 16, 17, C1}
20. Q believes that S believes that K is a suitable secret for use between P and Q. {7, 19, 18, J2}
21. Q believes that K is a suitable secret for use between P and Q. {6, 20, J1}

Proof for P believes that S once conveyed K:

1. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) -> (K is a suitable secret for use between P and Q). {Step}
2. P possesses (P, Q, Np, Np1, Kps). {Assumption}
3. P believes that Np is fresh. {Assumption}
4. P believes that Np is recognizable. {Assumption}
5. P believes that Kps is a suitable secret for use between P and S. {Assumption}
6. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {1, T3}
7. P possesses (Q, Np, Np1, Kps). {2, P4}
8. P was told E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {6, T2}
9. P possesses (Np, Np1, Kps). {7, P4}
10. P possesses (Np1, Kps). {9, P4}
11. P possesses Kps. {10, P4}
12. P was told (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {8, 11, T6}
13. P possesses (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use

between P and Q)). {12, P1}
14. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is recognizable. {4, 13, R3'}
15. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is fresh. {3, 13, F3'}
16. P believes that S once conveyed (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {6, 11, 5, 14, 15, C1}
17. P believes that S once conveyed (Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {16, C9}
18. P believes that S once conveyed (K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {17, C9}
19. P believes that S once conveyed K. {18, C9}

Proof for Q believes that S once conveyed K:

1. Q was told (*E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q), Np1). {Step}
2. Q possesses (Nq, Nq1, Kqs). {Assumption}
3. Q believes that Nq1 is fresh. {Assumption}
4. Q believes that Nq1 is recognizable. {Assumption}
5. Q believes that Kqs is a suitable secret for use between Q and S. {Assumption}
6. Q was told *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {1, T4}
7. Q possesses (Nq1, Kqs). {2, P4}
8. Q was told *E(Kqs : K, Nq1, P). {6, T3}
9. Q possesses Kqs. {7, P4}
10. Q was told E(Kqs : K, Nq1, P). {8, T2}
11. Q was told (K, Nq1, P). {10, 9, T6}
12. Q possesses (K, Nq1, P). {11, P1}
13. Q believes that (K, Nq1, P) is recognizable. {4, 12, R3'}
14. Q believes that (K, Nq1, P) is fresh. {3, 12, F3'}
15. Q believes that S once conveyed (K, Nq1, P). {8, 9, 5, 13, 14, C1}
16. Q believes that S once conveyed K. {15, C9}

Proof for P believes that Q believes that K is a suitable secret for use between P and Q:

1. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) -> (K is a suitable secret for use between P and Q). {Step}
2. P was told *E(K : Nq, Np1) -> (K is a suitable secret for use between P and Q). {Step}
3. P possesses (P, Q, Np, Np1, Kps). {Assumption}
4. P believes that Np1 is fresh. {Assumption}
5. P believes that Np1 is recognizable. {Assumption}
6. P believes that Np is fresh. {Assumption}
7. P believes that Np is recognizable. {Assumption}
8. P believes that Kps is a suitable secret for use between P and S. {Assumption}
9. P believes that S has jurisdiction over the statement "K is a suitable secret for use between P and Q". {Assumption}
10. P believes that S is trustworthy. {Assumption}
11. P believes that Q is trustworthy. {Assumption}
12. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {1, T3}
13. P was told *E(K : Nq, Np1). {2, T3}
14. P possesses (Q, Np, Np1, Kps). {3, P4}

15. P was told E(K : Nq, Np1). {13, T2}
16. P was told E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {12, T2}
17. P possesses E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {16, P1}
18. P possesses E(K : Nq, Np1). {15, P1}
19. P possesses (Np, Np1, Kps). {14, P4}
20. P possesses (Np1, Kps). {19, P4}
21. P possesses Kps. {20, P4}
22. P was told (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {16, 21, T6}
23. P possesses (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {22, P1}
24. P possesses (Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {23, P4}
25. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is recognizable. {7, 23, R3'}
26. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is fresh. {6, 23, F3'}
27. P believes that E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is fresh. {26, 21, 17, F5}
28. P believes that S once conveyed E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) -> (K is a suitable secret for use between P and Q). {1, 21, 8, 25, 26, C1}
29. P believes that S believes that K is a suitable secret for use between P and Q. {10, 28, 27, J2}
30. P possesses (K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {24, P4}
31. P believes that K is a suitable secret for use between P and Q. {9, 29, J1}
32. P possesses K. {30, P4}
33. P was told (Nq, Np1). {15, 32, T6}
34. P possesses (Nq, Np1). {33, P1}
35. P believes that (Nq, Np1) is recognizable. {5, 34, R3'}
36. P believes that (Nq, Np1) is fresh. {4, 34, F3'}
37. P believes that E(K : Nq, Np1) is fresh. {36, 32, 18, F5}
38. P believes that Q once conveyed E(K : Nq, Np1) -> (K is a suitable secret for use between P and Q). {2, 32, 31, 35, 36, C1}
39. P believes that Q believes that K is a suitable secret for use between P and Q. {11, 38, 37, J2}

Proof for Q believes that P believes that K is a suitable secret for use between P and Q:

1. Q was told (*E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q), Np1). {Step}
2. Q was told *E(K : F(Nq)) -> (K is a suitable secret for use between P and Q). {Step}
3. Q possesses (Nq, Nq1, Kqs). {Assumption}
4. Q believes that Nq1 is fresh. {Assumption}
5. Q believes that Nq1 is recognizable. {Assumption}
6. Q believes that Nq is fresh. {Assumption}
7. Q believes that Nq is recognizable. {Assumption}
8. Q believes that Kqs is a suitable secret for use between Q and S. {Assumption}
9. Q believes that S has jurisdiction over the statement "K is a suitable secret for use between P and Q". {Assumption}
10. Q believes that S is trustworthy. {Assumption}
11. Q believes that P is trustworthy. {Assumption}

12. Q was told *E(K : F(Nq)). {2, T3}
13. Q was told *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {1, T4}
14. Q possesses (Nq1, Kqs). {3, P4}
15. Q was told E(K : F(Nq)). {12, T2}
16. Q was told *E(Kqs : K, Nq1, P). {13, T3}
17. Q possesses E(K : F(Nq)). {15, P1}
18. Q possesses Kqs. {14, P4}
19. Q was told E(Kqs : K, Nq1, P). {16, T2}
20. Q was told (K, Nq1, P). {19, 18, T6}
21. Q possesses (K, Nq1, P). {20, P1}
22. Q possesses E(Kqs : K, Nq1, P). {19, P1}
23. Q possesses K. {21, P4}
24. Q believes that (K, Nq1, P) is recognizable. {5, 21, R3'}
25. Q believes that (K, Nq1, P) is fresh. {4, 21, F3'}
26. Q believes that E(Kqs : K, Nq1, P) is fresh. {25, 18, 22, F5}
27. Q believes that S once conveyed E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {13, 18, 8, 24, 25, C1}
28. Q believes that S believes that K is a suitable secret for use between P and Q. {10, 27, 26, J2}
29. Q was told F(Nq). {15, 23, T6}
30. Q possesses F(Nq). {29, P1}
31. Q believes that F(Nq) is recognizable. {7, 30, R3''}
32. Q believes that F(Nq) is fresh. {6, 30, F3''}
33. Q believes that E(K : F(Nq)) is fresh. {32, 23, 17, F5}
34. Q believes that K is a suitable secret for use between P and Q. {9, 28, J1}
35. Q believes that K is a suitable secret for use between Q and P. {34, C20}
36. Q believes that P once conveyed E(K : F(Nq)) -> (K is a suitable secret for use between P and Q). {2, 23, 35, 31, 32, C1}
37. Q believes that P believes that K is a suitable secret for use between P and Q. {11, 36, 33, J2}

