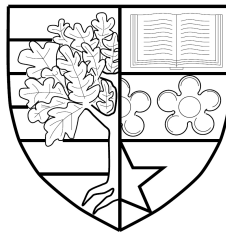


Model Checking Web Applications

by

Mohammed Yahya Alzahrani



SUBMITTED FOR THE DEGREE DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE

SCHOOL OF MATHEMATICAL AND COMPUTER SCIENCES

HERIOT-WATT UNIVERSITY

December 2015

The copyright in this thesis is owned by the author. Any quotation from the thesis or use of any of the information contained in it must acknowledge this thesis as the source of the quotation or information.

Abstract

The modelling of web-based applications can assist in capturing and understanding their behaviour. The development of such applications requires the use of sound methodologies to ensure that the intended and actual behaviour are the same.

As a verification technique, model checking can assist in finding design flaws and simplifying the design of a web application, and as a result the design and the security of the web application can be improved. Model checking has the advantage of using an exhaustive search of the state space of a system to determine if the specifications are true or not in a given model.

In this thesis we present novel approaches in modelling and verifying web applications' properties to ensure their design correctness and security. Since the actions in web applications rely on both the user input and the server status; we propose an approach for modelling and verifying dynamic navigation properties. The SPIN model checker has been used successfully in verifying communication protocols. However, the current version of SPIN does not support modelling time. We integrate discrete time in the SPIN model to allow the modelling of realistic properties that rely on time constraints and to analyse the sequence of actions and time. Examining the sequence of actions in web applications assists in understanding their behaviour in different scenarios such as navigation errors and in the presence of an intruder. The model checker UPPAAL is presented in the literature as an alternative to SPIN when modelling real-time systems. We develop models with real time constraints in UPPAAL in order to validate the results from the SPIN models and to compare the differences between modelling with real time and with discrete time as in SPIN. We also compare the complexity and expressiveness of each model checker in verifying web applications' properties.

The web application models in our research are developed gradually to ensure their correctness and to manage the complexities of specifying the security and navigation properties. We analyse the compromised model to compare the differences in the sequence of actions and time with the secure model to assist in improving early detections of malicious behaviour in web applications.

To my parents

Acknowledgements

I am grateful to my supervisor Dr. Lilia Georgieva for her guidance, support and encouragement throughout my research. Her extensive comments during the early stages of my Ph.D. to the writing up helped me to understand my research. Without her inspiration, knowledge and enthusiasm, this work would never have been finished.

I am also grateful to Professor Gerard Holzmann for his support and patience in answering my questions regarding the SPIN models.

I would also like to thank Vaggelis for all the encouragement and support during my Ph.D.

Also, many thanks to my office colleagues Prabhat and Konstantina for their support and useful comments.

My gratitude goes to Nesreen for being an endless source of inspiration and for supporting for me to undertake my Ph.D.

Last but not least, I am so grateful for my parents and my siblings (Elham, Seham, Ali, Wafaa, Waleed and Dana) whose constant encouragement, love and support helped me throughout my Ph.D. It is to them that I dedicate this work.

Contents

Abstract	i
Acknowledgements	iii
Contents	iv
List of Figures	vii
List of Tables	ix
1 Introduction	1
1.1 Web Applications	1
1.2 Formal Methods	4
1.3 State of the Problem	6
1.4 Research Aims and Objectives	6
1.5 Research Methodology	7
1.6 Contributions	10
1.7 Publications	12
1.8 Structure of Thesis	13
2 Background and Related Work	15
2.1 Web Applications	15
2.1.1 Security Threats for Web Applications	19
2.2 Web Navigation Properties	22
2.3 Web Security Properties	24
2.3.1 Session Management	27
2.3.2 Authentication	28
2.3.3 Control-Flow	29
2.4 Model Checking	31
2.4.1 Model Checking Tools	34
2.5 Temporal Logic	35
2.5.1 Linear Temporal Logic (LTL)	36
2.5.2 Computational Tree Logic (CTL)	37
2.5.3 Temporal Logic Patterns	39

2.6	Timed Automata Theory	40
2.6.1	Formal Syntax	41
2.7	Modelling an Intruder (Man in the Middle)	43
2.8	Conclusions	44
3	Modelling in SPIN	46
3.1	The SPIN Model Checker	46
3.1.1	Promela	48
3.1.2	Verification in Spin	53
3.1.3	Modelling Time in SPIN	55
3.2	Modelling Web Applications in SPIN	56
3.2.1	Model without Timer	58
3.2.1.1	Simulation and Verification Results of Model without Timer	64
3.2.2	Modelling Dynamic Navigation	79
3.2.2.1	Simulation and Verification Results of the Dynamic Navigation Model	82
3.2.3	Modelling with Time Constraints	85
3.2.3.1	Simulation and Verification Results of Timed Model	90
3.2.4	Adding an Intruder to the Model	90
3.2.4.1	Simulation and Verification Results of Model with In- truder	93
3.3	Summary	95
4	Modelling in UPPAAL	97
4.1	The UPPAAL Model Checker	97
4.1.1	The Modelling Language	98
4.1.2	Modelling Time in UPPAAL	101
	Locations in UPPAAL	105
4.2	Modelling Web Applications in UPPAAL	105
4.2.1	Model without Time Constraints	106
4.2.1.1	Simulation and Verification Results of Model without Time Constraints	110
4.2.2	Modelling Dynamic Navigation	112
4.2.2.1	Simulation and Verification Results of Dynamic Nav- igation Model	114
4.2.3	Modelling with Time Constraints	115
4.2.3.1	Simulation and Verification Results of Timed Model	116
4.2.4	Adding an Intruder to the Model	117
4.2.4.1	Simulation and Verification Results of Model with In- truder	118
4.3	Summary	119
5	Comparison	121
5.1	Modelling Web Applications in SPIN	121

5.2	Modelling Web Applications in UPPAAL	125
5.3	Comparison	127
5.4	Summary	130
6	Conclusion	131
6.1	An Overview of the Research	131
6.2	Summary of Thesis Contributions to Research Areas	133
6.2.1	Contributions to Model Checking Web Applications	133
6.2.2	Contributions to Model Checking Timed Models of Web Appli- cations	134
6.2.3	Contributions to Modelling Security Properties of Applications .	134
6.3	Future Work	135
	Bibliography	136
	Appendix A	145

List of Figures

1.1	Model Checking Web Application Properties.	9
2.1	Overview of Web Applications [Li and Xue, 2014]	18
2.2	Percentage of Common Vulnerability Types in Web Applications [Cen- zic, 2014]	20
2.3	Model Checking Process.	32
2.4	Attack Example	43
3.1	Model of Online Banking.	57
3.2	Safety Verification of Model without Timer	65
3.3	Message Sequence Chart of Model without Timer.	65
3.4	Page Sequence and LTL Proposition Letters.	66
3.5	Verification Result of Property 3.1	67
3.6	Verification Result of Property 3.2	68
3.7	Verification Result of Property 3.3	69
3.8	Verification Result of Property 3.4	69
3.9	Verification Result of Property 3.5	71
3.10	Never Claim for Property 3.5	72
3.11	Verification Result of Property 3.6	73
3.12	Never Claim for Property 3.6	74
3.13	Verification Error Result of Property 3.13	75
3.14	Error-trail File of Property 3.13	75
3.15	Verification Result of Property 3.7	76
3.16	Verification Error Result of Property 3.7	77
3.17	Verification Result of Property 3.8	77
3.18	Verification Result of Property 3.9	78
3.19	Safety Verification Result of Dynamic Navigation Model	82
3.20	Message Exchange Sample of Dynamic Navigation Model	83
3.21	Verification Result of Property 3.10	84
3.22	Verification Result of Global Variable Assertions	84
3.23	Verification Result of Property 3.12	85
3.24	Simulation Chart of the Discrete Time Model.	88
3.25	Simulation Results of the Discrete Time Model	89
3.26	Verification Result of Property 3.13	90
3.27	Secure Model.	94
3.28	Model with Intruder.	95

4.1	Path Formulas Supported in UPPAAL.	100
4.2	The Automata <i>P1</i> with <i>Obs</i> Observer.	101
4.3	Possible Behaviour of the First Example.	102
4.4	UPPAAL Verification Example.	103
4.5	UPPAAL Behaviour with Invariant.	103
4.6	UPPAAL Behaviour with Guard.	104
4.7	Location Types in UPPAAL.	105
4.8	Client Automaton.	109
4.9	Server Automaton.	110
4.10	Simulation Result of Model without Time Constraints.	111
4.11	Verification Result of CTL Formula 4.1	111
4.12	Dynamic Client Automaton.	113
4.13	Dynamic Server Automaton.	114
4.14	Verification Results of CTL Formula 4.6	114
4.15	Timed Client Automaton.	116
4.16	Timed Server Automaton.	116
4.17	Intruder Automaton.	118

List of Tables

2.1	Navigation Properties [Stock et al., 2014].	22
2.2	Session Management Properties [Stock et al., 2014].	28
2.3	Authentication Properties [Stock et al., 2014].	29
2.4	Control-Flow Properties [Stock et al., 2014].	30
2.5	LTL formula operators with their mathematical and SPIN notation . . .	37
2.6	Temporal Logic Patterns	39
3.1	Operators in Promela.	50
3.2	Verification Results of Properties for Model without Time.	78
3.3	Verification Results of the Secure and the Compromised Model.	94
4.1	CTL Syntax in UPPAAL	99
4.2	Verification Results of the Secure Model and Compromised Model in UPPAAL.	118
5.1	Web Applications' Properties Stock et al. [2014].	123
5.2	Comparison of Number of States between SPIN and UPPAAL.	127

Chapter 1

Introduction

In this chapter we first discuss the challenges of web applications' development that lead to security and design vulnerabilities. In Section [1.2](#) we provide an overview of the formal methods and present the model checking advantages over alternative verification methods. We then present the state of problem of our research in Section [1.3](#). In Section [1.4](#) we list and discuss the research aims and objectives. In Section [1.5](#) we present the research methodology. In Section [1.6](#) we show the contributions of our research. We conclude with a structure of the following chapters in Section [1.8](#).

1.1 Web Applications

Web applications are common in today's economic and social life. Such applications provide business services to customers, business to business communications, and various services to users around the world. Online businesses use web applications to reach more clients and to improve their services. Sectors such as banking, travel, education and governmental services rely on web applications to promote and increase

their operations [Ginige and Murugesan, 2001, Homma et al., 2011, Miao and Zeng, 2007]. The rapid spread of web applications in the areas of communications and business services has promoted them to one of the leading and most essential branches of the software development industry [Offutt, 2002]. Along with the increased demand for web applications, concerns have been raised about design flaws that are able to cause vulnerabilities in security and navigation properties [Huang and Lee, 2005].

The development of web applications has been evolving rapidly, resulting in poor quality, security vulnerabilities and maintenance challenges [Murugesan and Deshpande, 2002]. Unstable design and development processes, as well as poor project management practices are the main reasons for such problems [Ginige, 2002]. The data handled by web applications often contains sensitive values (e.g. credit card numbers) for both users and service providers. In 2015, the attack of several organizations' web applications was considered the most popular method that led to sensitive data disclosure [Hesseldahl, 2015, Solutions, 2015].

Web application vulnerabilities, which lead to the compromise of sensitive information, are regularly reported [Falk et al., 2008, Jovanovic et al., 2006], as indicated by the following reports:

- According to a report by [Cenzic, 2014], 96% of tested web applications in 2013 had vulnerabilities categorised as high risk. In addition, an average number of 14 vulnerabilities per web application found in 2013 due to design errors.
- A recent report by [Hoff, 2013] showed that there were more than 800 reported hacking incidents in 2012 alone, and 70% of those were carried out through web application vulnerabilities.

- A study carried out by [Falk et al., 2008] showed that 75% of online banking web sites have at least one major security flaw.
- In 2010, more than 8,000 online banking clients' credentials were stolen from a server where they were stored as plain text [Fundation, 2010].

A report by [Solutions, 2015] stated: “ by tracking user behaviour and using some form of fraud detection to get an early warning of suspicious behaviour ...can help to identify malicious activity before your last bit of sensitive data is fully exfiltrated.”

Larger and more complex web applications will also increase the need for rigorous methods of developing high quality applications that are secure and easy to maintain [Lee and Shirani, 2004, Ricca and Tonella, 2001, Taylor et al., 2002]. The development of such applications requires the use of sound methodologies to ensure that the intended and actual behaviour are the same. Also, web applications must satisfy essential security properties, such as authentication, session management and navigation properties [Stock et al., 2014].

In this thesis we apply model checking for the simulation and verification of time sensitive web applications. We model security and navigation properties which include session management properties, authentication properties and control flow properties. We use the model checking tools SPIN [Holzmann, 2004] and UPPAAL [Amnell et al., 2001] to verify an online banking web application of a client communicating with a server to complete a transaction.

1.2 Formal Methods

Formal methods are mathematical based languages, techniques and tools for verifying hardware and software systems. The process of using formal methods does not guarantee the correctness of a given system, but they can assist in increasing the understanding of a system's inconsistencies and incompleteness that can lead to design errors [Clarke and Wing, 1996].

Traditional validation techniques, such as testing, can be effective in the early stages of debugging. However, testing can not detect all the errors and in some cases it can miss errors in systems that have very large number of states, as the testing process can only explore part of the possible behaviour of the system. Furthermore, it is not evident when they have reached their limit, nor is there a clear estimate of the remaining number of bugs [Clarke et al., 1999, Donini et al., 2006].

In contrast, theorem proving and proof checking do not have this shortcoming. However, they are time consuming and often require that the design team includes an expert in both the language used to model the system and the mathematical background of the language. In addition, theorem proving is complex when timing requirements are included in verification [Davis, 2000].

An alternative approach is formal verification, which can exhaustively explore the possible behaviour of a system. In contrast to testing, where only some parts of the behaviour are explored, formal verification can show that a design is correct by exploring all possible states; thus not allowing a security vulnerability or design flaw [Clarke et al., 1999].

Model checking tools have played a key role in the design of concurrent and distributed systems and have also been reported in industrial applications [Baier et al., 2008,

[Clarke, 2008, Holzmann, 2004]. The model checking process assists designers to ensure the correctness of a system in the early stages of development.

In order for a model checking tool to verify a web application model, three main tasks need to be carried out. The first task is **modelling**, in which the systems' design is converted into a formalism that is accepted by the model checker tool. In some cases, this is a straightforward task, while complex systems may require the use of abstraction to eliminate unrelated or non-essential system details.

The second task is **specification**; stating the properties of the model that the system must satisfy. Model checking tools commonly use *temporal logic*, which can assert how the behaviour of the system evolves over time. The final task is **verification**. Ideally, this task is performed in a completely automated fashion. The model checker tool will provide an error trace (counterexample) that assists in locating where an error occurred in the case of a negative result. Each of these are examined and demonstrated further in Chapter 2.

Model checking has two important advantages over other techniques [Baier et al., 2008, Clarke, 2008, Clarke et al., 1999]:

- The process is fully automatic, so the user does not need to be an expert in mathematical disciplines such as logic and theorem proving.
- The model checker tool provides a (counterexample) that shows where the error has occurred if the property fails. This error trace provides an insight to understand the reason for the error, as well as essential clues to fix the problem.

The main disadvantage of model checking is the *state explosion problem* where the number of states of a system to be analysed or verified increases significantly in the state space [Holzmann, 2004, McMillan, 1992, Valmari, 1998].

1.3 State of the Problem

Web applications are dynamically changing and evolving. They are used in services such as banking, governmental and health sectors [Homma et al., 2011, Huang and Lee, 2005, Krishnamurthi, 2006], as web application often involve the transmission of sensitive data and they need to ensure correctness to avoid vulnerabilities. Security is a major concern for developers, since simple errors could lead to the loss of valuable information and threaten the privacy of online users. As a result, the need for automated tools that detect vulnerabilities and protect users against attacks is evident. Verifying web applications using model checking is an emerging research area, and there is a clear gap between the theory and practice. This research investigates web application behaviour under different situations (e.g. in the presence of an attacker or different server status). Realistic web application models are built and extended with time constraints to verify and analyse their behaviour.

1.4 Research Aims and Objectives

In this research we apply model checking for modelling and verifying web application behaviour under different scenarios. In particular, the focus is on web application security and navigation properties. This aim can be achieved by fulfilling three interconnected objectives, as follows:

- Develop web application models that extend and verify its properties by adding time constraints to achieve realistic models. In addition, secure models will be investigated and compared with a model in the presence of an attacker to study the weaknesses of the specifications and the sequence of timing and actions.

The results could capture the behaviour of the attacker in order to identify vulnerabilities in the models and to analyse compromised and secure models.

- Apply model checking to verify web applications' behaviour and compare it with other verification methods.
- Finally, we present a critical review of the formal methods and investigate the landscape of web application modelling and verification techniques.

In this thesis we present a novel approach for the modelling of web applications. We gradually include features to the models in verify additional properties in each model. We integrate discrete time in the SPIN model checker [Holzmann, 2004] to model properties that rely on time constraints. The advantage of using discrete time is that we were able to capture the value of time at each step in order to compare the behaviour of different models. UPPAAL [Amnell et al., 2001] uses real time modelling, which we first use to validate the results obtained from the SPIN. Secondly, we compare both tools in the context of verifying web applications.

1.5 Research Methodology

Modelling can provide significant benefits to web application development. The view of a system shifts from basic implementation to more detailed aspects, such as security that will improve the quality of the final product. Model checking assists in understanding the interactions and states of web applications, reducing design flaws and ensuring consistent conditions and well-defined behaviour [Schätz, 2004]. Additional benefits of model checking for web applications are [Baier et al., 2008, Clarke, 2008, Clarke et al., 1999]:

Modelling phase: Describing and analysing the high-level, abstract and non-deterministic behaviour of the application avoids the cost of implementation details that could complicate the design. Errors can be caught more easily earlier in a less expensive development phase.

Properties definition The properties of the web application model to be verified are defined by using temporal logic formulas, for example in SPIN the Linear Temporal Logic (LTL) [Burstall, 1974, Kröger, 1977, Pnueli, 1981] is used, while UPPAAL uses a subset of the Computation Tree Logic (CTL) [Huth and Ryan, 2006].

Simulation phase: Using simulation and verification to analyse the model and interactions, we identify potential issues, such as the undesired behaviour of the system and modelling errors.

Verification phase: In this phase we verify that the model guarantees the properties of the system in different scenarios. If the property fails, the model checking failure analysis assists in finding the error through a trace. We either use temporal logic formulas or simple assertions.

Figure 1.1 shows the process we will use in Chapters 3 and 4 to analyse and verify the properties of web applications.

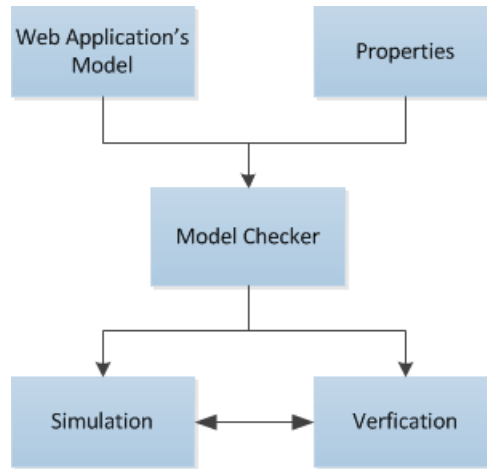


FIGURE 1.1: Model Checking Web Application Properties.

According to [Fenton and Bieman, 2014], a formal experiment is a rigorous and controlled investigation of a model in which important variables are identified and changed such that the outcome can be validated. [Mendes and Mosley, 2006] stated that formal investigation is best suited to the web applications research community, as it is applicable across various types of projects and processes. In a formal investigation a variable is manipulated such that all possible variable value are validated.

The *formal analysis framework* used in this research consists of four components. The application and properties are expressed using *formal semantics*. A *formal language* is then used to describe the system. Next, a formal language is used to describe the property under analysis. Finally, a *formal technique* checks whether the application satisfies the property.

In our research, we are interested in the verification of the applications' behaviour properties, rather than the data transmission properties. We model both web page transactions and the input of web applications, as the dynamic nature of web applications means that the input could lead to different pages (e.g. wrong authentication credentials). The dynamic nature of web applications could be affected by different input from the user, or by the server state.

As SPIN does not support the modelling of time constraints, it will be extended with discrete time, enabling the construction of realistic web application models. In the model checking of timed models, discrete time is preferred to reduce the risk of *state space explosion* [Valmari, 1998], which is one drawback of model checking. Modelling with real time could result in an increase of the system's states up to an intractable level. *The state space explosion* problem will be discussed in Chapter 2.

1.6 Contributions

In this thesis, we made the following contributions:

- The challenges in adopting model checking for the analysis and verification of web applications are critically reviewed. The usage of model checking is examined for critical properties of web applications, such as security, navigation and time-sensitive properties. After providing background information on the current challenges in verifying web applications, methods are devised to develop more secure and easy-to-maintain web applications. In Chapter 2 we present and discuss the challenges in more detail.
- We design a novel web application model and extend it with the novel approach of time constraints to enable the modelling of web application properties. The time constraints assist in time stamping the messages exchanged between parties in the communication and also in the analysis of the sequence of actions. By adding time we are able to express properties, such as modelling session management properties and dynamic navigation properties, where a timeout can lead to different pages. Chapter 3 describes the steps in modelling time constraints.

- We develop web application models in the SPIN model checker. We first analyse the models without time constraints to understand the difference when we add time and to ensure the correctness of the models. We then model dynamic navigation properties by showing how different input from both sides could affect the simulation and verification process. We then introduce a novel approach for modelling the discrete time process so we can model further time-related properties such as session management properties. Finally, we add an intruder to the model to analyse the behaviour of the system in different scenarios. Chapter [3](#) describes the modelling steps in further detail.
- By analysing the time sequence and action sequence within a web application session, we can identify the difference between a secure session and a compromised session, with the presence of an intruder. Understanding the web application behaviour in different scenarios leads to an improved security and more stable development. Furthermore, our approach can assist in developing methods to detect malicious behaviour at early stages. This is analysed in Chapter [4](#).
- In addition to modelling the static properties of web applications, a novel approach was developed for modelling the dynamic properties of web applications, in which a single input can lead to different pages based on time constraints and server state. As highlighted in the literature review, there is a gap in modelling the dynamic navigation properties of web applications. Our research shows how it is possible to model web applications using the model checking tool's existing capabilities, resulting in simplified models that contain security and navigation properties. We present the models in further detail in Chapter [3](#) and Chapter [4](#).

- We verify web applications' properties in the UPPAAL real-time model checker. UPPAAL has a graphical editor which makes it easy to design a system model, along with a graphical simulator that shows the possible dynamic behaviour of a system description. We compare the models in Chapter 5.
- A comparison was made between the SPIN and UPPAAL model checkers for web application analysis and verification. This comparison aims to answer the following questions:
 - What is the complexity and expressiveness level of the model checking tool to verify the properties of web applications models?
 - To what extent can the property specification language be adapted to the specification of web application properties?
 - How capable is the model checker for verifying models without resulting in a state explosion problem?
 - How are the results different when integrating a simple timing constraint into SPIN, in contrast to UPPAAL, which is based on timed automata specifications?

The outcome results validate the rationale for using model checking in web application development, as explained in Chapter 5.

1.7 Publications

Part of the work presented in this thesis has been published and presented in peer-reviewed conferences and workshops:

1. Alzahrani, M. & Georgieva, L. (2012) Modelling Trusted Web Applications. 1st International Workshop on Trustworthy Multi-Agent Systems. KES-AMSTA Special Session, Dubrovnik, Croatia, 25-27 June.
2. Alzahrani, M. & Georgieva, L. (2012) Analysing Data-Sensitive and Time-Sensitive Web Applications at the 19th Automated Reasoning Workshop, University of Manchester. 2nd-4th April.
3. Alzahrani, M. & Georgieva, L. (2013) Comparative analysis of time-sensitive web applications using SPIN and UPPAAL at the 20th Automated Reasoning Workshop, University of Dundee on 11-12 April.
4. Alzahrani, M. (2015) Model Checking Web Applications using SPIN and UPPAAL at 15th International Workshop on Automated Verification of Critical Systems, Edinburgh, 1-4 September.

1.8 Structure of Thesis

The remainder of this thesis is organised as follows:

Chapter 2 summarises the background on web application fundamentals and properties, and provides a comparison of the analysis and verification methods found in the literature. Model checking and the tools used in this research are then described, as a basis for subsequent chapters.

Chapter 3 presents the first model checker, SPIN. First the tool and its input language PROMELA are described. Second, the web application is modelled, and

a description of the steps followed during the modelling and verification is provided. The secure and compromised models are presented, and then the simulation and verification results are shown.

Chapter 4 describes the second model checking tool, UPPAAL. First a brief description of the tool is presented, followed by background information on timed automata theory as a basis for modelling web applications. A comparison of the secure and compromised models is made, and subsequently, the simulation and verification results are provided.

Chapter 5 provides a comparison between the results obtained from Chapter 3 and Chapter 4. The results of the experiments are analysed, illustrating the differences between the tools, as well as the challenges of modelling web applications.

Chapter 6 assesses the results that were obtained and presents conclusions, contributions, limitations and possible future work.

Chapter 2

Background and Related Work

This chapter describes the development process of web applications and the challenges arising in both the design and implementation phases. We then outline evolving trends and discuss related work. We then present an overview of model checking principles for web applications. Moreover, we list the verification requirements for web applications. The OWASP Application Security Verification Standard [Stock et al., 2014], which is updated annually, is used to illustrate a list of verification properties. This chapter provides the research context and lays the foundation for the modelling and analysis work described in the next chapters.

2.1 Web Applications

Web applications enable much of today’s online business; including banking, social networking and governmental activities, to thrive. As a result of the rapid development of new programming models and technologies, web applications are evolving

continuously. The results of such rapid change for web applications brings new challenges [Alpuente et al., 2010, Armando et al., 2010, Conallen, 1999, Di Sciascio et al., 2003].

The security of web applications is a challenging task. Security is a continuous process of identifying and analysing potential threats [MSDN, 2011].

Furthermore, new security challenges emerge due to the increasing amount of application code being moved to the client's side. With larger amounts of code exposed to the user comes greater vulnerability risks. Attackers are able to gain knowledge of the code and are therefore, more likely to compromise the server-side application state. The data protected by web applications are security sensitive in most cases, including credit card details and personal information, and are typically significantly valuable for both users and service providers. Emerging types of attacks, such as the HTTP parameter pollution attack, place a wider range of web applications at risk [Balduzzi et al., 2011]. As a result, major companies offer rewards for finding vulnerabilities within their web applications [Google, 2015].

This inherent complexity poses challenges to the modelling, analysis and verification of this type of application. Some of these challenges are summarised below [Alalfi et al., 2009, Li and Xue, 2014]:

- The complex nature of the web application environment may lead to integration difficulties with other diverse hardware and software platforms. The analysis of many components could make the verification extremely difficult.
- The dynamic behaviour, such as the dynamic interaction between clients and servers, and the continual changes in the system's context and web technologies can be another major challenge.

- Web applications may have several entry points, allowing interaction with the system in a way that cannot be predicted (due to design errors) and that cannot be blocked by the web application.
- Another challenge is the efficient monitoring and tracking of outputs of web applications. Examining the change of states between different components is often difficult to analyse.

The early websites only contained a collection of documents with static content, encoded in the HyperText Markup Language (HTML). Since then, web applications have evolved from static hypermedia to complex and dynamic infrastructures. In addition, development technologies shifts the focus of web applications from information delivery only, to include application execution [Casteleyn et al., 2009].

New technologies have been developed to enable web applications to change from simple static HTML pages to dynamic web pages that are able to interact with other systems [Casteleyn et al., 2009, Conallen, 1999]. Web pages and various elements of web applications are stored on the server. Users primarily interact with the browser; the request from the client's side is sent to the web server and in turn to the database management system. Servers respond to the user's request and carry out data processing to complete the transaction. The processed results are then returned to the user via the web browser.

Web applications are commonly designed as a three-tiered architecture (shown in Figure 2.1) and consist of the following components:

Web browser is the software application that serves as a user interface for presenting information.

Web application server manages the dynamic flow-control of the web application.

The web application server receives user input via the web browser and results from the database server. The code is constructed dynamically and the challenge arises when checking or modifying the incoming data before processing it or when passing it to the lower tiers for execution. Failure in this process can lead to compromising the security of the web application.

Database server provides management and database persistent functionality.

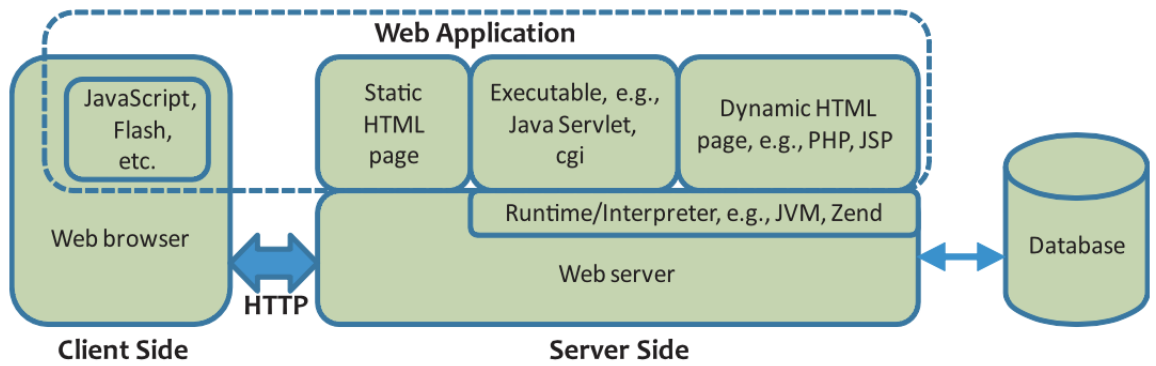


FIGURE 2.1: Overview of Web Applications [Li and Xue, 2014]

Accordingly, the features that differentiate web applications from traditional software and information systems can be summarised below [Casteleyn et al., 2009, Fraternali, 1999]:

- **Accessibility:** Users with different levels of computing skills and with different needs are able to access web applications.
- **Data management:** The data in web applications is distributed in different formats and using various technologies.
- **Architecture complexity:** Web application accessibility requires distributed, multi-tier architectures to access the full range of information and services.

2.1.1 Security Threats for Web Applications

Web applications are built on complex systems consisting of various components and technologies. The current web application development and testing frameworks offer limited support for security validation. Web application development is an error-prone process, and the implementation of security metrics requires substantial effort [Alalfi et al., 2009]. Security relies on the following attributes [MSDN, 2011]:

- **Authentication:** The process of knowing who is accessing the information on the server. All principals of a communication need to prove their identities in order to gain access.
- **Authorization:** The process of controlling the information and actions that an authenticated principal is permitted to access.
- **Auditing:** Developing effective auditing prevent clients from denying their transactions.
- **Integrity:** Ensuring that transmitted data is protected from accidental or deliberate malicious modification.
- **Confidentiality:** Ensuring that the data remains private and confidential from unauthorized users or eavesdroppers who monitor the flow of traffic across a network.
- **Availability:** Ensuring that systems remain available for legitimate users. Denial of service attacks cause the system to crash so that other users cannot gain access.

A large number of web applications deployed on the Internet are open to security vulnerabilities. According to a report by [Cenzic, 2014], 96% of tested web applications

in 2013 had vulnerabilities categorised as “high risk”. In addition, in 2013 an average of 14 vulnerabilities was estimated per web application. A recent report by [Hoff, 2013] showed that in 2012 alone there were more than 800 reported hacking incidents; 70% of those were carried out by exploring web application vulnerabilities.