Proof for P believes that Q possesses K:

1. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) -> (K is a suitable secret for use between P and Q). {Step}
2. P was told *E(K : Nq, Np1) -> (K is a suitable secret for use between P and Q). {Step}
3. P possesses (P, Q, Np, Np1, Kps). {Assumption}
4. P believes that Np1 is fresh. {Assumption}
5. P believes that Np1 is recognizable. {Assumption}
6. P believes that Np is fresh. {Assumption}
7. P believes that Np is recognizable. {Assumption}
8. P believes that Kps is a suitable secret for use between P and S. {Assumption}
9. P believes that S has jurisdiction over the statement "K is a suitable secret for use between P and Q". {Assumption}
10. P believes that S is trustworthy. {Assumption}
11. P was told *E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {1, T3}
12. P was told *E(K : Nq, Np1). {2, T3}
13. P possesses (Q, Np, Np1, Kps). {3, P4}
14. P was told E(K : Nq, Np1). {12, T2}
15. P was told E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {11, T2}
16. P possesses E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for

use between P and Q)). {15, P1}
17. P possesses (Np, Np1, Kps). {13, P4}
18. P possesses (Np1, Kps). {17, P4}
19. P possesses Kps. {18, P4}
20. P was told (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {15, 19, T6}
21. P possesses (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {20, P1}
22. P possesses (Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {21, P4}
23. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is recognizable. {7, 21, R3'}
24. P believes that (Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is fresh. {6, 21, F3'}
25. P believes that E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) is fresh. {24, 19, 16, F5}
26. P believes that S once conveyed E(Kps : Np, Q, K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)) -> (K is a suitable secret for use between P and Q). {1, 19, 8, 23, 24, C1}
27. P believes that S believes that K is a suitable secret for use between P and Q. {10, 26, 25, J2}
28. P possesses (K, *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q)). {22, P4}
29. P believes that K is a suitable secret for use between P and Q. {9, 27, J1}
30. P possesses K. {28, P4}
31. P was told (Nq, Np1). {14, 30, T6}
32. P possesses (Nq, Np1). {31, P1}
33. P believes that (Nq, Np1) is recognizable. {5, 32, R3'}
34. P believes that (Nq, Np1) is fresh. {4, 32, F3'}
35. P believes that Q possesses ((Nq, Np1), K). {12, 30, 29, 33, 34, C1}
36. P believes that Q possesses K. {35, C19}

Proof for Q believes that P possesses K:

1. Q was told (*E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q), Np1). {Step}
2. Q was told *E(K : F(Nq)) -> (K is a suitable secret for use between P and Q). {Step}
3. Q possesses (Nq, Nq1, Kqs). {Assumption}
4. Q believes that Nq1 is fresh. {Assumption}
5. Q believes that Nq1 is recognizable. {Assumption}
6. Q believes that Nq is fresh. {Assumption}
7. Q believes that Nq is recognizable. {Assumption}
8. Q believes that Kqs is a suitable secret for use between Q and S. {Assumption}
9. Q believes that S has jurisdiction over the statement "K is a suitable secret for use between P and Q". {Assumption}
10. Q believes that S is trustworthy. {Assumption}
11. Q was told *E(K : F(Nq)). {2, T3}
12. Q was told *E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {1, T4}
13. Q possesses (Nq1, Kqs). {3, P4}
14. Q was told E(K : F(Nq)). {11, T2}
15. Q was told *E(Kqs : K, Nq1, P). {12, T3}
16. Q possesses Kqs. {13, P4}
17. Q was told E(Kqs : K, Nq1, P). {15, T2}
18. Q was told (K, Nq1, P). {17, 16, T6}

19. Q possesses (K, Nq1, P). {18, P1}
20. Q possesses E(Kqs : K, Nq1, P). {17, P1}
21. Q possesses K. {19, P4}
22. Q believes that (K, Nq1, P) is recognizable. {5, 19, R3'}
23. Q believes that (K, Nq1, P) is fresh. {4, 19, F3'}
24. Q believes that E(Kqs : K, Nq1, P) is fresh. {23, 16, 20, F5}
25. Q believes that S once conveyed E(Kqs : K, Nq1, P) -> (K is a suitable secret for use between P and Q). {12, 16, 8, 22, 23, C1}
26. Q believes that S believes that K is a suitable secret for use between P and Q. {10, 25, 24, J2}
27. Q was told F(Nq). {14, 21, T6}
28. Q possesses F(Nq). {27, P1}
29. Q believes that F(Nq) is recognizable. {7, 28, R3''}
30. Q believes that F(Nq) is fresh. {6, 28, F3''}
31. Q believes that K is a suitable secret for use between P and Q. {9, 26, J1}
32. Q believes that K is a suitable secret for use between Q and P. {31, C20}
33. Q believes that P possesses (F(Nq), K). {11, 21, 32, 29, 30, C1}
34. Q believes that P possesses K. {33, C19}

Appendix C

GYPSIE Specification Experiment

The questionnaire in this appendix was used to obtain an indication of how protocol engineers interact with the GYPSIE design environment. Twenty individuals were used in this experiment and it was conducted over the period 21 to 25 May 2001.

Section 1

Specify the following protocols in SPEAR II and then save the resultant protocol definition. Try to be as accurate as possible while constructing the specification, but also note that the time taken to create each model will be timed and used to compute your overall result.

Voting Protocol

(1) **Q** $\longrightarrow$ **Pi**: N_{q}
(2) **Pi** $\longrightarrow$ **Q**: Pi, N_{i}, vi, $H(N_{\mathrm{q}}, < S_{\mathrm{i}} >, \mathrm{vi})$
(3) **Q** $\longrightarrow$ **Pi**: result, $H(N_{\mathrm{i}}, < S_{\mathrm{i}} >, \mathrm{result})$

Authentication Protocol

(1) **A** $\longrightarrow$ **B**: A, N_{a}
(2) **B** $\longrightarrow$ **A**: $\{\mathrm{B}, \{N_{\mathrm{a}}\}_{-K_{\mathrm{b}}}, K_{\mathrm{ab}}, N_{\mathrm{b}}\}_{+K_{\mathrm{a}}}, \{N_{\mathrm{a}}\}_{K_{\mathrm{ab}}}$
(3) **A** $\longrightarrow$ **B**: $\{N_{\mathrm{b}}\}_{K_{\mathrm{ab}}}$

Needham-Schroeder Protocol

(1) **P** $\longrightarrow$ **Q**: P
(2) **Q** $\longrightarrow$ **P**: $\{\mathrm{P}, N_{\mathrm{q1}}\}_{K_{\mathrm{qs}}}$
(3) **P** $\longrightarrow$ **S**: P, Q, N_{p}, $\{\mathrm{P}, N_{\mathrm{q1}}\}_{K_{\mathrm{qs}}}$
(4) **S** $\longrightarrow$ **P**: $\{N_{\mathrm{p}}, \mathrm{Q}, K_{\mathrm{pq}}, \{K_{\mathrm{pq}}, N_{\mathrm{q1}}, \mathrm{P}\}_{K_{\mathrm{qs}}}\}_{K_{\mathrm{ps}}}$
(5) **P** $\longrightarrow$ **Q**: N_{p1}, $\{K_{\mathrm{pq}}, N_{\mathrm{q1}}, \mathrm{P}\}_{K_{\mathrm{qs}}}$
(6) **Q** $\longrightarrow$ **P**: $\{N_{\mathrm{q}}, N_{\mathrm{p1}}\}_{K_{\mathrm{pq}}}$
(7) **P** $\longrightarrow$ **Q**: $\{F(N_{\mathrm{q}})\}_{K_{\mathrm{pq}}}$

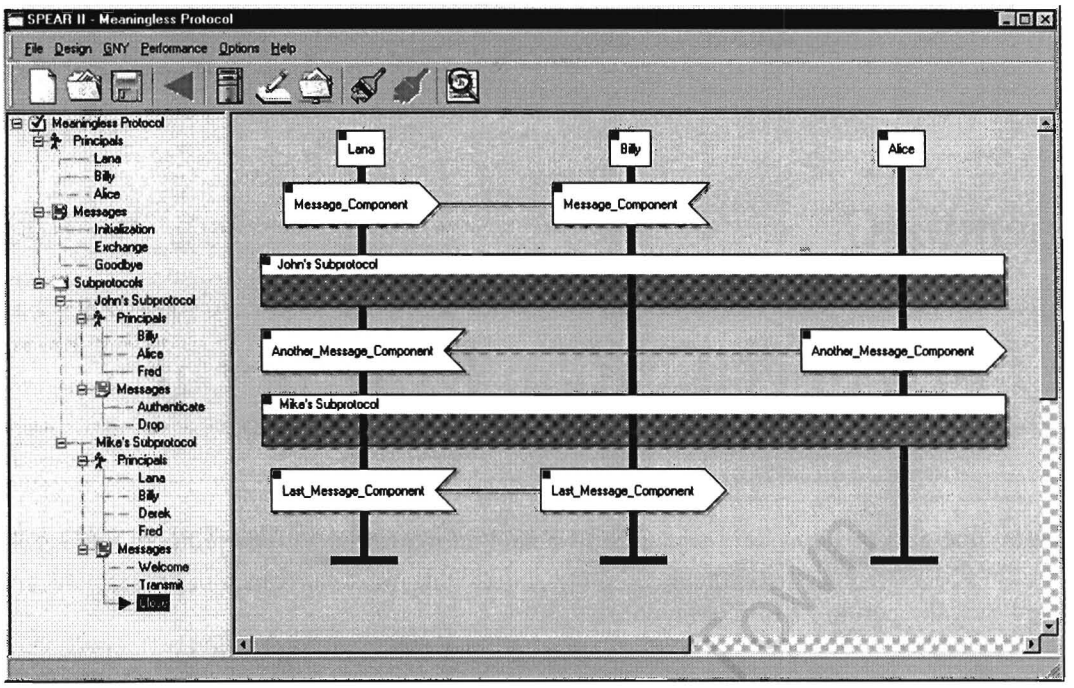


Figure C.1: A SPEAR II screenshot.

Section 2

Examine the protocol model specified in the screenshot displayed in Figure C.1 and then answer the questions that follow.

1. What is the name of this protocol specification?

   _____

2. How many subprotocols does the specification contain?

   _____

3. How many messages does the specification contain? Don't count the messages transmitted in subprotocols.

   _____

4. List all of the principals who send or receive messages in the specification. Exclude the principals defined in any subprotocols from this answer.

   _____

5. List the principals defined in *Mike's Subprotocol.*

   _____

6. (a) Which principals do *John's Subprotocol* and its parent protocol have in common?

   (b) How did you ascertain this fact?

7. How many messages are defined in *Mike's Subprotocol*?

Section 3

Open the *Kerberos* protocol specification file and then carrying out the following operations:

1. Use the Component Tracker to highlight the ticket $\{T_s, L, K_{ab}, A\}_{K_{bs}}$.
2. Swop the positions of the first and second message.
3. Make S the sender of the fourth message.
4. Make S the receiver of the third message.
5. Rename principal S to AS.

Appendix D

Visual GNY Comprehension Experiment

The questionnaire in this appendix was used to examine whether Visual GNY aids in the effective construction of syntactically and semantically correct GNY expressions. Twenty individuals were used in this experiment and it was conducted over the period 14 to 17 November 2000.

Section 1

Translate the following English statements into both VGNY and GNY. Write the GNY statements on this answer sheet, and then compile the VGNY statements the using SPEAR II.

1. A believes that N_a is fresh.

   _____

2. A believes that N_b is recognizable.

   _____

3. A believes that C has jurisdiction over the statement "K_{ac} is a suitable secret for use between A and C".

   _____

4. B believes that C is trustworthy.

   _____

5. A believes that B once conveyed (N_a, $Login\_Name$).

   _____

6. A believes that B possesses ($Data$, $Password$).

   _____

7. C believes that $+K_b$ is a suitable public key for B.

   _____

8. B believes that A is told $(N_b, Server\_Name)$.

9. A believes that B believes that K_{ab} is a suitable secret for use between A and B.

10. A believes that B is not the first principal to originate N_a.

11. A possesses $Data$.

12. B possesses $(N_b, Password)$.

13. A possesses $E(K_a : T_a, Data)$.

Section 2

Translate the following GNY statements into English. Write the solutions on this answer sheet.