Figure 2.2 shows the percentage of web applications with respective different common types of vulnerabilities. This increases the difficulty of finding a universal solution for each type, as each one requires a different fix. The three main categories of threats were: session management vulnerabilities; (found in 79% of web applications in 2013); Cross-Site Scripting (XSS) vulnerabilities; (60%); and authentication and authorisation vulnerabilities (56%).

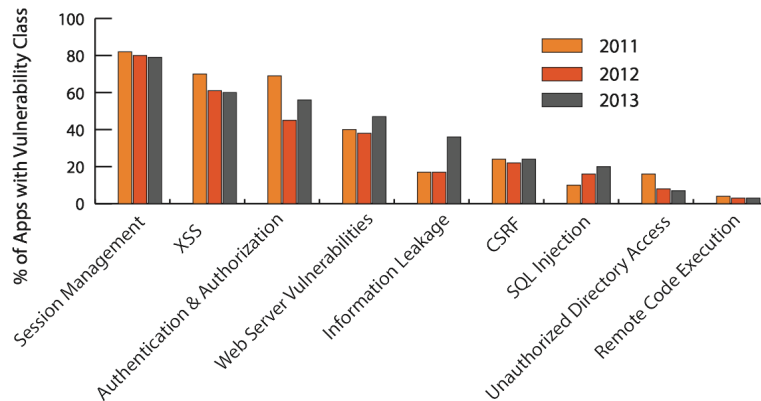


FIGURE 2.2: Percentage of Common Vulnerability Types in Web Applications [Cenzic, 2014]

The development of web applications requires careful consideration with a focus on security and navigation correctness. The use of model-based verification assists in capturing the system’s behaviour. Furthermore, model-based verification can simplify future analysis in order to improve or measure the quality of the system.

In addition, modelling plays an important role during the the software development phase by formally defining the requirements and providing exhaustive detail. A central goal of model-based development is to enable an analysis of the system, thus

ensuring quality at model level. There is a need to consider certain properties of the system prior to implementation, such as deadlock freedom, timing consistency and the availability of memory resources [Engels et al., 2003].

Following traditional software verification, the use of forward engineering-based verification simplifies the development process and establishes a basis for later phases, such as verification. On the other hand, the use of reverse engineering methods to extract models from existing applications simplifies maintenance and evaluation. Forward-engineering verification employed in early development stages enables error detection, alleviating the costs and effort of rectification with respect to errors in later development stages [Huth and Ryan, 2006].

Web application modelling is viewed from different perspectives based on the purpose of the verification.

In order to present and discuss the level of modelling and the scope of properties under this research, we will demonstrate and discuss primary web application properties and the attempts of both researchers and the industry to ensure correctness.

According to a survey carried out by [Alalfi et al., 2009], the level of web applications modelling can be viewed from three perspectives: web navigation, web behaviour and web content. Web content properties are outside the scope of this research, since they mostly rely on checking the programming language and content components used. The other two perspectives of web properties; web navigation and web behaviour are discussed, along with related work, in the following sections.

2.2 Web Navigation Properties

Navigation within a web application is key to ensuring both its security and usability [Kappel et al., 2006]. The web navigation properties are divided into three categories. First, *static navigation* properties address properties such as broken links, reachability (e.g., links to home page), consistency of frame structure and cost of navigation, such as the longest path analysis.

Second are *dynamic navigation* properties, whereby some links may lead to different web pages depending on the input. Input can be provided by either the user or the system. The action then depends on the server that uses information, such as session information, time or date, to apply access control and user privileges.

The third category of properties is *interaction navigation analysis*, which focuses on properties outside the control of the web application, such as user interaction with the web browser, e.g., the back and forward buttons. Table 2.1 lists an example of navigation properties.

Property	Description
d1	The page is reachable from the top page and always has a next page in the transition.
d2	Every page is reachable from the top page.
d3	The top page is reachable from all pages.
d4	Eventually a chosen-page is visited.
d5	The first page is the login-page and the next page is either the login-error-page or the home-page.
d6	Whenever the login-page is visited, the next page is either the login-error-page or the login-success-page.

TABLE 2.1: Navigation Properties [Stock et al., 2014].

The work of [Homma et al., 2011] uses the SPIN model checker [Holzmann, 2004] to model web application navigation properties. The authors propose a method to use

two finite state automata, with the first representing page transitions and the second modelling the internal state transitions of the web application.

[Castelluccia et al., 2006, Di Sciascio et al., 2003] modelled web applications as a directed graph in which pages, links, windows and actions are represented as states. The implemented prototype system embeds a component which automatically imports web applications design from a UML tool; and then Computational Tree Logic (CTL) specifications are added and translated as source code for the NuSMV model checker [Cimatti et al., 2000]. The main advantage of this method is the ability to perform a priori verification of the web application design by applying the verification process to the UML-design of the web application in a single automated process using the verification tool *Waver*.

[Ricca and Tonella, 2000, 2001] propose a model of web applications using a UML class diagram. The model is used for reachability checking and semi-automatic test case generation.

[Han and Hofmeister, 2006] present an approach that uses state charts to formally model the adaptive navigation of web applications and checks for unreachable web pages. This model only focuses on user mode (e.g., whether the user is logged in or not) and page history (e.g., what pages the user has previously visited).

The work of [Haydar et al., 2005] proposes a way to discriminate states of interest by introducing a specialised operator for Linear Temporal Logic (LTL), which is used to verify web applications. This focuses more on the distinction of states; rather than on the modelling of web applications.

In [Yuen et al., 2006], the authors propose a behavioural model of web applications, called Web Automata, which is based on the Model, View and Control (MVC) model architecture. They model the behaviour of a web application with dynamic content

as an extension of links-automata with the constraint logic feature of Extended Finite Automata (EFA). They also present a testing framework for web applications based on the behavioural model.

In [Haydar et al., 2004] the authors present a formal approach for modelling web applications using communicating automata. They observe the external behaviour of an explored part of a web application using a monitoring tool. The observed behaviour is then converted into communicating automata representing all windows, frames and frame sets of the application being tested by intercepting HTTP requests and responses using a proxy server. Their model differs from the one proposed in this research, as it focuses on external behaviour.

The approaches described in this section use either the graph-based model or UML to represent the navigation properties of a web application. UML is considered as the modelling standard for a wide range of applications and systems [Alalfi et al., 2009]. However, UML is not a suitable method for the verification of web applications as the models need to be translated into formal specifications. The alternative method is to use graph-based modelling methods that can be directly translated to a verification form that is accepted by model checking tools. From the research listed in this section, we identify the need for a sound method that also includes the dynamic behaviour of web applications.

2.3 Web Security Properties

Since web applications are developed with availability across the Internet their security is a major concern for developers and users [Huang and Lee, 2005, McClure et al., 2003, Tracy et al., 2002]. Web application developers review web application

vulnerabilities regularly. The Open Web Application Security Project (OWASP) is concerned with web application security, and publishes a list of the most recent attacks of web applications each year [OWASP, 2013], as well as general guidance for building and verifying web applications. In addition, technical reports are published by other organisations, such as Microsoft, which focus on developing secure web applications using the .NET framework [Microsoft, 2011]. The Web Applications Security Consortium (WASC) published a report on security threat classifications [WASC, 2011], summarising the most common security threats of web applications.

The modelling of web behaviour properties is divided into two categories. The first category is *security properties*, which focuses on access control and session control mechanisms. Security properties are related to navigation properties. For example, a wrongly designed navigation link could lead to unauthorised access to sensitive information in the web application. The second category of web behaviour is *instruction processing properties*; this type of modelling addresses issues related to execution and state changes at both ends, without communicating with each other.

Predicting the kinds of attacks that could affect the security of a web application is challenging when observing the diversity of these attacks. However, by modelling specific web application properties, it is possible to model the cause and effects of the attacks [Corin et al., 2003].

According to a survey by [Li and Xue, 2014], the three primary security aspects that should be considered to achieve an accepted level of security are:

- User inputs are potentially dangerous and cannot be trusted in an open environment. Thus, *input validation* is an essential aspect of the web application

security to detect untrusted user inputs. Due to the unique features of web applications in contrast with other applications, input validation is a challenging task.

- It is equally as important to employ *session management* to correlate web requests from the same user into one web session during a certain period of *time*. Communication between a user and a server is carried out through HTTP, which is a stateless protocol. As a result, multiple inputs from the same user are processed as independent requests originating from multiple users of the web application. The session variables can be stored either at the client side (via cookies) or at the server side (using files or databases). In the latter case, a unique identifier *session ID* is assigned to index the explicit session variables, which are stored at the server side and issued to the client.
- Additionally, the implementation of *control flow* between the user and server must be accurate to protect sensitive information. This can be achieved explicitly through source code security checks or implicitly through the navigation paths presented to users. Security checks examine the state of a web application by relying on session variables and persistent objects in the database before revealing sensitive information to the user. Authentication and authorisation are the most common mechanisms for control flow in data-sensitive web applications, enabling an application to restrict its sensitive information and privileged operations from authorised users.

In this research, the focus is on the three primary aspects of *session management*, *authentication properties* and *control flow properties*. The remainder of this chapter describes and discusses the aforementioned security aspects and the most common attacks that can exploit vulnerabilities.

2.3.1 Session Management

Web application session management is essential to track and record user input and to maintain accurate application states. Session management is accomplished through collaboration between the client and the server. The simplest approach is for the server to send a unique identifier (i.e. session ID) to the client.

Since the session ID is the sole proof of the client's identity, its confidentiality, authenticity and integrity need to be secured in order to avoid session hijacking. First, a session ID should be randomly generated for each user's visit and should expire after a short period of inactivity timeout. Second, transmissions between the parties should be protected by a secure transport layer protocol (i.e. SSL security protocol), to ensure that attackers are unable to deduce the session ID and eventually control the session. Finally, the user should make sure that the session ID provided by the server is unique by not adopting a session ID from an external source. Attackers can set a session ID to a value that is known to them.

A web session is formed as a pair of network HTTP request and response transactions associated with the same user. Complex web applications require the retention of information or status about each user for the duration of multiple requests [Stock et al., 2014]. Therefore, sessions provide the ability to establish variables, such as access rights and localisation settings, which will apply to each interaction between the user and the web application for the duration of the session.

Web applications create sessions to keep track of anonymous users after the very first user request. An example is saving the preferences of the user's language. In addition, web applications make use of sessions once the user has been authenticated. The process ensures the ability to associate the user to any following requests, and

also employs security access controls, enables authorised access to the user’s private data and enhances the usability of the application.

Table 2.2 lists examples of the session management properties:

Property	Description
b1	Verify that sessions are invalidated when the user logs out.
b2	Verify that sessions time-out after a specified period of inactivity.
b3	Verify that the application does not permit duplicate concurrent user sessions, originating from different machines.
b4	Verify that sessions time-out after an administratively-configurable maximum time period regardless of activity (an absolute time-out).

TABLE 2.2: Session Management Properties [Stock et al., 2014].

2.3.2 Authentication

Authentication is the process of verifying that an individual or entity is who they claim to be. Authentication is commonly performed by submitting a user name or ID and one or more items of private information that only a given user should know [Stock et al., 2014]. A session record is then created with a cookie set, which the browser will send with each subsequent request to the application. The application then shows data related to the authenticated user (e.g., shopping cart content, posts, and stored files) during their use of the application. Table 2.3 lists the most important web application (authentication properties).

Property	Description
a1	Verify that all pages and resources require authentication except those specifically intended to be public.
a2	Verify that all authentication controls are enforced on the server side.
a3	Verify that re-authentication is required before any application-specific sensitive operations are permitted as per the risk profile of the application.
a4	Verify that a failure of the authentication controls ensures that attackers cannot log in.

TABLE 2.3: Authentication Properties [Stock et al., 2014].

2.3.3 Control-Flow

Each web application maintains its own application control flow (also known as business logic). Ensuring the correctness of the control flow is key to a secure web application, and this mainly depends on the intended functionality of the application. The main logic property is that users can only access authorised information and perform operations allowed by the intended work flow of the web application [Li and Xue, 2014].

Web developers attempt to prevent such vulnerabilities. The interface-hiding mechanism, which uses the principle of *security through obscurity*, has been widely used as an access control mechanism in web applications. However, this mechanism alone is not sufficient to ensure the control flow of web applications. Attackers can simply expose hidden links to access unauthorised information and operations. Secondly, developers may manually use explicit security checks prior to all sensitive operations.

It is difficult to check and anticipate all possible execution paths that may lead to a security vulnerability. It is likely that there will be a missing security check on certain paths that will lead vulnerabilities to be exposed to attackers.

As discussed above, control flow vulnerabilities depend on the intended purpose of a web application. For example, an online banking website may have a certain vulnerability that allows attackers to bypass vital security pages or steps. The 2013 OWASP report states that top ten security risks for web applications [Stock et al., 2014] can be attributed to application logic vulnerabilities (i.e. missing functional access control, invalid redirects and/or forwards). Table 2.4 lists control-flow vulnerabilities.

Property	Description
c1	Verify that the application does not allow spoofed high value transactions, such as allowing Attacker User A to process a transaction as Victim User B by tampering with or replaying a session, transaction state, transaction or user IDs.
c2	Verify that the application is protected against information disclosure attacks, such as direct object reference, tampering, session brute force.
c3	Verify that the application will only process business logic flows in sequential step order, with all steps being processed in realistic human time, and restricting out of order, skip steps, process steps from another user, or submitting transactions too quickly.

TABLE 2.4: Control-Flow Properties [Stock et al., 2014].

2.4 Model Checking

Model checking is based on a collection of techniques for the automatic analysis of a system. A formal definition of model checking is

“ Model checking is an automated technique that, given a finite state model of a system and a formal property, systematically checks whether this property holds for (a given state in) that model [Baier et al., 2008].”

The model checker tool takes as input the description of the system and the properties of that system. The system, in most cases, is defined as a finite state system, and its properties are expressed as temporal logic formulas. The model checker verifies whether the properties hold or not. In most cases, if a property does not hold, the model checker provides a counterexample.

In practice, the model of the system being analysed is approximate, thus the results are limited. Errors in the model may still remain after the verification. When applying model checking to a system’s design, three main phases may be identified, as described in [Baier et al., 2008, Clarke, 2008, Clarke et al., 1999]:

- *Modelling Phase.* The modelling phase consists of modelling the system in a language acceptable to the one used by the model checker, then using the simulation on the model and finally using the property specification language to formalise the property to be checked.
- *Running Phase.* The system is checked to see if the properties defined using the model checker hold.

- *Analysis Phase.* This phase checks whether the properties specified are satisfied or not. Depending on the result, the model is then refined, the properties are re-designed and the process is repeated.

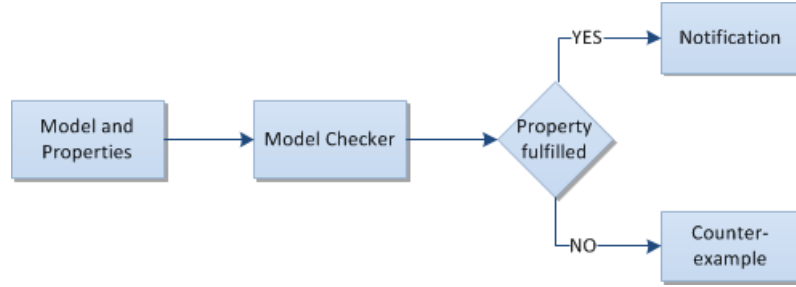


FIGURE 2.3: Model Checking Process.

Figure 2.3 gives an overview of the model checking approach. The requirements of the system under consideration are first identified and these requirements are then formalised in a property specification language. The system is then modelled in a language acceptable to the model checker. A combination of the model and the properties of the model are fed into the model checker. The model checker outputs the results as ‘satisfied’ if no property is ‘violated,’ or ‘violated’ if a property fails. To build any model for verification purposes, there are guidelines to be followed, in order to correctly model the system under consideration.

The problem of model checking can easily be stated as defined by [Clarke, 2008]:

Let M be a Kripke structure [Kripke, 2007] which is graphical transition system that represent the behaviour of a system . Let f be a formula of temporal logic (i.e. the specification). Find all states s of M such that $M, s \models f$ (see Figure 2.3). The term *model* refers to whether the structure M was a *model* for the formula f . It does imply the abstract model of the system under study.

Some of the main advantages of model checking over other verification techniques, such as automated theorem proving or proof checking [Clarke, 2008], are:

- The checking process is automatic. Users need to enter a formal description of the system model and its specifications and the model checking tool will produce a result.
- Model checking is faster than traditional verification techniques (i.e. testing), and therefore it saves time and expense.
- The model checker produces a counterexample if the specification is not satisfied. Counterexamples are important to show the reason why the specification does not hold; this assists in the debugging of complex systems.
- It can evaluate partial specifications. In complex systems, model checking can be used during the design phase.
- Temporal logic can express essential properties for reasoning about concurrent systems.

In contrast, one of the main disadvantages of model checking is the *state explosion problem*, since model checking searches the state space of a system, which may increase exponentially with the system's description size.

This problem comes from having so many possible, interleaved interactions between processes that the state space grows exponentially compared to the number of processes. If such behaviour is inherent to the system, the only way out of it is to use bit-state hashing, which implies only partial checking of the state space. This is based on the idea that the presence of errors is easier to detect than their absence [Bosnacki, 1998, Clarke, 2008]. Partial-order reduction, which is used by the SPIN model checker, is one of the most effective solutions to this problem [Clarke, 2008]. Partial-order reduction means that instead of generating all possible, interleaving execution paths in the state space, it is possible to generate only representatives of certain classes of

execution paths. As [Holzmann, 1997] describes, the reduction is based on the observation that the validity of an LTL formula is often insensitive to the order in which concurrent or independently executed events are interleaved in the depth-first search.

This makes it possible to record state changes and in this way to ascertain that two different paths of execution are the same, which in turn enables the verifier to remove the other paths, as they would not contribute anything new to the verification. In contrast, state compression means simply compressing the state data, which naturally incurs runtime overhead. Both partial-order reduction and state compression are guaranteed not to make the system states unreachable [Holzmann, 1997].

2.4.1 Model Checking Tools

Model checking tools are being developed continuously in both industry and the research community. Tools such as PROVERIF [Blanchet, 2001], SCYTHER [Cremers, 2008], NuSMV [Cimatti et al., 2000] and Tamarin [Meier et al., 2013] are examples of model checking tools used for the verification of security protocols and can provide simulation and verification of their properties. Each of the tools has a different level of ability to model specific properties of the system under verification. Moreover, tools such as ProVerif provide support for modelling intruders and cryptographic primitives.

The SPIN model checker [Holzmann, 2004] is the primary tool used in this research; it was chosen for its simplicity and high degree of expressiveness. In contrast to other model checkers, SPIN has the ability to provide insights into the first stages of modelling through its simulation charts. Since SPIN does not support the modelling of time we integrate discrete time into the models.

The second model checker used in our research is UPPAAL [Amnell et al., 2001]. UPPAAL is used for the verification of real-time systems. The graphical editor of UPPAAL makes it easy to design a system model, along with a graphical simulator that shows the possible dynamic behaviour of a system description.

2.5 Temporal Logic

Temporal logics are used to describe event sequences in time without the explicit use of time. Temporal logics were developed to investigate how time is used in natural language arguments by philosophers and linguists [Clarke, 2008, Hughes and Cresswell, 1996]. This work uses two model checking tools: the SPIN model checker uses LTL, while the UPPAAL model checker use CTL for verification [Burstall, 1974, Kröger, 1977, Pnueli, 1981].

The usage of temporal logics for reasoning about systems was proposed by [Burstall, 1974, Kröger, 1977, Pnueli, 1981]. The two most-used branches of temporal logic are the *Linear-time Temporal Logic* (LTL) and the *Computation Tree Logic* (CTL). LTL considers every event in time as having a unique possible future; the events are checked over a classical timeline. In contrast, CTL expresses each moment in time as having several possible futures. Thus, CTL views the structure of time as a tree, rooted in the current time with any number of branching paths from each node of the tree. LTL and CTL have a common subset of properties, but neither of them completes the other one. Properties exist that are expressible in LTL but cannot be expressed in CTL, and vice versa. CTL* is another temporal logic that contains both LTL and CTL.

Commonly, three different types of properties are distinguished in verification:

Safety Property: Describes a behaviour that may not occur on any path ("Something bad may not happen"). In order to verify a safety property, all execution paths need to be exhaustively checked.

Invariance Property: Describes a behaviour that must hold on all paths.

Liveness Property: A liveness property implies that "something good eventually happens", and a certain state will always be reached in a system.

2.5.1 Linear Temporal Logic (LTL)

Linear Temporal Logic (LTL) [Fisher, 2011, Holzmann, 1997, Venema, 2001] reasons over linear traces through time. At each time instant, there is only one real future state that will occur. Traditionally, that timeline is defined as starting "now," in the current time step, and progressing infinitely into the future.

Syntax of LTL formulas are composed of a finite set $Prop$ of atomic proposition variables (denoted by letters p, q, \dots), the Boolean connectives $\neg, \wedge, \vee, \rightarrow$ and the temporal connectives \mathcal{U} (until), \mathcal{X} (next time), \mathcal{G} (globally, also known as the \Box sample) and \mathcal{F} (eventually, also known as the \Diamond sample). Intuitively, the $\mathcal{X}\varphi$ states that φ is true in the next time step after the current one. The $\varphi\mathcal{U}\psi$ states that either ψ is true now or φ is true now and φ remains true until such a time when ψ holds. Finally, $\Box\varphi$ means that φ is true in every step, while $\Diamond\varphi$ designates that φ must either be true now or at some future time step. Table 2.5 lists all the operators used in LTL formulas and their equivalent used in the SPIN model checker verification:

Formally, an LTL formula φ has the following syntax, where p is an atomic proposition from some set $atoms$:

<i>Operator</i>	<i>Math or Logic</i>	SPIN
not	\neg or \bar{p}	!
and	\wedge	&&
or	\vee	
implies	\rightarrow	- >
equivalent	\leftrightarrow	< - >
until	\mathcal{U}	U
always or globally	\square or \mathcal{G}	[]
eventually or in the future	\diamond or \mathcal{F}	<>
next	\mathcal{X}	X

TABLE 2.5: LTL formula operators with their mathematical and SPIN notation

$$\varphi ::= p \mid \neg\varphi \mid \varphi \wedge \varphi \mid \varphi \vee \varphi \mid \mathcal{G}\varphi \mid \mathcal{F}\varphi \mid \mathcal{U}\varphi \mid \mathcal{X}\varphi$$

For example, we verify a navigation property “the home page is always followed by an account page” in a web application model. We use the LTL formula in 2.1 where p is defined in the SPIN model checker as the home page and q as the account page and \diamond means eventually.

$$([\![p - > <> q]\!]) \tag{2.1}$$

More properties will be expressed in LTL later in Chapter 3.

2.5.2 Computational Tree Logic (CTL)

Computational Tree Logic (CTL)[Fisher, 2011] is a branching time logic that reasons over many possible traces through time. Unlike LTL, for which every time instance has exactly one immediate successor, CTL has a finite, non-zero number of immediate

successors. CTL was the first logic to be used in model checking [Clarke and Emerson, 1982]. The CTL branching timeline starts in the current time step and may progress to any one of potentially many possible infinite futures. Furthermore, in reasoning along a timeline, CTL operators must also reason and include all possible branches. CTL is similar to LTL, in so far as the temporal operators are all two-part operators, with one part specifying the location to occur along a future timeline and another specifying whether this action takes place on at least one branch or all branches. The path operators are:

- **A**: On all future paths, starting from the initial state.
- **E**: On some future paths, starting from the initial state.

The second model checker used in this research, UPPAAL, uses a simplified subset of CTL. The properties of web applications are expressed in CTL.

For the purpose of understanding the logical expressiveness of LTL and CTL in verifying web application properties, we consider the differences between the logics. [Vardi, 2001] states that the linear and branching time logics correspond to two distinct views of time, and therefore LTL and CTL are expressively incomparable. In general, CTL allows explicit existential quantification over paths, which gives it an expressive nature in cases where there is a need to reason about the possibility of the existence of a specific path through the transition system Model M . This includes instances such as when M is best modelled as a computation tree, such as the dynamic nature of multi-pages from a single page. For example, there are no LTL matches of the CTL formulae $\mathbf{A} \mathbf{X}p$, since LTL cannot express the possibility of p occurring on some path but not all paths next time or in the future. Moreover, it is impossible to express in LTL scenarios where distinct behaviour on distinct branches occurs at the same time.

On the other hand, it is difficult to use the CTL logic in situations where the same behaviour may occur on distinct branches at distinct times; here the ability of LTL to describe individual paths is important and useful. [Vardi, 2001] explains that the former rarely happens, and LTL is found to be more expressive in other ways than CTL.

A further comparison between the logics will be discussed in Chapter 5.

2.5.3 Temporal Logic Patterns

Temporal logic formalisms are commonly used to describe states and event sequences in systems. Defining temporal formulas can be a straight-forward process if the property is small. However, when there is a need to verify complex properties, it is advisable to use the formula patterns described in [Dwyer et al., 1999, Salamah et al., 2005].

Table 2.6 shows the temporal patterns that were collected from research studies:

<i>Pattern</i>	<i>Description</i>
Absence	A given state q does not occur within a state scope.
Existence	A state q must occur within a scope.
Bounded Existence	A given state q must occur a number of times in a scope.
Universality	A state q must occur through the scope.
Precedence	A given state p must always be preceded by a state q within a scope.
Response	A given state p must always be followed by a given state q within a scope.

TABLE 2.6: Temporal Logic Patterns

[Corbett et al., 2000] developed pattern scopes, where the execution of the temporal formula takes place in a specified region of the scope. There are five basic scopes where a pattern can hold:

- Globally: A given state must hold throughout the system's execution.
- After: A given state p must occur after the first occurrence of a state q .
- Before: A given state p must occur before the first occurrence of a state q .
- Between: A given state p must occur between a pair of designated states.
- After and until: The state must occur after one state until the next occurrence of another state.

The use of the temporal logic patterns assists in reducing the complexity of using temporal logic formulas. We use the patterns in verifying properties in SPIN in Chapter 3.

2.6 Timed Automata Theory

The second model checker used in this research, UPPAAL, is based on the theory of timed automata, hence the importance of introducing the theory in this section.

Timed automata theory [Alur and Dill, 1994, Bengtsson and Yi, 2004, Kourkoui and Hassapis, 2005] is defined as a formal framework for modelling and analysing the behaviour of real-time systems. Real-time systems are described as systems that must fulfil time constraints, such as deadlines, response time, communication delays and execution. In secure web applications, time constraints are crucial to deliver security to both the user and the web application. Timed automata are finite-state directed

graphs extended with real clock variables; time can elapse when it is in a state. When a transition occurs, some of the clocks may be reset to zero, and the reading of a clock is equal to the time that has elapsed since the last time the clock was reset.

Clock constraints, called guards, are used with each transition to restrict the behaviour of the automaton. A transition represented by an edge can be accepted when the clock's values satisfy the guard labelled on the edge. A clock constraint is also associated with each location of automaton; this is called the *invariant* of the location. Time can elapse in the location only as long as the invariant of the location is true.

Timed automata theory can assist in reachability analysis by determining a final state or a set of final states; the analysis should determine if those states are reachable by means of triggering transitions from the initial state.

The rest of this section discusses formal definitions of the syntax and semantics for timed automata based on [Behrmann et al., 2004, Bengtsson and Yi, 2004].

2.6.1 Formal Syntax

Let X be a set of clock variables, then the set $\Phi(X)$ of clock constraints ϕ is defined by the grammar $\phi ::= x \sim c \mid \phi_1 \wedge \phi_2$, where $x \in X$, $c \in \mathbb{N}$, and $\sim \in \{<, \leq, =, \geq, >\}$. A clock interpretation ν for a set X is a mapping from X to \mathbb{R}^+ where \mathbb{R}^+ denotes the positive real numbers including zero. A clock interpretation ν for X satisfies a clock constraint ϕ over X , denoted by $\nu \models \phi$, if and only if ϕ evaluates to *true* with the values for the clocks given by ν . For $\delta \in \mathbb{R}^+$, $\nu + \delta$ denotes the clock interpretation which maps every clock x to the value $\nu(x) + \delta$. For a set $Y \subseteq X$, $\nu[Y := 0]$ denotes the clock interpretation for X which assigns 0 to each $x \in Y$ and agrees with ν over

the rest of the clocks. We let $\Gamma(X)$ denote the set of all the clock interpretations for X .

Definition 1 (Timed Automaton) A timed automaton then is a tuple $(L, l^0, \Sigma, X, I, E)$, where

- L is a finite set of locations.
- $l^0 \in L$ is the initial location.
- Σ is a finite set of labels.
- X is a finite set of clocks.
- I is a mapping that labels each location $l \in L$ with some clock constraints in $\Phi(X)$ (the location invariant).
- $E \subseteq L \times \Sigma \times \Phi(X) \times 2^X \times L$ is a set of edges.

An edge $(l, a, \phi, \lambda, l')$ represents a transition from location l to location l' on the symbol a . The clock constraint ϕ specifies when the edge is enabled and the set $\lambda \subseteq X$ gives the clocks to be reset with this edge. The semantics of a timed automaton $(L, l^0, \Sigma, X, I, E)$ are defined by associating a transition system with it. A state is a pair (l, ν) , where $l \in L$, and $\nu \in \Gamma(X)$ such that $\nu \models I(l)$. The initial state is (l^0, ν^0) , where $\nu^0(x) = 0$ for all $x \in X$. There are two types of transitions (let $\delta \in \mathbb{R}^+$ and let $a \in \Sigma$). First, $((l, \nu), (l, \nu + \delta))$ is a δ -delay transition iff $\nu + \delta' \models I(l)$ for all $0 \leq \delta' \leq \delta$. Second, $((l, \nu), (l', \nu'))$ is an a -action transition iff an edge $(l, a, \phi, \lambda, l')$ exists such that $\nu \models \phi$, $\nu' = \nu[\lambda := 0]$ and $\nu' \models I(l')$.

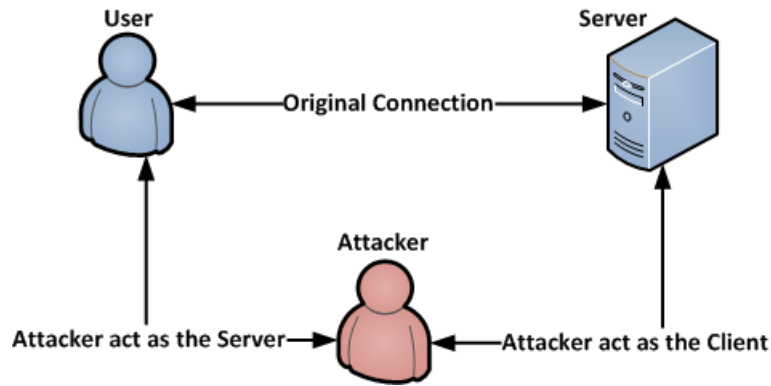


FIGURE 2.4: Attack Example

2.7 Modelling an Intruder (Man in the Middle)

In this section, we provide a description of the attack model used later in the modelling of a compromised system in the SPIN and UPPAAL chapters.