1. $A \mid\equiv B \lhd (N_a, Data)$

2. $B \mid\equiv C \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$

3. $A \mid\equiv B \mid\Longrightarrow B \mid\equiv *$

4. $B \mid\equiv A \mid\sim E(K : Data, N_a)$

5. $A \mid\equiv B \dashv (N_a)$

6. $A \ni K$

7. $C \mid\equiv \overset{+K_s}{\longmapsto} A$

8. $B \mid\equiv \phi(N_b)$

9. $C \mid\equiv D \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

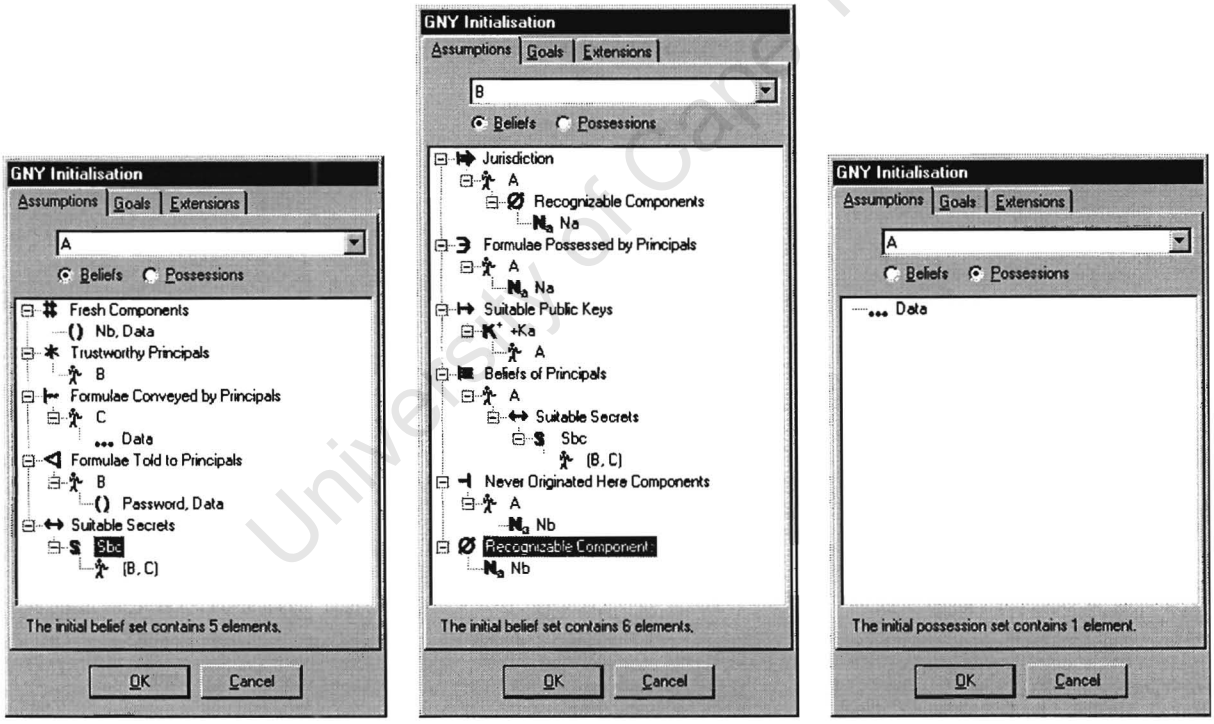
10. $A \mid\equiv \sharp(N_a)$

11. $C \mid\equiv A \ni (Data, Password)$

12. $A \mid\equiv B \overset{K_{bc}}{\longleftrightarrow} C$

Section 3

Translate the following VGNY statement trees into English. Start with the left pane, and then work downwards through each tree.



1. _____

2. _____

3. _____

4. _____

5. _____

6. _____

7. _____

8. _____

9. _____

10. _____

11. _____

12. _____

Appendix E

Further GNY Analyses with SPEAR II

The GNY analysis results in this appendix were all obtained from analyses that were conducted using the Visual GNY and GYNGER components of the SPEAR II framework. The analyses in Sections E.1 through to E.6 are based on the BAN analyses conducted in [1], while the last two analyses are based on protocols described in [35]. Due to a lack of space, we have omitted the proof for each of the successful goals. However, to aid in verifying these results, the source file for each of these analyses has been included with the SPEAR II application, available from *http://www.cs.uct.ac.za/Research/DNA/SPEAR2*.

E.1 Wide-Mouthed Frog

Idealized Protocol Steps

$A \longrightarrow S:\ \mathrm{A},\ *\{T_\mathrm{a},\ \mathrm{B},\ K_\mathrm{ab}\}_{K_\mathrm{as}} \rightsquigarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$S \longrightarrow B:\ *\{T_\mathrm{s},\ \mathrm{A},\ K_\mathrm{ab}\}_{K_\mathrm{bs}} \rightsquigarrow A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

Initial Possessions

$A \ni (K_\mathrm{as}, \mathrm{A}, T_\mathrm{a}, \mathrm{B}, K_\mathrm{ab})$ $\quad B \ni K_\mathrm{bs}$ $\quad S \ni (K_\mathrm{bs}, T_\mathrm{s}, K_\mathrm{as})$

Initial Beliefs

$A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$B \mid\equiv \sharp(T_\mathrm{s})$
$B \mid\equiv \phi(\mathrm{A})$
$B \mid\equiv S \mid\Longrightarrow S \mid\equiv *$

$B \mid\equiv B \overset{K_\mathrm{bs}}{\longleftrightarrow} S$
$B \mid\equiv S \mid\Longrightarrow A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \mid\Longrightarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$S \mid\equiv \sharp(T_\mathrm{a})$

$S \mid\equiv \phi(\mathrm{B})$
$S \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$S \mid\equiv A \overset{K_\mathrm{as}}{\longleftrightarrow} S$

Goals Achieved

$A \ni K_\mathrm{ab}$
$B \ni K_\mathrm{ab}$
$S \mid\equiv A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

$A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$B \mid\equiv S \mid\equiv A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

$B \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

E.2 Kerberos

Idealized Protocol Steps

$A \longrightarrow S:\ \mathrm{A},\ \mathrm{B}$
$S \longrightarrow A:\ *\{T_\mathrm{s},\ \mathrm{L},\ K_\mathrm{ab},\ \mathrm{B},\ *\{T_\mathrm{s},\ \mathrm{L},\ K_\mathrm{ab},\ \mathrm{A}\}_{K_\mathrm{bs}} \rightsquigarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B\}_{K_\mathrm{as}} \rightsquigarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$A \longrightarrow B:\ *\{T_\mathrm{s},\ \mathrm{L},\ K_\mathrm{ab},\ \mathrm{A}\}_{K_\mathrm{bs}} \rightsquigarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B,\ *\{\mathrm{A},\ T_\mathrm{a}\}_{K_\mathrm{ab}} \rightsquigarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$B \longrightarrow A:\ *\{inc(T_\mathrm{a})\}_{K_\mathrm{ab}} \rightsquigarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

Initial Possessions

$S \ni (K_\mathrm{as}, K_\mathrm{bs}, T_\mathrm{s}, \mathrm{L}, K_\mathrm{ab})$ $A \ni (K_\mathrm{as}, T_\mathrm{a}, \mathrm{A}, \mathrm{B})$ $B \ni K_\mathrm{bs}$

Initial Beliefs

$A \mid\equiv A \overset{K_\mathrm{as}}{\longleftrightarrow} S$
$A \mid\equiv B \mid\Longrightarrow B \mid\equiv *$
$A \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$A \mid\equiv S \mid\Longrightarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$A \mid\equiv \sharp(T_\mathrm{a})$
$A \mid\equiv \sharp(T_\mathrm{s})$

$A \mid\equiv \phi(T_\mathrm{s})$
$A \mid\equiv \phi(T_\mathrm{a})$
$B \mid\equiv B \overset{K_\mathrm{bs}}{\longleftrightarrow} S$
$B \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

$B \mid\equiv \sharp(T_\mathrm{a})$
$B \mid\equiv \sharp(T_\mathrm{s})$
$B \mid\equiv \phi(T_\mathrm{a})$
$B \mid\equiv \phi(T_\mathrm{s})$

Goals Achieved

$A \ni K_\mathrm{ab}$
$B \ni K_\mathrm{ab}$
$A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

$A \mid\equiv B \ni K_\mathrm{ab}$
$A \mid\equiv B \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

$B \mid\equiv A \ni K_\mathrm{ab}$
$B \mid\equiv A \mid\equiv A \overset{K_\mathrm{ab}}{\longleftrightarrow} B$