An intruder can tamper with the contents of individual messages during their transmission from the source to the destination. In the presence of such an intruder with certain rights, the aim is to validate whether the security specifications under study behave in a secure manner. Such tampering may go undetected; that is, the attacker may disguise itself to impersonate the trusted source. The attacker may tamper with the message to achieve denial-of-service (DoS) or replay attacks, in order to initiate its own session with one of the parties. Figure 2.4 illustrates the concept in more detail. In this figure, The user sends messages to the server. An attacker intercepts the messages, tampers with them and then forwards the modified messages to the server. This kind of tampering may go undetected, resulting in attacks, such as denial-of-service and replay attacks.

It is assumed that the attacker in this model has the abilities specified by [Dolev and Yao, 1983]. These are the ability to:

- Overhear and intercept all the messages over the network.

- Modify the messages.
- Generate new messages using information from overheard messages and some prior information.
- Send a new or captured message to another entity in the system.

The intruder collects knowledge about both parties. The knowledge is used during a session to create and send the intruder's own messages or to tamper with other agent communication.

SSL security protocol [Frier et al., 1996, Rescorla, 2001] is considered secure. However, intruders now attack the communication process not the SSL protocol itself. The process is used to attack the transition from an unsecured connection to a secure one, in this case from HTTP to HTTPS, where the intruder is attacking the bridge and can act as man-in-the-middle in an SSL connection before the communication occurs. Tools such as the *SSLstrip* assist in breaking the communication [Elks, 2011, Gangan, 2015].

2.8 Conclusions

This chapter discussed the main concepts of our research. The chapter provided the background for later chapters. The chapter can be summarized as follows:

- In Section 2.1, we discussed web application *development challenges*. The dynamic nature of web applications was identified as the main challenge. In Chapter 3, we show how it is possible to model the dynamic navigation of web applications.

- Section 2.2 identified *navigation properties* of web applications and discussed related work. We distinguished our research from related work by verification of both static and dynamic properties in a simple model.
- In Section 2.3, we discussed web application *security properties*. Session management, authentication and control flow vulnerabilities were analysed with a focus on the main properties.
- *Model Checking theory* was presented in section 2.4, with a brief presentation and comparison with other verification techniques.
- Section 2.5 is a primary section of our research. The *temporal logic branches* LTL and CTL will be used to verify properties in the following chapters. We discussed and compared both logics. We also listed the key logic patterns in Section 2.5.3 that will assist in verifying complex properties.
- Section 2.6 presented the *timed automata theory* and its formal syntax.
- Since we use an intruder to show the differences in the sequence of actions and timing in our models, Section 2.7 discussed how an *intruder is modelled*.

The next chapter presents the first model checking tool, SPIN, and describes how it is used to model and verify a web application.

Chapter 3

Modelling in SPIN

This chapter discusses the SPIN model checker used to model a web application and verifies the properties discussed in Chapter 2. Firstly, an overview of SPIN and its input language PROMELA is given. Secondly, as SPIN does not support the modelling time concept, the integration of time into PROMELA is discussed, along with examples from existing literature. There is then a detailed modelling of the web application example (online banking web application). Finally, the properties are verified and there is a discussion of the verification options in SPIN.

3.1 The SPIN Model Checker

SPIN is an explicit state model checker, developed by Gerard J. Holzmann [[Holzmann, 2004](#)] for verifying communication protocols. The specification language used for SPIN is called PROMELA. SPIN can be used in two methods, as a simulator and as a verifier. In the former, SPIN provides the advantage of creating a quick impression of the system behaviour as it is being built. SPIN uses a graphical user interface (ISPIN) in which a

model can be visualised during the simulation runs, and therefore assists in debugging the model from the first stages. Moreover, when the verifier is used, the simulator displays the error trace if an error is found. To tackle and reduce the state explosion problem; SPIN provides advanced features:

- partial-order reduction in order to reduce the number of reachable states;
- bit-state hashing;
- state compression in order to reduce the amount of memory needed to store all the states.

The SPIN model checker has three main components:

Simulator In the first stages of modelling a system, the simulator has three options to check the correctness of the model: random, interactive or guided simulation. With a random simulation SPIN chooses non-deterministically among all executable statements which one to execute next. Whereas in an interactive simulation, SPIN gives the user the option to choose from all possible statements. A guided simulation option is only activated when there is an invalid verification, i.e. if a property is not satisfied, the guided simulation will show the trail that leads to the counterexample.

LTL Translator An LTL formula is used to define a property φ , that should hold for a given system model. The SPIN verifier will, at this point, show the opposite, meaning the negated LTL formula $\neg\varphi$ is translated into a **never** claim to verify the model. Propositional variables in an LTL formula can only be variables of the Boolean type, which are defined as global variables or symbols in the model.

Verifier Generator The verifier generates a C program for the model that is checked by the program `pan` for counterexamples to a specified property φ

3.1.1 Promela

The PROMELA modelling language is a non-deterministic, multi-process language. The name PROMELA is an acronym for **Process Meta-Language**. It is not meant to be an implementation language, but rather a model description language. The emphasis therefore is on modelling on process synchronisation and coordination, and not on computation. This section is not meant to give a complete overview of PROMELA, but rather a brief introduction to enable the following implementation models to be understood. A further explanation of the language can be found in [Holzmann, 2004].

The basis building of PROMELA consists of asynchronous processes, buffered and unbuffered message channels, synchronising statements and structured data.

In order to create a model in SPIN, first a process is declared. Processes are the only **active** components in PROMELA; messages and variable states are only changed or inspected by a process. A process is declared using the keyword `proctype`.

```
1 proctype Client() {  
2 /* code for Client */  
3 }
```

This process type is named `Client`, and in the declaration body a list of local declarations and/or statements are added. A process is either executed by the call `run` or by adding the prefixed **active** statement before the `proctype` statement.

Processes communicate with each other through channels and global variables. There are four basic variable types: bit or bool (1 bit), byte (8 bits), short (16 bits), and

int (32 bits). Variables can be defined as global or local within each process. It is recommended to use a data type with as few bits as possible to reduce the state space during the verification phase. The variable type `mtype`, can be used to define the symbolic names of numeric constants. A `mtype` is used in a web application model to identify each message and to keep track of the sequence of communication. Table 3.1 shows the operators used in PROMELA. The symbols can be defined using (`define`).

The scope of a symbol is global, and an expression is a fixed number, a boolean expression, or an expression that assigns a value to the symbol.

Channels in PROMELA are used for processes to communicate with each other. A channel is defined using the keyword `chan`. There exist two types of channel in PROMELA, asynchronous or synchronous (rendezvous): The asynchronous channels are buffers that can store a finite number of elements. The synchronous channels have a length of zero. A channel is blocked unless there is a message waiting and the message matches the pattern of the receive statement. Likewise, writing to a channel is blocked if the channel is full. In the case of a synchronous channel, reading from it will always be blocked until the sender is ready to send, and the channel is always full until the receiver is ready to receive. The capacity of the buffer and the type of message can be given as parameters when the channel is declared. For example:

```

1 chan network = [0] of { mtype }
2 network!login      /*send the message "login"*/
3 network?login      /*receive the message login*/
4 network?x          /*receive the message, and store in variable x*/
5 network?eval(x); /*receive the message, only if it contains x*/
6 network?_          /*receive the message, and do not store it*/

```

In this Promela code, a channel called `network` is initialised as a synchronous channel with a length of zero. The message type is `mtype`. To send or to receive a message the symbols `!` and `?` are used. If a process sends a message, the other process can

receive this message and has the option to store its value to **x** of the same type **mtype**. With **network ? eval(login);** the message is only received and removed from the channel if the message's content is equal to the value of the variable **x**. The predefined write-only **_** is used to receive a message if the value of the message is irrelevant.

Operators	Associativity	Comment
()[] .	Left to right	Parentheses, array brackets, field selection
! ~ ++ -	Right to left	Negation, complement, increment, decrement
* / %	Left to right	Multiplication, division, modulo
+ -	Left to right	Addition, subtraction
<<>>	Left to right	Left and right shift
<<= >>=	Left to right	Relational operators
==, !=	Left to right	Equal, unequal
&	Left to right	Bitwise and
^	Left to right	Bitwise exclusive or
	Left to right	Bitwise or
&&	Left to right	Logical and
	Left to right	Logical or
- >:	Right to left	Conditional expression operators
=	Right to left	Assignment

TABLE 3.1: Operators in Promela.

In PROMELA, the control statements are **selection**, **repetition** and **jump**. SPIN executes the statements sequentially in the process body. If a statement blocks (evaluates to false), no other statement in the process can execute until the statement evaluates to true, which may be the result of a variable assignment or a sent message. In order to alter the process flow, PROMELA has two kinds of loop constructs: selection and repetitive constructs. The constructs consist of one or more option sequences. A sequence begins with a guard, and if the guard evaluates to true, the option sequence is chosen for execution.

Selection Construct A selection construct starts with the keyword **if** and ends

with the keyword **fi**. In between, several statements are defined. Each statement starts with a double colon, followed by a statement called **guard** that either evaluates to true or false, an arrow, and then a sequence of statements.

An example of a simple selection construct:

```
1 if  
2 :: (statue != HomePage) -> option 1  
3 :: (status == HomePage) -> option 2  
4 fi
```

Here, at most one option from the list will be executed. If more than one guard is true, the selection between them then becomes non-deterministic. A guard can be any PROMELA statement or expression which blocks until at least one guard is true. In the case that all the guard conditions in a loop construct evaluate to false, then there is no available option for execution and the execution process blocks. A special guard **else** statement is used in selection or repetition structures to define a condition that is true, if and only if, all other guard conditions in the sequence structure evaluate to false. In contrast to the predefined keyword **timeout**, **else** provides an escape clause at the process level, while **timeout** does so at the system level.

Repetition Construct The repetition construct starts with the keyword **do** and ends with **od**. It has the same syntax as the selection statements with guards and is followed by different statements. After the sequence of statements executed is finished, the loop is automatically repeated from the start, while the selection construct moves on to the next statement. The repetition construct can be broken by either transferring control explicitly with a **goto** statement or by executing a **break** statement.

An example of a simple repetitive construct:

```
1 do
2 :: (timer != 0) -> timer ++
3 :: (timer == 0) -> break
4 od
```

The body of the loop in this example will stop when the process reaches zero. If (timer == 0) the loop is exited. In the case of (timer != 0) the process increments the `timer`, and then continues around the timer. The keyword `break` is used here to exit the loop and the process executes the statements following the keyword `od`.

Jump statements In PROMELA it is possible to use the prefixed keyword `goto` with a `label` to jump to another statement within the same process. Moreover, labels are needed for the LTL formulae at the verification phase and there are three special labels `end`, `progress` and `accept` which will be discussed later in the verification section. Labels are marked with a single colon and put in front of a selection or repetition construct. Here is an example as used in the web application model to check the guard statement and then to move to another label that represent a different (Page).

```
1 HomePage:
2 statement 1 ; statement 2 -> goto AccountPage
3 AccountPage:
```

In SPIN an internal location counter points in every process to the statement that executes next. As SPIN cannot execute two statements at the same time, in every step it chooses non-deterministically between one of the executable statements of all processes. After executing the first statement, SPIN chooses another possible statement from the same process. In some cases it is important to group a sequence of

statements into one logical group that is to be executed as one indivisible unit, non-interleaved by other processes. PROMELA provides two structures to avoid this: an **atomic block** or **d_step** block, which abstracts the statements into one state change. Neither allow the interference of other processes until the statement sequence has finished. This is useful for reducing the size of the state space and avoiding unnecessary and trivial state changes. The **atomic** allows non-determinism while the **d_step** can be executed significantly more efficiently by the verifier, but **d_step** does not allow non-deterministic selection.

3.1.2 Verification in Spin

SPIN offers the following ways to express correctness properties:

Assertions A basic assertion expresses invariants and can occur anywhere in the model in the form of: **assert(expression)**. Assertions are the only constructs that are checked during the simulation phase. As long as the expression provided is evaluated to a true or a non-zero integer value, the assertion statement has no effect on the simulation process. If the assertion is false, SPIN reports an error and exits.

Trace assertions are similar to basic assertions but apply to channels. They are used to monitor the event sequences of messages exchanged between the processes. These events do not generate a new behaviour but they are required to match send or receive events on the same channels in the model.

Never Claim A never claim is used to express finite or infinite system behaviour that should never occur. It can be generated automatically from an LTL formula. In contrast to trace assertions, never claims match boolean propositions on system

states, while trace assertions match event occurrences that can occur in the transitions between system states.

Meta labels PROMELA uses special labels with the `goto` keyword, and it is evaluated during the run of the verification phase. The possible label types are: `end`, `progress` and `accept`. The first label type is the *End-state labels* that are used in SPIN for the verifier to be able to distinguish between valid and invalid end-states. When a timeout occurs, end-states are evaluated. Statements can be made valid end-states by adding the label `end`. In this case all other end-states are considered to be invalid.

The *Progress-state* labels are used to specify a liveness property. These labels are used to mark a state where effective progress is being made in an execution. For example, when a sequence number is being incremented or when a valid message is accepted by a receiver in a communication protocol. The last label type is the *Accept-state labels* which are mostly used in *never claims* that are generated from LTL formulae. States can be marked as `accept` to ensure that when the model is verified, the state must be reached at least once.

PROMELA has no mechanism for expressing time related properties or clocks. The `timeout` statement is the only approach in this direction; its value becomes true when there are no executable statements in the system. There are attempts to model real time in PROMELA, which are presented in the next section with examples from the literature about integrating time constraints into SPIN.

3.1.3 Modelling Time in SPIN

The PROMELA language is designed to specify systems of asynchronous processes and for proving functional correctness of a given model. It abstracts from the behaviour of the processes' scheduler and from any assumptions about the speed of execution. If integrating time or clock variables to SPIN, this could lead to an increase in execution time and memory and may reach a state explosion problem. There are methods to simulate time in SPIN such as an abstraction of time using constraints that is true or not, or by adding a process that controls time.

An example is the work of [Bosnacki, 1998], that creates a set of macros at the start of the model which add discrete time. However, by adding an extra timer process, the system can be difficult to follow and even predicting the outcome and the number of states can increase rapidly. In addition, all actions that take time have to synchronize on time and the passing of time acts like a scheduler for the model; this makes the model hard to analyse. [Tripakis and Courcoubetis, 1996] extended SPIN with timed Büchi automata, and created an extension called RT-PROMELA. The drawback is that it uses an old version of SPIN (version 2.9) while the current version is 6.1.

The work of [Brinksma et al., 2002, Ruys, 2003] used the *Variable time advance* method. By creating a variable that simulates time and changes forward to the next moment in time when some event triggers a state transition, all intervening time is skipped.

In order to model realistic web application properties that include time, a model will need to include time constraints to represent time-out situations and to keep track of the sequence of time and actions between different models.

3.2 Modelling Web Applications in Spin

The rest of this chapter will show how we develop a model of a simple transaction between a client and a server. For this purpose we use the model design approach developed by [Holzmann, 2004]. In order to achieve a correct model and to reduce the level of complexity, we start with the basic structure that captures the essential characteristics of the model. We then evolve the model gradually to cover more properties. As a result, we can analyse the behaviour of the model after each addition. In the case that the complexity grows rapidly, we use the verification tool to check that the model is built accurately and the results are correct. Failing to follow such an approach could lead to a complex model and a time consuming process with no clear results.

For modelling a web application in SPIN we start modelling gradually and then add a function with each step. The structure of developing our model is as follows:

- The first model in section 3.2.1 will only contain one session of the client logging into his/her online banking account to make a simple transaction, as shown in Figure 3.1 and the model terminates at the end of the run. We examine the basic properties of authentication, navigation and session management at this stage of the model.
- In the second model in section 3.2.2 we model a dynamic page navigation where different input lead to different pages. We keep the model simple and verify further properties since now the client can logout and re-login, the sequence of actions will be different.