E.3 Needham-Schroeder Public-Key

Idealized Protocol Steps

$A \longrightarrow S:\ \mathrm{A},\ \mathrm{B}$
$S \longrightarrow A:\ *\{+K_\mathrm{b},\ \mathrm{B},\ T_{\mathrm{s}1}\}_{-K_\mathrm{s}} \rightsquigarrow\!\!\!\!\mapsto^{+K_\mathrm{b}} B$
$A \longrightarrow B:\ \{*\{S_\mathrm{a},\ T_\mathrm{a}\}_{-K_\mathrm{a}} \rightsquigarrow A \overset{S_\mathrm{a}}{\longleftrightarrow} B,\ \mathrm{A}\}_{+K_\mathrm{b}}$
$B \longrightarrow S:\ \mathrm{B},\ \mathrm{A}$
$S \longrightarrow B:\ *\{+K_\mathrm{a},\ \mathrm{A},\ T_{\mathrm{s}2}\}_{-K_\mathrm{s}} \rightsquigarrow\!\!\!\!\mapsto^{+K_\mathrm{a}} A$
$B \longrightarrow A:\ *\{< S_\mathrm{a} >,\ S_\mathrm{b}\}_{+K_\mathrm{a}} \rightsquigarrow (A \overset{S_\mathrm{b}}{\longleftrightarrow} B,\ A \overset{S_\mathrm{a}}{\longleftrightarrow} B)$
$A \longrightarrow B:\ *\{< S_\mathrm{b} >\}_{+K_\mathrm{b}} \rightsquigarrow A \overset{S_\mathrm{b}}{\longleftrightarrow} B$

Initial Possessions

$B \ni (+K_s, -K_b)$ $A \ni (+K_s, -K_a, \mathrm{B}, \mathrm{A})$

Initial Beliefs

$B \mid\equiv \sharp(S_b)$
$B \mid\equiv \sharp(T_a)$
$B \mid\equiv \sharp(T_{s2})$
$B \mid\equiv \phi(S_b)$
$B \mid\equiv \phi(\mathrm{A})$
$B \mid\equiv \phi(T_a)$
$B \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow S \mid\equiv *$

$B \mid\equiv \overset{+K_s}{\longmapsto} S$
$B \mid\equiv \overset{+K_b}{\longmapsto} B$
$B \mid\equiv S \mid\Longrightarrow \overset{+K_a}{\longmapsto} A$
$B \mid\equiv A \mid\Longrightarrow A \overset{S_a}{\longleftrightarrow} B$
$B \mid\equiv A \overset{S_b}{\longleftrightarrow} B$
$A \mid\equiv \sharp(S_a)$
$A \mid\equiv \sharp(T_{s1})$
$A \mid\equiv \phi(S_a)$

$A \mid\equiv \phi(\mathrm{B})$
$A \mid\equiv \overset{+K_s}{\longmapsto} S$
$A \mid\equiv \overset{+K_a}{\longmapsto} A$
$A \mid\equiv B \mid\Longrightarrow B \mid\equiv *$
$A \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$A \mid\equiv S \mid\Longrightarrow \overset{+K_b}{\longmapsto} B$
$A \mid\equiv B \mid\Longrightarrow A \overset{S_b}{\longleftrightarrow} B$
$A \mid\equiv A \overset{S_a}{\longleftrightarrow} B$

Goals Achieved

$B \ni S_a$
$B \ni +K_a$
$A \ni S_b$
$A \ni +K_b$

$B \mid\equiv \overset{+K_a}{\longmapsto} A$
$B \mid\equiv A \mid\equiv A \overset{S_a}{\longleftrightarrow} B$
$B \mid\equiv A \mid\equiv A \overset{S_b}{\longleftrightarrow} B$
$B \mid\equiv A \overset{S_a}{\longleftrightarrow} B$

$A \mid\equiv \overset{+K_b}{\longmapsto} B$
$A \mid\equiv B \mid\equiv A \overset{S_b}{\longleftrightarrow} B$
$A \mid\equiv B \mid\equiv A \overset{S_a}{\longleftrightarrow} B$
$A \mid\equiv A \overset{S_b}{\longleftrightarrow} B$

E.4 Yahalom

Idealized Protocol Steps

$A \longrightarrow B:\ \mathrm{A},\ N_a$
$B \longrightarrow S:\ \mathrm{B},\ *\{\mathrm{A},\ N_a,\ N_b\}_{K_{bs}}$
$S \longrightarrow A:\ *\{\mathrm{B},\ K_{ab},\ N_a,\ N_b\}_{K_{as}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B,\ *\{\mathrm{A},\ K_{ab}\}_{K_{bs}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$A \longrightarrow B:\ *\{\mathrm{A},\ K_{ab},\ N_b\}_{K_{bs}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B,\ *\{N_b\}_{K_{ab}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$

Initial Possessions

$S \ni (K_{bs}, K_{as}, K_{ab})$ $A \ni (K_{as}, \mathrm{A}, N_a)$ $B \ni (K_{bs}, \mathrm{B}, N_b)$

Initial Beliefs

$S \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$S \mid\equiv A \overset{K_{as}}{\longleftrightarrow} S$
$S \mid\equiv B \overset{K_{bs}}{\longleftrightarrow} S$
$A \mid\equiv \phi(N_a)$
$A \mid\equiv \sharp(N_a)$

$A \mid\equiv A \overset{K_{as}}{\longleftrightarrow} S$
$A \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$A \mid\equiv S \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv \phi(N_b)$
$B \mid\equiv \sharp(N_b)$

$B \mid\equiv B \overset{K_{bs}}{\longleftrightarrow} S$
$B \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$

Goals Achieved

$A \ni K_{ab}$
$B \ni K_{ab}$
$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \ni K_{ab}$

E.5 Otway-Rees

Idealized Protocol Steps

$A \longrightarrow B:\ N_c,\ *\{N_a,\ N_c\}_{K_{as}}$
$B \longrightarrow S:\ N_c,\ *\{N_a,\ N_c\}_{K_{as}},\ *\{N_b,\ N_c\}_{K_{bs}}$
$S \longrightarrow B:\ \text{RunID},\ *\{N_a,\ K_{ab}\}_{K_{as}} \rightsquigarrow (A \overset{K_{ab}}{\longleftrightarrow} B,\ B \mid\sim N_c),\ *\{N_b,\ K_{ab}\}_{K_{bs}} \rightsquigarrow (A \overset{K_{ab}}{\longleftrightarrow} B,\ A \mid\sim N_c)$
$B \longrightarrow A:\ \text{RunID},\ *\{N_a,\ K_{ab}\}_{K_{as}} \rightsquigarrow (A \overset{K_{ab}}{\longleftrightarrow} B,\ B \mid\sim N_c)$
where $N_c = \text{RunID},\ \text{A},\ \text{B}$

Initial Possessions

$S \ni (K_{as}, K_{bs}, K_{ab})$
$B \ni (K_{bs}, N_b)$
$A \ni (K_{as}, N_a, N_c)$

Initial Beliefs

$B \mid\equiv \sharp(N_b)$
$B \mid\equiv \phi(N_b)$
$B \mid\equiv S \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv S \mid\Longrightarrow A \mid\sim N_c$
$B \mid\equiv B \overset{K_{bs}}{\longleftrightarrow} S$
$B \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$A \mid\equiv \sharp(N_c)$
$A \mid\equiv \sharp(N_a)$
$A \mid\equiv \phi(N_a)$
$A \mid\equiv S \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$A \mid\equiv S \mid\Longrightarrow B \mid\sim N_c$
$A \mid\equiv A \overset{K_{as}}{\longleftrightarrow} S$
$A \mid\equiv S \mid\Longrightarrow S \mid\equiv *$