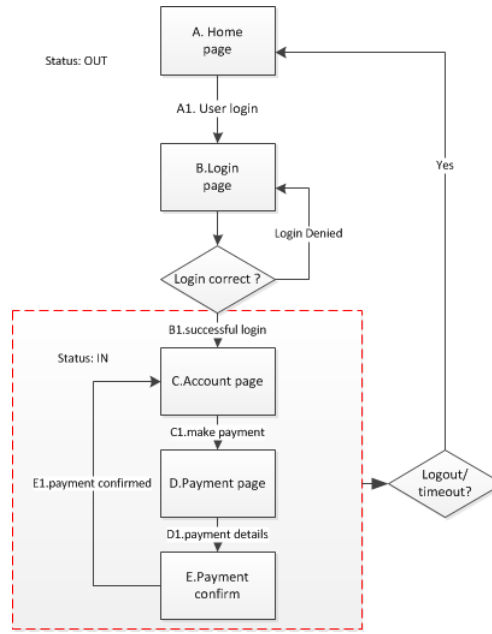


FIGURE 3.1: Model of Online Banking.

- In the third model we introduce a novel approach for modelling discrete time in SPIN. We add an extra process that controls the session, and we then verify properties that rely on timing, such as session management properties.
- The final model in section 3.2.4 includes an intruder that becomes active in the middle of the communication. We examine the difference in the sequence of actions and timing between a secure model and a compromised model.

Each model will include a description of the processes involved, and we show the outcome of the simulation phase of the first execution. We then show how to verify the properties in SPIN, and we will include the result for each property. The results from SPIN show the following:

- Property verification result, if the output result shows zero then there is no error found, but if the output shows one then the property does not hold for the model under verification.
- Depth reached.

- Number of States (stored and matched).
- Number of transitions.
- Total actual memory usage.
- Elapsed verification time.

In the verification phase in each model we use different approaches to verification as discussed in section 3.1.2 depending on the properties. Some properties can be checked by assertions in the code, while in other properties it is more effective to use LTL formulas.

3.2.1 Model without Timer

The first experiment of modelling a web application is without the addition of time constraints. This will assist in understanding the difference between both models (with timed constraints and without the notion of time) in terms of timing, modelling complexity, and verification results. In the first stage in explaining the process of the first model we explain the symbolic names (**mtype**) used to define the names of the range of messages used in the interprocess message exchange and all the variables that keep track of the client's location and status. Then, the specification of the client-side in PROMELA will be discussed. Finally, the possible properties' verification will be discussed.

In this example a model is created in PROMELA containing a client communicating with a server to enter his account and proceeding with a simple transaction (e.g. making a payment). At each stage of the communication a record of the client's location and status (i.e. logged-in or logged-out) will be saved to two global variables

(**Status** and **Page**) in order to be used for the verification when the properties are specified using LTL and for the flow control of the model.

```

1 mtype page = err;
2 mtype Status = err;
3 mtype partnerA = err;
4 mtype partnerB = err;
5 mtype statusA = err;
6 mtype statusB = err;

```

LISTING 3.1: Global variables of first model

The global variables shown in listing 3.1 are first used to track the message exchange and the status of both the client and server in the model, and also to assist in the authentication phase. Six variables are defined to keep track of the communication and to check the LTL properties in a later phase:

The first two global variables **Status** and **page** are used to keep track of the location of the client and the status of the communication (logged-in, logged-out and timed-out). **partnerA** is set to true when the client sends a request to login with the server. **PartnerB** is set to true when the client is known to the server. **StatusA** is set to true when the authentication is acknowledged by the server. **StatusB** is similar as the client is now authenticated to the server. We assign each of the variables to the symbolic name **err** to avoid being left uninitialised, the value of the variable is zero, and in this case it is outside the value range of possible **mtype** values.

Next we declare the symbolic names **mtype** as in listing 3.2; which is used to define the names of the range of messages used in an interprocess message exchange. The first set represent messages that the client sends in order to navigate within the web application. While the second set is used by the status variables **Status** and **Page**, and

is also used by the client process when creating new messages. SPIN parser assigns a unique positive integer value to each name which represents it internally. The **Status** and **Page** variables can take any value from the **mtype** declaration. To represent a simple abstract of an encrypted message, the **typedef** in line 5 is used to declare an abstract of an encrypted message **Crypt** that contains two entries: a session id and the content that represents the request and the response between the client and the server in the model. It will be used at the server side in order to check the content and to process an action.

```

1 mtype = {ok, err, ack, msg1, msg2, msg3, msg4, msg5,
2         Logout, sessionId, user, server};
3 mtype = {home, account, payment, confirm, logout, error, login
4         };
5 typedef Crypt { mtype session, content1}

```

LISTING 3.2: Symbolic names of first model

The network in listing 3.3 is modelled as a global single message channel that is shared by all on the network. In order to keep the model simple and to reduce the size of the model, synchronous communication is used which is shown by a buffer length of 0. We define a message on the network as triple that consists of a message number, the intended receiver, and the encrypted message **Crypt** which contains the values of the session id and the type of request.

```

1 chan network = [0] of {mtype, mtype, Crypt};

```

LISTING 3.3: Channel of first model

The first process in the model is the client process that initiates the communication with the server process as shown in listing 3.4. At the start, the client process is

declared with the prefix **active** defining that it starts automatically during the execution. At first the client assemble a message (**msg1**) that contains the session id and the client id to the server. This exchange represents the request to login to his account. If the server process replies with a (**ack**) it will then be checked using the condition as shown in line 13, that it is from the server and that the session id is the same. Then it will set the global variable **statusB** to **ok**. The authentication process is a simple abstract that only serves the model requirements.

```

1 active proctype Client()

3 Crypt messageUS;    /*    encrypted message to Server    */
4 Crypt data;         /* received encrypted message        */
5 A:
6 partnerA = server;

8 /* login message */
9 messageUS.session = sessionId;
10 messageUS.content1 = user;
11 network! msg1 (partnerA, messageUS);
12 network? ack (partnerA, data);
13 (data.session == sessionId) && (data.content1 == server)
14 statusB = ok;

```

LISTING 3.4: Client process of first model

In each message as in 3.5, the client sends his session id and the request for each page by changing the **messageUS.content1** to a page name as declared in the symbolic names set **mtype**, and then waits for a response from the server side. Since there is

no timer, at this stage it is only a normal session with no time-out. At the end of the session the client requests to logout and then the session terminates.

```

1 /*  Second message */
2 messageUS.session = sessionId;
3 messageUS.content1 = account;
4 network! msg2 (partnerA, messageUS);
5 network? ack (partnerA, data);

```

LISTING 3.5: Second message at the client process

Secondly, the server process receives the `client` messages and control the session. At this side of the communication, the server keeps track of the status of the client and the location at each page using the global variables we declared at the beginning.

Firstly, the first action at the `Server` process 3.6 shown in listing is labelled with a position label (`HomePage`) that represents the pages of the web application model. When it receives the first message and its contents (session id, user id) it then starts a selection statement (`if - fi`). The first statement after a double colon has a guard statement: `(data.session == sessionId)&& (data.content1 == user)`. If at this stage the client id or the session id is not correct then it will execute the `goto HomePage` jump statement and redirect the user again from the start. If it is successful, the variable `statusA` representing the authentication stage is set to `ok`, and the global variable `Status` is set to `login`. The global variable `page` is changed on each page to the current location, and first it is set to `home`. At the end of the statement the `goto AccountPage` jump statement is used to move to the next page.

```

1 active proctype Server()
2 mtype sessionId;
3 Crypt messageSU; /* encrypted message to the user */

```



```
4 Crypt data;          /* received encrypted message */
5 partnerB = user;
6 messageSU.session = sessionId;
7 messageSU.content1 = server;
8 HomePage: page = home;
9 network? msg1 (partnerB, data);
10 if
11     ::(data.session == sessionId) && (data.content1 == user) ->
12     network ! ack (partnerB, messageSU);
13     statusA = ok;
14     Status = login;
15     goto AccountPage;
16     ::else -> goto HomePage;
17 fi;
18
```

LISTING 3.6: Server process of first model

When the server receives the second message as in listing 3.7, it will start another selection statement (`if - fi`) that is similar to the first statement to check the session id, and the content of the message. At this stage the client requests to move to the account page. An assertion `assert (page == home && Status == login)` is made after receiving the second message that the page sequence is correct and the client is logged in.

```
1 AccountPage:
2 network? msg2 (partnerB, data);
3 assert ( page == home && Status == login )
```

```

5 if ::(data.session == sessionId)&&(data.content1 == account) ->
6     page = account;
7     network ! ack (partnerB,messageSU);
8     goto PaymentPage;

10 ::else -> goto HomePage;
11 fi;

```

LISTING 3.7: Second message of the server process

3.2.1.1 Simulation and Verification Results of Model without Timer

In order to understand the behaviour of the model and the properties the model captures, simulations are performed at the first stage. The simulation phase allows the detection of any early mistakes in the model design and any deviation from the expected behaviour of the model. First the model is checked for safety properties such as deadlocks or assertion violations. The following command produces a result that verifies that the model is deadlock free and there are no assertion violations. Figure 3.2 shows the result of the check. The verification run confirms that there are no problems and that the model is still simple with only 53 states. There were no unreachable states, meaning that the flow of our model is correct. The elapsed time shows zero, as the model is still simple and does not take time to verify.

```
$ spin -run -DSAFETY Session1.pml
```

The sequence chart in Figure 3.3 assists in the simulation stage to understand how the parties in the communication exchange messages. We can see that the messages start from `msg1` to `msg5` by changing the content of the message to the page name.

```

State-vector 40 byte, depth reached 51, errors: 0
  53 states, stored
  4 states, matched
  57 transitions (= stored+matched)
128.730      total actual memory usage
unreached in proctype Server
  (0 of 51 states)
unreached in proctype User
  (0 of 25 states)
pan: elapsed time 0 seconds

```

FIGURE 3.2: Safety Verification of Model without Timer

The result proves the correct sequence as we intended for the first model. Later when we add more to the model, the sequence chart will show how it is different.

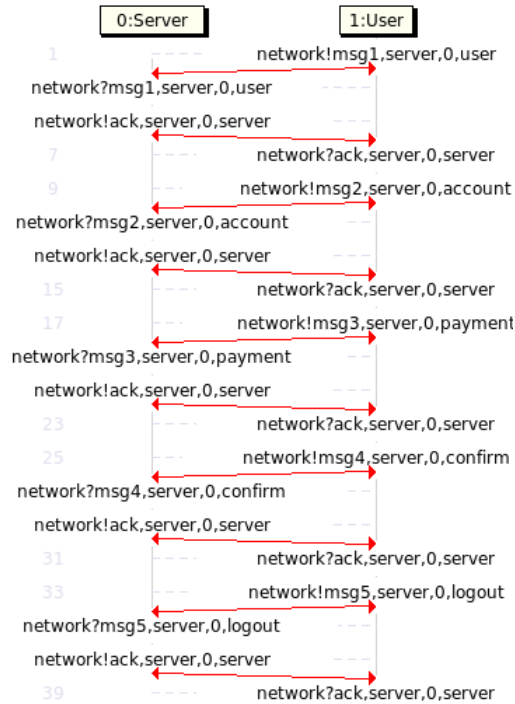


FIGURE 3.3: Message Sequence Chart of Model without Timer.

In listing 3.8 we verify the sequence of messages and the correctness of the model behaviour, by using the **trace** assertion to specify the order of messages exchanged between processes. We need to include all the network activity from sending and

receiving. In the case the order is incorrect, the verifier will show an assertion violation at the run result.

```

1 trace
2 {do
3 ::network!msg1;network?msg1;
4 ::network!ack;network?ack;
5 ::network!msg2;network?msg2;
6 ::network!msg3;network?msg3;
7 ::network!msg4;network?msg4;
8 ::network!msg5;network?msg5;
9 od;}

```

LISTING 3.8: Trace assertion for first model

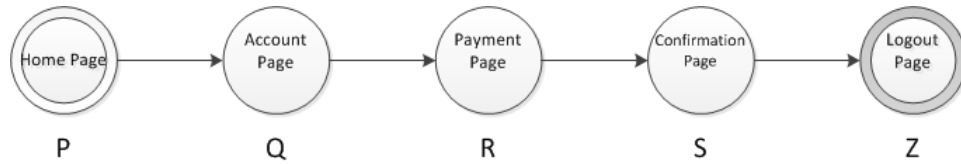


FIGURE 3.4: Page Sequence and LTL Proposition Letters.

For the next step, we use LTL to verify authentication, navigation and session management properties. We created Figure 3.4 to represent the sequence of the pages in the first model. This shows a finite number of pages that represent the correct navigation path. Each page is labelled with a propositional symbol. Each LTL formula will contain a condition based on a global variable, or a remote reference a label in the model that represent pages, for example `Server@HomePage` returns nonzero if and only if the location counter of the server process is at the point labelled by the `HomePage`. In listing 3.9 we define all the pages and give them a propositional symbol as shown in Figure 3.4.

```

1 #define p (Server@HomePage)
2 #define q (Server@AccountPage)
3 #define r (Server@PaymentPage)
4 #define s (Server@ConfirmPage)
5 #define z (Server@Logoutpage)

```

LISTING 3.9: Pages references at first model

The first LTL formula (3.1) is to verify an abstract authentication, and each of the parties in the communication need to ensure that they are communicating with the intended agent. Figure 3.5 shows the result that the formula is correct in the model. Since the authentication is at the start of the model, SPIN finds a result fast and the number of states stored is lower than in the first run. The transitions increased in the verification when SPIN generates a never claim automaton to verify the property.

$$\begin{aligned}
 &\langle \rangle (partnerA == server \ \&\& \ partnerB == user) - \rangle \\
 &\langle \rangle (statusA == ok \ \&\& \ statusB == ok)
 \end{aligned}
 \tag{3.1}$$

```

State-vector 48 byte, depth reached 26, errors: 0
    33 states, stored (49 visited)
    16 states, matched
    65 transitions (= visited+matched)
128.730          total actual memory usage
pan: elapsed time 0 seconds

```

FIGURE 3.5: Verification Result of Property 3.1

The next LTL formula 3.2 is another way to verify a successful authentication for the model. Figure 3.6 shows the result from the verification, and since the condition is at the start of the execution, the number of states is smaller than the formula (3.1).

$$< > (statusA == ok \&\& statusB == ok) \quad (3.2)$$

```
State-vector 48 byte, depth reached 26, errors: 0
    17 states, stored (34 visited)
    13 states, matched
    47 transitions (= visited+matched)
128.730          total actual memory usage
pan: elapsed time 0 seconds
```

FIGURE 3.6: Verification Result of Property 3.2

In LTL formula 3.3 we verify that the client is in a logged in status at the account page and cannot bypass the authentication process at the start. Also, the same LTL formula could be applied to all other pages. Figure 3.7 shows the result. Since the formula is more complex we notice that the depth reached 99 from the initial system state, and the number of transitions increased.

$$[]((page == account) - > < > (statusA == ok \&\& statusB == ok)) \quad (3.3)$$

```

State-vector 48 byte, depth reached 99, errors: 0

    59 states, stored
    6 states, matched
    65 transitions (= stored+matched)
128.730          total actual memory usage
pan: elapsed time 0 seconds

```

FIGURE 3.7: Verification Result of Property 3.3

In LTL formula 3.4 the position state label is used to ensure that the client is at status login when he is at the payment page. The symbol letter *r* refers to *Server@PaymentPage* as explained in listing 3.9. Figure 3.8 shows the results of the verification with an increase in the number of states, transitions and depth reached, due to the location of the page under verification.

$$[] (r) \rightarrow \langle \rangle \text{Status} == \text{login} \quad (3.4)$$

```

State-vector 48 byte, depth reached 112, errors: 0

    68 states, stored
    9 states, matched
    77 transitions (= stored+matched)
128.730          total actual memory usage
pan: elapsed time 0 seconds

```

FIGURE 3.8: Verification Result of Property 3.4

The next step is to verify navigation properties, and SPIN can check the sequence and order of pages in the model by using LTL patterns as discussed early in the background chapter, section 2.5.3. We use the response and precedence patterns to

verify the order of states (i.e. pages and session status). The first property we check is that the home page is always followed by an account page. We use the global response pattern in formula 3.5 where p is defined as the home page and q as the account page. Running a verification of this formula shows no errors and the number of transitions has increased compared to the previous results as shown in Figure 3.9.