Achieved Goals

$S \ni N_a$
$S \ni N_b$
$B \ni K_{ab}$
$A \ni K_{ab}$
$B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv S \mid\equiv A \mid\sim N_c$
$B \mid\equiv A \mid\sim N_c$
$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$A \mid\equiv S \mid\equiv B \mid\sim N_c$
$A \mid\equiv B \mid\sim N_c$
$A \mid\equiv B \ni N_c$

E.6 Andrew Secure RPC Handshake

Idealized Protocol Steps

$A \longrightarrow B:\ \text{A},\ N_a$
$B \longrightarrow A:\ *\{N_a,\ K'_{ab},\ N_b\}_{K_{ab}} \rightsquigarrow A \overset{K'_{ab}}{\longleftrightarrow} B$
$A \longrightarrow B:\ *\{N_b\}_{K'_{ab}} \rightsquigarrow A \overset{K'_{ab}}{\longleftrightarrow} B$
$B \longrightarrow A:\ N'_b$

Initial Possessions

$B \ni (K_{ab}, K'_{ab}, N'_b)$ $A \ni (K_{ab}, N_a)$

Initial Beliefs

$B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \overset{K'_{ab}}{\longleftrightarrow} B$
$B \mid\equiv \sharp(N_b)$
$B \mid\equiv \sharp(N'_b)$
$B \mid\equiv \phi(N_b)$
$B \mid\equiv \phi(N'_b)$
$B \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$A \mid\equiv \sharp(N_a)$
$A \mid\equiv \phi(N_a)$
$A \mid\equiv B \mid\Longrightarrow A \overset{K'_{ab}}{\longleftrightarrow} B$
$A \mid\equiv B \mid\Longrightarrow B \mid\equiv *$

Achieved Goals

$A \ni K'_{ab}$
$B \ni K'_{ab}$
$A \mid\equiv A \overset{K'_{ab}}{\longleftrightarrow} B$
$A \mid\equiv B \mid\equiv A \overset{K'_{ab}}{\longleftrightarrow} B$
$A \mid\equiv B \ni K'_{ab}$
$B \mid\equiv A \overset{K'_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \mid\equiv A \overset{K'_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \ni K'_{ab}$

E.7 Gong Rounds Paper Case 2

Idealized Protocol Steps

$A \longrightarrow S$: A, B
$S \longrightarrow A$: $*\{\text{S, A, A, } K_{ab}\text{, B, } T_s\}_{K_{as}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$, $*\{\text{S, B, A, } K_{ab}\text{, B, } T_s\}_{K_{bs}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$A \longrightarrow B$: $*\{\text{S, B, A, } K_{ab}\text{, B, } T_s\}_{K_{bs}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$, $*\{\text{A, B, } T_a\}_{K_{ab}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$B \longrightarrow A$: $*\{\text{B, A, } T_b\}_{K_{ab}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$

Initial Possessions:

$S \ni (K_{bs}, K_{as}, T_s, \text{S}, K_{ab})$ $B \ni (K_{bs}, T_b)$ $A \ni (K_{as}, T_a, \text{B}, \text{A})$

Initial Beliefs:

$S \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$S \mid\equiv A \overset{K_{as}}{\longleftrightarrow} S$
$S \mid\equiv B \overset{K_{bs}}{\longleftrightarrow} S$
$B \mid\equiv \phi(\text{B})$
$B \mid\equiv \sharp(T_a)$
$B \mid\equiv \sharp(T_s)$
$B \mid\equiv B \overset{K_{bs}}{\longleftrightarrow} S$
$B \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$A \mid\equiv \phi(\text{A})$
$A \mid\equiv \sharp(T_b)$
$A \mid\equiv \sharp(T_s)$
$A \mid\equiv A \overset{K_{as}}{\longleftrightarrow} S$
$A \mid\equiv B \mid\Longrightarrow B \mid\equiv *$
$A \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$A \mid\equiv S \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$

Achieved Goals

$A \ni K_{ab}$
$B \ni K_{ab}$
$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

$A \mid\equiv B \ni K_{ab}$
$A \mid\equiv B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

$B \mid\equiv A \ni K_{ab}$
$B \mid\equiv A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

E.8 Gong Rounds Paper Case 4

Idealized Protocol Steps

$A \longrightarrow S:\ \mathrm{A},\ \mathrm{B},\ *\{\mathrm{B},\ K_{ab},\ T_a\}_{K_{as}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B,\ *\{\mathrm{A},\ T_a\}_{K_{ab}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$S \longrightarrow B:\ \mathrm{A},\ \mathrm{B},\ *\{\mathrm{A},\ K_{ab},\ T_s\}_{K_{bs}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B,\ *\{\mathrm{A},\ T_a\}_{K_{ab}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$
$B \longrightarrow A:\ *\{\mathrm{B},\ T_b\}_{K_{ab}} \rightsquigarrow A \overset{K_{ab}}{\longleftrightarrow} B$

Initial Possessions

$S \ni (K_{as}, K_{bs}, T_s)$ $A \ni (K_{ab}, K_{as}, T_a, \mathrm{B}, \mathrm{A})$ $B \ni (T_b, K_{bs})$

Initial Beliefs

$S \mid\equiv \phi(\mathrm{B})$
$S \mid\equiv \sharp(T_a)$
$S \mid\equiv A \overset{K_{as}}{\longleftrightarrow} S$
$S \mid\equiv B \overset{K_{bs}}{\longleftrightarrow} S$
$S \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$S \mid\equiv A \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$

$A \mid\equiv \phi(\mathrm{B})$
$A \mid\equiv \sharp(T_b)$
$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$A \mid\equiv A \overset{K_{as}}{\longleftrightarrow} S$
$A \mid\equiv B \mid\Longrightarrow B \mid\equiv *$
$B \mid\equiv \phi(\mathrm{A})$

$B \mid\equiv \sharp(T_s)$
$B \mid\equiv \sharp(T_a)$
$B \mid\equiv B \overset{K_{bs}}{\longleftrightarrow} S$
$B \mid\equiv A \mid\Longrightarrow A \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow S \mid\equiv *$
$B \mid\equiv S \mid\Longrightarrow A \overset{K_{ab}}{\longleftrightarrow} B$

Achieved Goals

$A \ni K_{ab}$
$B \ni K_{ab}$
$A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

$A \mid\equiv B \ni K_{ab}$
$A \mid\equiv B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$
$B \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

$B \mid\equiv A \ni K_{ab}$
$B \mid\equiv A \mid\equiv A \overset{K_{ab}}{\longleftrightarrow} B$

Bibliography

[1] M. Abadi, M. Burrows, and R. Needham. A Logic of Authentication. In *Proceedings of the Royal Society*, Series A, 426, 1871, pages 233 – 271, December 1989.

[2] M. Abadi and R. Needham. Prudent Engineering Practice for Cryptographic Protocols. *IEEE Transactions on Software Engineering*, 22(1):6 – 15, January 1996.

[3] M. Abadi and M. Tuttle. A Semantics for a Logic of Authentication. In *Proceedings of the Tenth ACM Symposium on Principles of Distributed Computing*, pages 201 – 216. ACM Press, August 1991.

[4] A.V. Aho, R. Sethi, and J.D. Ullman. *Compilers: Principles, Techniques and Tools*. Addison-Wesley, 1986.

[5] J.P. Bekmann, P. De Goede, and A.C.M. Hutchison. SPEAR: Security Protocol Engineering and Analysis Resources. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*. Rutgers University, September 1997.