The reason for the increase is that SPIN generates a more complex never claim automaton for the LTL formula 3.5. To understand how the never claim works, we can generate it using the command (`spin -f '[] (p -> <>q)'`). The result in Figure 3.10 shows the never claim that is included during the verification run of the LTL formula. The guard condition (1) in all the repetition loops is equivalent to the boolean value true, and (`||`) is a symbol of the logical operator (or). We do not generate a never claim for each verification as in the latest version of SPIN it is possible just to write the LTL formula at the end of the PROMELA code using the command `ltl name formula`. SPIN does not search for system executions that satisfy the property but in executions that violate it, meaning in property 3.5 we verify that p becomes true first and then q .

$$([](p - > <> q)) \tag{3.5}$$


```
State-vector 48 byte, depth reached 93, errors: 0
    71 states, stored (89 visited)
    25 states, matched
    114 transitions (= visited+matched)
    0 atomic steps
128.730          total actual memory usage
pan: elapsed time 0 seconds
```

FIGURE 3.9: Verification Result of Property [3.5](#)

```

never {    /* [] (p -> <>q) */
T0_init:
    do
        :: (((! ((p))) || ((q)))) -> goto accept_S20
        :: (1) -> goto T0_S27
    od;
accept_S20:
    do
        :: (((! ((p))) || ((q)))) -> goto T0_init
        :: (1) -> goto T0_S27
    od;
accept_S27:
    do
        :: ((q)) -> goto T0_init
        :: (1) -> goto T0_S27
    od;
T0_S27:
    do
        :: ((q)) -> goto accept_S20
        :: (1) -> goto T0_S27
        :: ((q)) -> goto accept_S27
    od;
}

```

FIGURE 3.10: Never Claim for Property 3.5

We can also use a specific precedence LTL formula pattern in 3.6 which means that the account page *q* cannot become true until the home page *p* is true. Both LTL formulas 3.5 and 3.6 could be applied to all other pages in the model by changing the proposition symbol letter. Though both formulas specify the same meaning of the

order of pages. We compare the differences in the number of states between a simple formula and a complex formula.

$$(!q U p) \tag{3.6}$$

```
State-vector 48 byte, depth reached 8, errors: 0
    7 states, stored (14 visited)
    4 states, matched
    18 transitions (= visited+matched)
    0 atomic steps
128.730          total actual memory usage
pan: elapsed time 0 seconds
```

FIGURE 3.11: Verification Result of Property 3.6

Figure 3.6 shows that the number of states and depth reached is smaller than the result in 3.5, though the formula serves the same purpose as pages order. The reason for this is that it finds that the home page p is true at the start of the execution and then it is followed by the account page. The number of transitions is also lower (18 transitions) since the never claim generated by SPIN has less checking loops, as shown in Figure 3.12.

In order to check that the LTL formula we used is correct, we can do the opposite and change the order of pages such as the account page comes before the home page in the LTL formula $!p U q$. When we run the verification, an error is reported in Figure 3.13 with an increase in the number of states, translations and depth reach.

```
never {    /* !q U p */
T0_init:
    do
        :: atomic { ((p)) -> assert(!((p))) }
        :: (! ((q))) -> goto T0_init
    od;
accept_all:
    skip
}
```

FIGURE 3.12: Never Claim for Property 3.6

When an error is reported SPIN produces a error-trail file. We use the trace file to locate the state where the property is violated, using the command (`spin -t -p file.pml`). The option (`-t`) refers to the trace file while (`-p`) prints all the statements from the start of the verification.

Since we kept the model small, this made it easy to verify and follow the sequence of actions. We know that the account page comes after the home page and we asked to prove that the home page becomes true after the account page. In Figure 3.14 SPIN shows that an assertion is violated after both parties in the model prepare the messages to send and the verifier finds that the server is at the home page.

```

State-vector 48 byte, depth reached 30, errors: 1

    19 states, stored (33 visited)

    13 states, matched

    46 transitions (= visited+matched)

    0 atomic steps

hash conflicts:          0 (resolved)

    128.730          total actual memory usage

pan: elapsed time 0 seconds

```

FIGURE 3.13: Verification Error Result of Property 3.13

```

spin -t -p Session1.pml

ltl m1: (! ((Server@HomePage))) U ((Server@AccountPage))

starting claim 2

using statement merging

Never claim moves to line 4          [(!((Server._p==AccountPage)))]

2:proc  1 (User:1) Session1.pml:115 (state 1)[partnerA = server]

4:proc  1 (User:1) Session1.pml:120 (state 2)[messageUS.session = sessionId]

4:proc  1 (User:1) Session1.pml:121 (state 3)[messageUS.content1 = user]

6:proc  0 (Server:1) Session1.pml:37 (state 1)[partnerB = user]

8:proc  0 (Server:1) Session1.pml:40 (state 2)[messageSU.session = sessionId]

8:proc  0 (Server:1) Session1.pml:41 (state 3)[messageSU.content1 = server]

spin: _spin_nvr.tmp:5, Error: assertion violated

spin: text of failed assertion:

    assert(!(((!!((Server._p==HomePage)))&&!((Server._p==AccountPage))))))

Never claim moves to line 5

[assert(!(((!!((Server._p==HomePage)))&&!((Server._p==AccountPage)))))]

spin: trail ends after 9 steps

```

FIGURE 3.14: Error-trail File of Property 3.13

Until this stage, we checked simple LTL formulas. We need to check that a certain action becomes true after a step and until another action occurs. For example, after the login the status global variable is set to 'login' and remains true until the logout page. In this scenario we can use a more complex LTL formula shown in the LTL formula 3.7, the symbol *p* is defined as `Server@HomePage`, while the *z* symbol is defined as `Server@LoginPage` and the symbol *m* is a new definition with the condition `# define Status ==login`. We verify that the condition is true after the login page until the logout page. The results in Figure 3.15 shows no errors and the behaviour of the model is correct. The number of transitions is still small at this stage of the model.

$$[] (p \&\& !z \rightarrow (!z U (m \&\& !z))) \quad (3.7)$$

```
State-vector 48 byte, depth reached 93, errors: 0
    71 states, stored (89 visited)
    25 states, matched
    114 transitions (= visited+matched)
    0 atomic steps

128.730          total actual memory usage
pan: elapsed time 0 seconds
```

FIGURE 3.15: Verification Result of Property 3.7

Again for the same LTL formula 3.7 we can modify the condition to `# define Status ==logout`, meaning that during the session the status of the client was inactive. The results in Figure 3.16 shows an error is located and now the elapsed time has increased

to (0.03) considering the size of the model. This is due to the complexity of proving this property.

```

State-vector 48 byte, depth reached 74, errors: 1

    34 states, stored
    0 states, matched
    34 transitions (= stored+matched)
128.730          total actual memory usage

pan: elapsed time 0.03 seconds

```

FIGURE 3.16: Verification Error Result of Property 3.7

We also use the same value of the global variable `# define Status ==logout` and check that after the logout page the session expires, as shown if LTL response formula 3.8, the `z` symbol is defined as `Server@LogoutPage`. The result shows no errors were found, as in 3.17

$$[] (z - > < > m) \quad (3.8)$$

```

State-vector 48 byte, depth reached 93, errors: 0

    56 states, stored (59 visited)
    8 states, matched
    67 transitions (= visited+matched)
128.730          total actual memory usage

pan: elapsed time 0 second

```

FIGURE 3.17: Verification Result of Property 3.8

It is also possible to check the status of the session on each page such as in 3.9. Where `q` is the account page and `m` is defined as `# define Status ==login`. Furthermore, changing the page locations using the current formula with different conditions makes it possible to understand the behaviour of the model. The results in Figure 3.18 show no errors were found, and since the account page is at the start it was possible with a small number of states and transitions.

$$<> (q \&\& m) \quad (3.9)$$

```
State-vector 48 byte, depth reached 30, errors: 0

    24 states, stored (48 visited)

    27 states, matched

    75 transitions

128.730          total actual memory usage

pan: elapsed time 0 second
```

FIGURE 3.18: Verification Result of Property 3.9

LTL formula	Depth reached	States Stored	Transitions	Total Memory (Mb)	Time (sec)
3.1	26	33	65	128.730	0
3.2	26	17	47	128.730	0
3.3	99	59	65	128.730	0
3.4	112	68	77	128.730	0
3.5	93	71	114	128.730	0
3.6	8	7	18	128.730	0
3.7	93	71	114	128.730	0
3.8	93	56	67	128.730	0
3.9	30	24	75	128.730	0

TABLE 3.2: Verification Results of Properties for Model without Time.

In Table 3.2 we list all the verification results for the first model. We notice that by keeping the model simple we managed to verify main the properties efficiently without increasing the complexity level. The depth reached and the number of states relies on

the property under examination and it increases with the increase in the complexity of the LTL formula. More web applications' properties can be verified using the LTL patterns. The next section presents the model with dynamic navigation properties.

3.2.2 Modelling Dynamic Navigation

In the this model of the web application we consider the dynamic behaviour of the transitions. We do not assume the generation of new content but the modification of the navigation path based on the client input and server status as discussed in the background, Section 2.2. The main challenge in modelling with SPIN is to understand the execution behaviour of the PROMELA code.

In order to model a correct behaviour that represents the client and server exchanging messages and changing transitions simultaneously we developed a new approach. The new approach takes into account the execution behaviour of SPIN and the dynamic transitions during the session. For example, in the current PROMELA model if the server process is at the certain location and the client requests to be redirected to a previous page; SPIN will reach a deadlock situation since the server process will be expecting a different message than the client process sent.

The method to solve the problem is to add all the navigation possibilities of the model under each page (i.e. location label in PROMELA). In detail, if the client is on the account page, there will be two options in our model, to logout of the session or to move to the next page which is the payment page in this model. The next example will explain further. In the current model the run is infinite (in order to analyse dynamic navigation properties where a single input could lead to different pages) in contrast to the first model in 3.2.1 where both processes terminate at the end of the session.

At the server side in listing 3.10, we have two options in the `if-fi` selection loop, if the message contains the request payment then the client will be directed to the next page (line 4). If the content of the request contains logout (line 9), the server will redirect the client to the home page. The nature of the execution is non-determinism in SPIN.

```

1 AccountPage:
2 if
3   :: network? msg2 (partnerB, data)->                                /*
      Move to Payment Page*/
4     (data.session == sessionId) && (data.content1 ==
      payment) ->
5       page = account
6       network ! ack (partnerB, messageSU)
7       goto PaymentPage
8   :: network ? msg5 (partnerB, data) ->                                /*
      Client request logout*/
9     (data.session == sessionId) && (data.content1 == logout) ->
10      page = home; network ! ack (partnerB, messageSU)
11      statusA = err
12      Status = logout
13      goto HomePage
14 fi

```

LISTING 3.10: Server Dynamic Navigation

In the client side in listing 3.11, we use the repetition (`do - od`) loop on each page to represent a continuous change of request. The client sends the session id and page request, and SPIN redirects the client process to a different page. We used a different labelling system at the client side to represent the pages. For example in line 1 C

represents the payment page. This means at the current stage the client is on the payment page. Inside the (do - od) loop there are three options. First the client may request to confirm the payment or go back to the account page or logout of the session. In any scenario, the server process will also have all the options under each page, resulting in a deadlock free model.

```

1 C:      /*  Payment Page */
2 do
3         ::messageUS.session = sessionId; messageUS.content1 =
         confirm;
4         network! msg3 (partnerA, messageUS);
5         network? ack (partnerA, data)->goto D; /* D =
         Confirmation Page */
7         ::messageUS.session = sessionId; messageUS.content1 =
         account;
8         network! msg2 (partnerA, messageUS);
9         network? ack (partnerA, data)->goto B; /* B =
         Account Page */
11        ::messageUS.session = sessionId; messageUS.content1 =
         logout;
12        network! msg5 (partnerA, messageUS);
13        network? ack (partnerA, data)->statusB = err; goto A;
         /* A = Home Page */
14 od;

```

LISTING 3.11: Client Dynamic Navigation

3.2.2.1 Simulation and Verification Results of the Dynamic Navigation Model

In order to understand the behaviour of the model at the current stage we run the following tests. First we check that the model is deadlock free and there are no assertion violations. SPIN shows in Figure 3.19 the result of the verification. Also, it shows that both processes never reach an end state, meaning that the model had infinite executions of a client logging in and logging out of any page. As a result, more properties can be verified than in the first model in 3.2.1. The results show that the model is still small with only 92 states and 110 transitions.

```

State-vector 40 byte, depth reached 52, errors: 0
    92 states, stored
    18 states, matched
    110 transitions (= stored+matched)
128.730          total actual memory usage
unreached in proctype Server
    DynamicModel.pml:124, state 66, "-end-"
    (1 of 66 states)
unreached in proctype User
    DynamicModel.pml:181, state 59, "-end-"
    (1 of 59 states)
pan: elapsed time 0 seconds

```

FIGURE 3.19: Safety Verification Result of Dynamic Navigation Model

The sequence of transitions is different, and the client can logout of any page in the current model stage of the web application. We use the following command line (`spin -r -s -u50 DynamicModel.pml`) with the run options (-s) to show the send events on the channel, (-r) to show the receive events on the channel and (-u50) to execute

the first 50 steps of the model. Figure 3.20 shows the simulation run and it shows that the client starts with logging in `msg1` and then logs out (line 16), then the second time the user processes login and proceeds to other pages. Each line of the simulation contains the step number, process number, file name, and channel action and content.

```

6:      proc  1 (User:1) DynamicModel.pml:134 Sent msg1,server,0,user
6:      proc  0 (Server:1) DynamicModel.pml:44 Recv msg1,server,0,user
8:      proc  0 (Server:1) DynamicModel.pml:46 Sent ack,server,0,server
8:      proc  1 (User:1) DynamicModel.pml:135 Recv ack,server,0,server
17:     proc  1 (User:1) DynamicModel.pml:145 Sent msg5,server,0,logout
17:     proc  0 (Server:1) DynamicModel.pml:61 Recv msg5,server,0,logout
20:     proc  0 (Server:1) DynamicModel.pml:64 Sent ack,server,0,server
20:     proc  1 (User:1) DynamicModel.pml:146 Recv ack,server,0,server
29:     proc  1 (User:1) DynamicModel.pml:134 Sent msg1,server,0,user
29:     proc  0 (Server:1) DynamicModel.pml:44 Recv msg1,server,0,user
31:     proc  0 (Server:1) DynamicModel.pml:46 Sent ack,server,0,server
31:     proc  1 (User:1) DynamicModel.pml:135 Recv ack,server,0,server
40:     proc  1 (User:1) DynamicModel.pml:142 Sent msg2,server,0,account
40:     proc  0 (Server:1) DynamicModel.pml:55 Recv msg2,server,0,account

```

FIGURE 3.20: Message Exchange Sample of Dynamic Navigation Model

The assumptions now include that the client can logout and login again. As a result, we need to check that on any page after a successful authentication that session is active. We can use the LTL formula 3.10 to verify that client cannot be on the account page when the status of the session is logout. The `q` symbol is defined as `(Server@AccountPage)`, while `m` is defined as the condition `(session == logout)`. The result of the verification shows an error is found and that proves that when the client is at the account page the session is set to login.

$$\langle \rangle ((q) \&\& (m)) \quad (3.10)$$

```

State-vector 48 byte, depth reached 26, errors: 1
    19 states, stored (37 visited)
    18 states, matched
    55 transitions (= visited+matched)
128.730          total actual memory usage
pan: elapsed time 0 seconds

```

FIGURE 3.21: Verification Result of Property 3.10

Another method of verifying session management properties is by assertions, as at each page in the server process SPIN can check the assertions on global variables. For example we use `(assert (Status ==login))` before each page followed by the login page to ensure that the client is always logged in at this stage before making any further transitions. Figure 3.22 shows that there are no assertion violations.

```

State-vector 48 byte, depth reached 93, errors: 0
    56 states, stored (59 visited)
    8 states, matched
    67 transitions (= visited+matched)
128.730          total actual memory usage
pan: elapsed time 0 second

```

FIGURE 3.22: Verification Result of Global Variable Assertions

In order to verify the navigation path in the current stage, we can use either the LTL response or precedence patterns (2.5.3). For example, it is always the case that the account page comes before the payment page and for that we use the precedence formula

3.11. However, since we now have different possibilities of next pages from each page we use the response LTL formula 3.12 and we include all the pages that come after an initial state. For example, from the account page there are three options (payment page, logout page, or home page). In 3.12 the symbol q represents the account page and is defined as `Server@AccountPage`, r is defined as `Server@PaymentPage`, z is defined as `Server@LogoutPage` and, p is defined as `Server@HomePage`. So from the account page there will be three options, and we use the logical operator (or) represented by the symbol \parallel between pages. The result of this property shows no error, as in Figure 3.23, and there is an increase in the number of states and transitions.

$$!rUq \quad (3.11)$$

$$\Box(q \rightarrow \langle \rangle r \parallel \langle \rangle z \parallel \langle \rangle p) \quad (3.12)$$

```
State-vector 48 byte, depth reached 135, errors: 0
    141 states, stored (169 visited)
    69 states, matched
    238 transitions (= visited+matched)
128.730          total actual memory usage
pan: elapsed time 0 seconds
```

FIGURE 3.23: Verification Result of Property 3.12

3.2.3 Modelling with Time Constraints

Modelling using discrete time allows us to represent more session and authentication properties and to simulate time-out scenarios in a web application. In order to model

time in SPIN, we add a discrete time process that controls the session.