[6] J.P. Bekmann, P. De Goede, and A.C.M. Hutchison. Concurrency and Synchronisation Issues in Security Implementations. In *Proceedings of the 14th International Information Security Conference (IFIP/SEC'98)*, pages 333 – 343, Vienna, Austria and Budapest, Hungary, 1998.

[7] S. Bellman, G. Lohse, and E. Johnson. Predictors of Online Buying Behaviour. *Communications of the ACM*, 42(12):32 – 38, December 1999.

[8] M.S. Berry, A.C.M. Hutchison, and E. Saul. Predicting the Performance of Transactional Electronic Commerce Protocols. In *Proceedings of the Seventh Annual Working Conference on Information Security Management and Small Systems Security*, pages 161 – 175, Amsterdam, Netherlands, September 1999. Kluwer Academic Publishers.

[9] D. Bolignano. An Approach to the Formal Verification of Cryptographic Protocols. In *Proceedings of the Third ACM Conference on Computer and Communications Security*, pages 106 – 118. ACM Press, 1996.

[10] T. Bolognesi and E. Brinksma. Introduction to the ISO Specification Language LOTOS. *Computer Networks and ISDN Systems*, 14(1):25 – 59, 1987.

[11] M. Booyens and P.S. Kritzinger. SNAPL/1: A Language to Describe and Evaluate Queuing Network Models. *Performance Evaluation*, 4(3):171 – 181, August 1984.

[12] C. Boyd. Extensional Goals in Authentication Protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*. Rutgers University, September 1997.

[13] S.H. Brackin. A HOL Extension of GNY for Automatically Analyzing Cryptographic Protocols. In *Proceedings of the Ninth IEEE Computer Security Foundations Workshop*, pages 62 – 77, County Kerry, Ireland, June 1996.

[14] S.H. Brackin. Deciding Cryptographic Protocol Adequacy with HOL: The Implementation. In *The 1996 International Conference on Theorem Proving in Higher Order Logics*, pages 61–76, Turku, Finland, August 1996.

[15] S.H. Brackin. An Interface Specification Language for Automatically Analyzing Cryptographic Protocols. In *Proceedings of the Internet Society Symposium on Network and Distributed System Security*, February 1997.

[16] S.H. Brackin. Evaluating and Improving Protocol Analysis by Automatic Proof. In *Proceedings of the Eleventh IEEE Computer Security Foundations Workshop*. IEEE Computer Society Press, 1998.

[17] S.H. Brackin and R.W. Lichota. CASE for High Assurance: Utilizing Commercial Technology for Automated Cryptographic Protocol Analysis. In *Proceedings of the Sixth Annual Dual-Use Technologies and Applications Conference*, June 1996.

[18] S. Budkowski and P. Dembinski. Introduction to Estelle: A Specification Language for Distributed Systems. *Computer Networks and ISDN Systems*, 14(1):3 – 23, 1987.

[19] M. Butow, P.S. Kritzinger, M. Mestern, and C. Schapiro. Performance Modelling with the Formal Specification Language SDL. In *Proceedings of the FORTE/PSTV '96: XVth International Symposium on Protocol Specification, Testing and Verification*, pages 213 – 225, Kaiserslautern, Germany, 1996.

[20] E.A. Campbell and R. Safavi-Naini. On Automating the BAN Logic of Authentication. In *Proceedings of the Fifteenth Australian Computer Science Conference*, 1992.

[21] CCITT. *CCITT X.509, The Directory — An Authentication Framework*, 1988.

[22] J. Clark. Attacking Authentication Protocols. *High Integrity Systems*, 1(5):465 – 474, August 1996.

[23] J. Clark and J. Jacob. *A Survey of Authentication Protocol Literature: Version 1.0*, November 1997.

[24] S.C. Clark, S.B. Freedman, and J.K. Millen. The Interrogator: Protocol Security Analysis. *IEEE Transactions on Software Engineering*, SE-13(2), 1987.

[25] W.F. Clocksin and C.S. Mellish. *Programming in Prolog*. Springer-Verlag, New York, 1981.

[26] Data Network Architectures Laboratory, University of Cape Town. *MicroSnap – Multiclass Queuing Network Analyser (User Manual)*, third edition, August 1990.

[27] B. de Decker and F. Piessens. CryptoLog: A Theorem Prover for Cryptographic Protocols. In *DIMACS Workshop on Design and Formal Verification of Security Protocols*. Rutgers University, September 1997.

[28] G. Denker and J. Millen. CAPSL and CIL Language Design. Technical Report SRI-CSL-99-02, SRI International, February 1999.

[29] D. Dolev and A. Yao. On the Security of Public Key Protocols. *IEEE Transactions on Information Theory*, 29(2):198 – 208, 1983.

[30] N. Ford. *Prolog Programming*. John Wiley and Sons, 1989.

[31] P. Georgiadis, S. Gritzalis, and D. Spinellis. Security Protocols Over Open Networks and Distributed Systems: Formal Methods for Their Analysis, Design and Verification. *Computer Communications*, 22(8):695 – 707, May 1999.

[32] V.D. Gligor, L. Gong, R. Kailar, and S. Stubblebine. Logics for Cryptographic Protocols – Virtues and Limitations. In *Proceedings of the Fourth IEEE Computer Security Foundations Workshop*, pages 219 – 226, Franconia, New Hampshire, October 1991. IEEE Computer Society Press.

[33] L. Gong. *Cryptographic Protocols for Distributed Systems*. PhD thesis, University of Cambridge, April 1990.

[34] L. Gong. Lower Bounds on Messages and Rounds for Network Authentication Protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 26 – 37, Fairfax, Virginia, November 1993.

[35] L. Gong. Efficient Network Authentication Protocols: Lower Bounds and Optimal Implementations. *Distributed Computing*, 9(3):131 – 145, 1995.

[36] L. Gong, R. Needham, and R. Yahalom. Reasoning about Belief in Cryptographic Protocols. In *Proceedings of the 1990 IEEE Symposium on Research in Security and Privacy*, pages 234 – 248, Oakland, California, 1990. IEEE Computer Society Press.

[37] L. Gong and P.F. Syverson. Fail-Stop Protocols: An Approach to Designing Secure Protocols. In *The Fifth International Working Conference on Dependable Computing for Critical Applications*, pages 44 – 55. Springer-Verlag, September 1995.

[38] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, Cambridge, UK, 1993.

[39] G.L. Hammonds, R. Burns, Y. Koh, R.W. Lichota, and S.H. Brackin. Applying CASE Tools to Security Engineering. *CrossTalk: The Journal of Defense Engineering*, 9(12):14, December 1996.

[40] N. Heintze and J.D. Tygar. A Model for Secure Protocols and Their Compositions. In *1994 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 2 – 13. IEEE Computer Society Press, May 1994.

[41] Interactive Development Environments. *Creating OMT Models*, February 1994.

[42] Interactive Development Environments. *Fundamentals of StP, Release 1*, February 1994.

[43] International Telecommunication Union, Geneva. *ITU-TS Recommendation Z.100: Specification and Description Language (SDL)*, 1993.

[44] International Telecommunication Union, Geneva. *ITU-TS Recommendation Z.120: Message Sequence Chart (MSC)*, 1993.

[45] J. Gosling, H. McGilton. *The Java Language Environment*. Sun Microsystems, May 1996.

[46] R. Kemmerer. Analyzing Encryption Protocols using Formal Verification Techniques. *IEEE Journal on Selected Areas in Communications*, 7(4):448 – 457, 1989.

[47] D. Kindred and J.M. Wing. Fast, Automatic Checking of Security Protocols. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, Oakland, California, November 1996.

[48] D.E. Knuth. *The Art of Computer Programming*, volume 1. Addison-Wesley Publishing Company, 1968. Fundamental Algorithms.

[49] J. Kohl and C. Neuman. *RFC 1510: The Kerberos Network Authentication Service (V5)*, September 1993.

[50] R. Lichota, G. Hammonds, and S.H. Brackin. Verifying the Correctness of Cryptographic Protocols using Convince. In *Proceedings of the Twelfth IEEE Computer Security Applications Conference*, pages 117 – 128. IEEE Computer Society Press, 1996.

[51] R.W. Lichota, G.L. Hammonds, and S.H. Brackin. Verifying Cryptographic Protocols for Electronic Commerce. In *Proceedings of the Second USENIX Workshop on Electronic Commerce*, Oakland, California, November 1996.

[52] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. In *Proceedings of TACAS*, pages 147 – 166. Springer Verlag, 1996.

[53] A. Mathuria, R. Safavi-Naini, and P. Nickolas. Some Remarks on the Logic of Gong, Needham and Yahalom. In *Proceedings of the International Computer Symposium 1994*, volume 1, pages 303 – 308, Hsinchu, Taiwan, December 1994.

[54] A. Mathuria, R. Safavi-Naini, and P. Nickolas. On the Automation of GNY Logic. In *Proceedings of the 18th Australian Computer Science Conference*, volume 17, pages 370 – 379, Glenelg, South Australia, February 1995.

[55] C. Meadows. Analyzing the Needham-Schroeder Public Key Protocol: A Comparison of Two Approaches. In *Proceedings of the European Symposium on Research in Computer Security 1996*, volume 1146 of *Lecture Notes in Computer Science*, pages 351 – 364. Springer Verlag, 1996.

[56] C.A. Meadows. Formal Verification of Cryptographic Protocols: A Survey. In *Advances in Cryptology - Asiacrypt '94*, pages 133 – 150. Springer-Verlag, 1995.

[57] C.A. Meadows. The NRL Protocol Analyzer: An Overview. *Journal of Logic Programming*, 26(2):113 – 131, February 1996.

[58] M. Meeker and S. Pearson. *The Internet Retailing Report.* Morgan Stanley, Dean Witter, Discover & Co., New York, May 1997. [http://www.morganstanley.com/techresearch/inet/morganx.html].

[59] J. Millen, C. Neuman, J. Schiller, and J. Saltzer. *Kerberos Authentication and Authorization System.* Project Athena Technical Plan, M.I.T., Massachusetts, 1987. Section E.2.1.

[60] J. Miller. Clean Up: C++ Garbage Collection. *BYTE*, 21(1):157 – 158, January 1996.

[61] D. Monniaux. Decision Procedures for the Analysis of Cryptographic Protocols. In *Proceedings of the Twelfth IEEE Computer Security Foundations Workshop*, Mordano, Italy, June 1999. IEEE Computer Society Press.

[62] Mudge and B. Schneier. Cryptanalysis of Microsoft's Point-to-Point Tunneling Protocol (PPTP). In *Proceedings of the 5th ACM Conference on Computer and Communications Security*, pages 132–141, 1998.

[63] R.M. Needham and M.D. Schroeder. Using Encryption for Authentication in Large Networks on Computers. *Communications of the ACM*, 21(12):993 – 999, December 1978.

[64] D. Otway and O. Rees. Efficient and Timely Mutual Authentication. *ACM Operating Systems Review*, 21(1):8 – 10, 1987.

[65] L. Paulson. Mechanized Proofs for a Recursive Authentication Protocol. In *Proceedings of IEEE Security Foundations Workshop X*, pages 84 – 94. IEEE Computer Society Press, 1997.

[66] L. Paulson. Proving Properties of Security Protocols by Induction. In *Proceedings of IEEE Security Foundations Workshop X*, pages 70 – 83. IEEE Computer Society Press, 1997.

[67] J.B. Postel. *RFC 821: Simple Mail Transfer Protocol*, August 1982.

[68] J. Preece, Y. Rodgers, H. Sharp, D. Benyon, S. Holland, and T. Carey. *Human-Computer Interaction.* Addison-Wesley, 1994.

[69] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *In Theory And Practice of Objects Systems*, 4(1):27 – 50, 1998.

[70] N. Rowe. *Artificial Intelligence Through Prolog.* Prentice-Hall, 1988.

[71] E. Saul and A.C.M. Hutchison. SPEAR II: The Security Protocol Engineering and Analysis Resource. In *Second Annual South African Telecommunications, Networks and Applications Conference*, pages 171 – 177, Durban, South Africa, September 1999.

[72] E. Saul and A.C.M. Hutchison. A Generic Graphical Specification Environment for Security Protocol Modelling. In *Proceedings of the Sixth Annual Working Conference on Information Security*, pages 311–320, Beijing, China, August 2000. Kluwer Academic Publishers.

[73] E. Saul and A.C.M. Hutchison. A Graphical Environment for the Facilitation of Logic-Based Security Protocol Analysis. *South African Computer Journal*, (26):196 – 200, November 2000.

[74] E. Saul and A.C.M. Hutchison. An Environment to Facilitate the Teaching of GNY-Based Security Protocol Analysis Techniques. In *Proceedings of the Second World Conference in Information Security Education*, pages 285–305, Perth, Western Australia, July 2001. Edith Cowen University.

[75] B. Schneier. Why Cryptography is Harder than it Looks. *Information Security Bulletin*, 2(2):31 – 36, March 1997.

[76] E. Snekkenes. *Formal Specification and Analysis of Cryptographic Protocols.* PhD thesis, University of Oslo, Norway, 1995.

[77] D. Steedman. *Abstract Syntax Notation One (ASN.1) : The Tutorial and Reference.* Technology Appraisals, Twickenham, UK, 1993.

[78] J. Steiner, C. Neuman, and J. Schiller. An Authentication Service for Open Network Systems. In *Proceedings of the USENIX Winter Conference*, pages 191 – 202, Oakland, California, February 1988.

[79] P.F. Syverson and P.C. van Oorschot. On Unifying Some Cryptographic Protocol Logics. In *Proceedings of the 1994 IEEE Symposium on Research in Security and Privacy*, pages 14 – 29, Oakland, California, May 1994. IEEE Computer Society Press.

[80] J. Tardo and K. Alagappan. SPX: Global Authentication Using Public-Key Certificates. In *Proceedings of the Symposium on Security and Privacy*, pages 232 – 244, Oakland, California, 1991. IEEE Computer Society Press.

[81] P.C. van Oorschot. Extending Cryptographic Logics of Belief to Key Agreement Protocols (Extended Abstract). In *Proceedings of the First ACM Conference on Computer and Communications Security*, pages 232 – 243. ACM Press, August 1991.

[82] H. Wang, M.K.O. Lee, and C. Wang. Consumer Privacy Concerns About Internet Marketing. *Communications of the ACM*, 41(3):63 – 70, March 1998.

[83] Jan Wielemaker. *SWI-Prolog 3.4 Reference Manual*, September 2000. [http://www.swi.psy.uva.nl/projects/SWI-Prolog].

Index