The clock process contains a loop that controls the `timer` variable. The timer expires when its value reaches some value controlled by the pre-defined variable `MAX`. The clock process contains two statements; the first statement checks the condition of the timer is less than the value `MAX` and then increments it. The statement starts with `atomic` to preserve the exclusive privilege to execute before other processes in the model, and this it ends this with `skip` so another process can execute its statements. This method prevents the timer from controlling the session and from reaching a "infinite loop" status. The second statement contains a condition that checks if the timer is equal to to the value `MAX` and then it assign it to zero and skip. The server controls the timer and resets the value after each receiving of a message during a valid session. We model the timer to verify the security properties, So if the value reaches `MAX` the server will terminate the session and redirect the client to the home page.

```

1 MAX = n
2 proctype Clock()
3
4 do
5     ::atomic {timer < MAX -> timer ++} skip;
6     ::else -> atomic {timer == MAX -> timer = 0} skip;
7 od;
```

LISTING 3.12: Clock Process

At the server side, it is the same as in the previous models. We added a global variable (`Session`) to use it with the timer in the verification phase. Also, as in the Clock process, each atomic block ends with a skip, in order for other processes to execute the next statement. The first assertion (`assert (timer <= MAX && Session == valid && Status == login)`) checks that at this stage of the session the timer

is less than the timeout value, the session is valid and the status is login. When the server receives the second message it will start another selection statement (`if - fi`) that is similar to the first statement that checks the session id, and only the content of the message. In this case the client requests to move to the Account page. An assertion is made after receiving the second message that the page sequence is correct and the client is logged in. If there is a time-out the server will send an error message. Figure 3.24 shows that after (msg3) there was a time-out and the server sends an error message.

```

2 HomePage:
3 page = home /* LOGIN */
4 messageSU.session = sessionId
5 messageSU.content1 = server
6 if
7   ::network ? msg1 (partnerB, data)-> atomic {
8     (data.session == sessionId) && (data.content1 == user)
9     ->
10     network ! ack (partnerB, messageSU)
11     statusA = ok
12     Session = valid
13     Status = login }
14   skip;
15 fi
16 AccountPage: assert (timer <= MAX && Session == valid && Status
17   == login)
18 if
19   ::network? msg2 (partnerB, data)-> if

```

```

18                                     /* Move to Payment Page*/
19  ::atomic {(data.session == sessionId) &&
20    (data.content1 == account)&&(timer < MAX) ->
21    page = account ;timer = 0
22    network ! ack (partnerB, messageSU)
23    goto PaymentPage;} skip;

25  ::else->
26  atomic {Session = invalid;
27  network!error(partnerB, messageSU);
28  Status = logout;flag ++;MSG;goto HomePage;}
29  skip;                                     fi;
30 fi

```

LISTING 3.13: Server Process with Timer

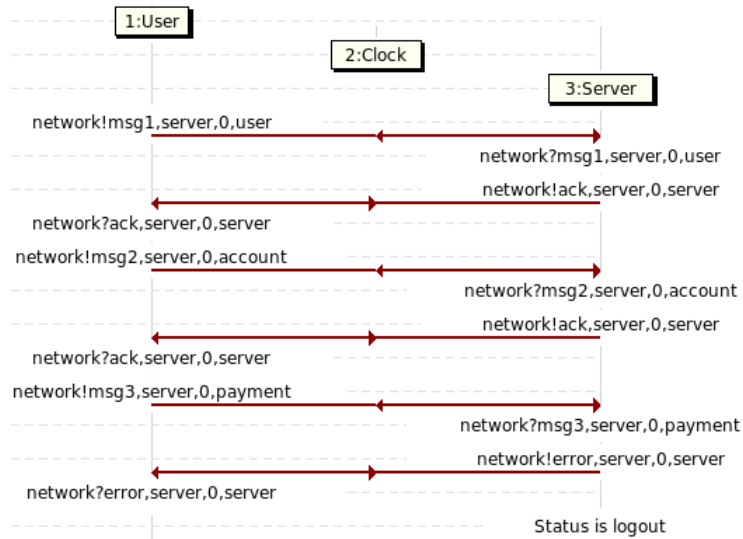


FIGURE 3.24: Simulation Chart of the Discrete Time Model.

We also add a global variable (flag) to each page and whenever the server reaches a time-out and redirects the client to the home page (flag) is incremented. We also

add (flag2, flag3, and flag4) at each page construct to assist in understanding and debugging our code. This method ensures that the timer is active in all the sessions and in different executions. The global macro MSG is defined as `printf("Status is %e n", Status)` to print the status of the user after each logout from the session.

The value of MAX has an effect on the behaviour and number of states in the model so we set it to the value 2 to keep the number of states reachable. The run of the model is infinite in order to achieve a realistic scenario of a client being logged off after a period of inactivity. Figure 3.25 shows the result of the simulation after 1,000 steps. Through the execution of the model there was a time-out on each page.

```
depth-limit (-u10000 steps) reached
#processes: 4

    partnerA = server
    partnerB = user
    statusA = ok
    statusB = ok
    page = confirm
    Status = login
    Session = valid
    timer = 0
    MAX = 2
    flag = 12
    flag2 = 4
    flag3 = 5
    flag4 = 2
```

FIGURE 3.25: Simulation Results of the Discrete Time Model

3.2.3.1 Simulation and Verification Results of Timed Model

We now use the global variable `MAX` in the LTL formulas, for example we verify that the client cannot be logged in and the timer is more than `MAX`. We use the 'Absence' pattern of LTL formula 3.13 where `q` is the account page and `m` is defined as (`timer > MAX`). The result shows no error as shown in Figure 3.26. It is clear that the number of states has increased compared to in the previous models.

$$\Box(q \rightarrow \Box(!m)) \quad (3.13)$$

```
State-vector 64 byte, depth reached 250, errors: 0
    1182 states, stored
    736 states, matched
    1918 transitions (= stored+matched)
128.730          total actual memory usage
pan: elapsed time 0 seconds
```

FIGURE 3.26: Verification Result of Property 3.13

3.2.4 Adding an Intruder to the Model

At this stage we add an intruder process in our model. The model process is based on the intruder model designed by [Dolev and Yao \[1983\]](#). The intruder collects knowledge about both parties during the session. The knowledge is used during a session to create and send his own messages or to tamper with other agent communication. The assumption is that the intruder has the session identity and the client identity. The intruder first overhears and intercepts all the messages over the network, then chooses a recipient and then generates new messages using information from overheard

messages and some prior information, and finally sends a new or captured message to another entity in the system.

The purpose of introducing an intruder to the model is to examine the differences between the sequence of actions and timing. Though we assume that an online banking system is using the HTTPS communication protocol, the intruder can still be in the middle of the communication parties. The assumption for this model is that the intruder has already started to collect information about the parties in the communication and then he becomes active in the middle. We do not include any discussions about cryptography in the model as it is beyond the scope of this research. The main aim is to analyse the order of events in the communication and timing.

We use the same timer process from the previous model [3.2.3](#) and add the intruder process. The first part of the intruder process code is that the intruder receives a message from the client and stores it in the in local variable `intercepted`. The whole process after that contains one repetition construct that involves storing messages, choosing a message, choosing a recipient, assembling the message content and finally sending the message.

```

1 active proctype Intruder() {
2     mtype msg, recpt;
3     Crypt data, intercepted;
4     do
5         :: network ? (msg, _, data) ->
6             if /* store the message */
7                 :: intercepted.session    = data.session;
8                 intercepted.content1 = data.content1;
9                 :: skip;
10    fi;

```

```
11 }
```

LISTING 3.14: Intruder Process

Secondly, the intruder chooses a message that is being exchanged between the client and the server, and then at this stage the intruder chooses a recipient and assembles the message.

```
2      if /* choose message type */
3
4          :: msg = msg1;
5
6          :: msg = msg2;
7
8          :: msg = msg3;
9
10         :: msg = msg4;
11
12         :: msg = msg5;
13
14     fi;
15
16     if /* choose a recipient */
17
18         :: recpt = user;
19
20         :: recpt = server;
21
22     fi;
23
24     if /* assemble the message */
25
26         :: data.session      = intercepted.session;
27
28         data.content1      = intercepted.content1;
29
30     fi;
```

LISTING 3.15: Selection Loop at Intruder Process

The final stage is to assemble the content of the message and then sends the message
Listing 3.16.

```
1      :: if /* assemble content1 */
```

```

2          :: data.content1 = server;
3          :: data.content1 = user;
4          :: data.content1= login;
5          :: data.content1 = home;
6          :: data.content1 = account;
7          :: data.content1 = payment;
8          :: data.content1 = confirm;
9          :: data.content1 = logout;
10         ::skip;
11     fi;
12 fi;
13 network ! msg (recpt, data);
14 od

```

LISTING 3.16: Method to assemble messages in Intruder Process

3.2.4.1 Simulation and Verification Results of Model with Intruder

The first result to be noticed is that the number of states increased dramatically during the execution of the compromised model, in contrast to the secure model. Table 3.3 shows the differences between the two models. The verification run time increases to 14 seconds in the compromised model and the depth reaches the maximum limit of 9999. By default the search depth is restricted to 10,000 steps. From the results it is clear that adding an intruder to the model can increase the number of transitions and states. The explanation for the increase is that at each step the intruder receives and sends the messages between the client and the server. Also, each storing and evaluating of the content of the messages increases the number of states.

Model	Depth reached	States Stored	Transitions	Total Memory (Mb)	Time (sec)
Secure	90	2436	3838	128.827	0
Intruder	9999	14666773	33523244	1023.944	14

TABLE 3.3: Verification Results of the Secure and the Compromised Model.

The next stage is to analyse the order of events and timing. We add the macro definition `# define MSG printf("Time is %d n ", timer)` to print the value of the time during the run of the model. After each message is received from the server side, SPIN prints out the value of the time, and this simulates a time stamp for the message exchange process.

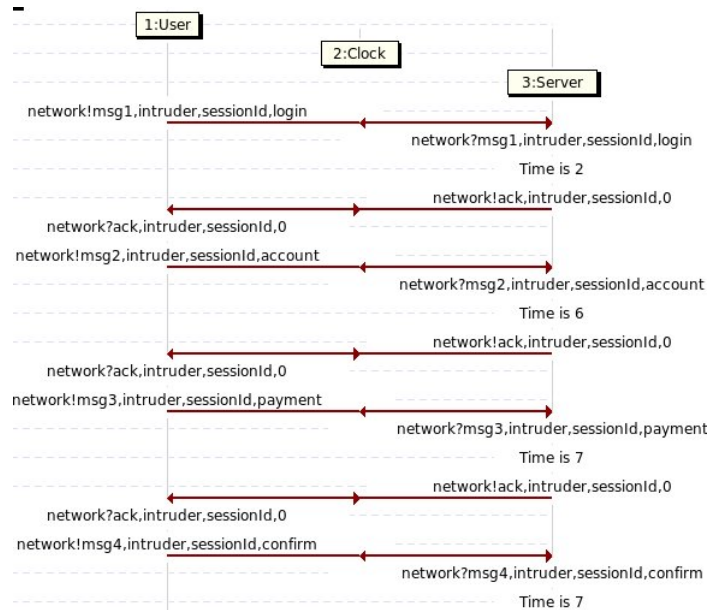


FIGURE 3.27: Secure Model.

Figures 3.28 and 3.27 show the differences between the two models. In Figure 3.27 we find that the client and server end the first exchange of messages at the time of value 2. While in the compromised model there was a delay of 1 time unit shown in Figure 3.28. The sequence of actions and timing were different in the compromised model.

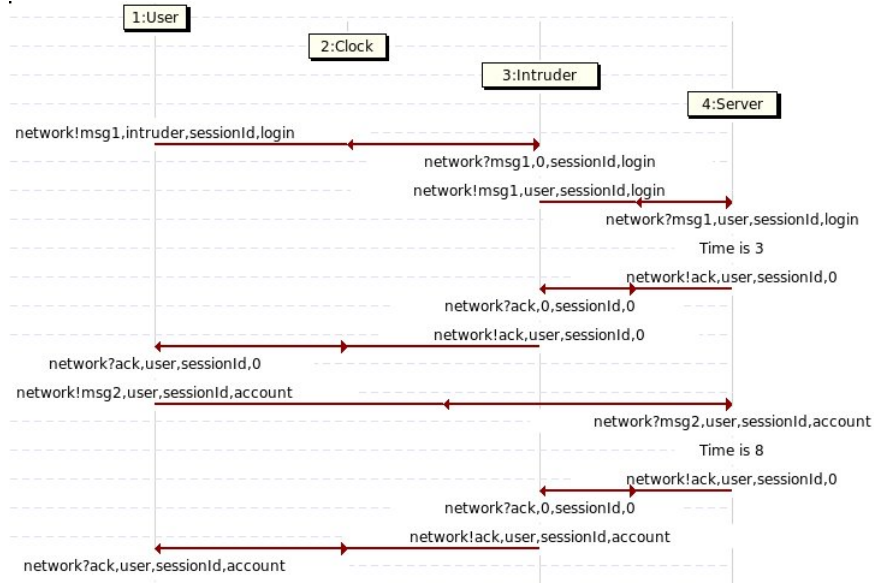


FIGURE 3.28: Model with Intruder.

3.3 Summary

This chapter presented the first model checking tool used in our research SPIN. We then modelled a web application in four different scenarios based on the requirements of verification. The models in this chapter were reviewed by *Professor Gerard Holzmann* who complemented the work on its soundness. Concluding remarks can be summarised as follows:

- In the model without timer [3.2.1](#), we showed how to verify basic web application properties using LTL. The model without timer was presented as a basic for the following models. We developed the model to represent the basic specifications of an online banking system. We used the simulation and verification to ensure the correctness of the behaviour.

- The model with dynamic navigation [3.2.2](#) presented a novel approach for modelling the dynamic exchange of messages based on different input from the user of the server status.
- In the timed model [3.2.3](#), we developed a discrete timer that suits the requirements of verifying web applications' properties without reaching the state explosion problem.
- The model with the intruder [3.2.4](#) proved that with the presence of an intruder in the middle of the session there will be a difference in the sequence of timing and actions.

In the next chapter we present the second model checking tool UPPAAL, and will show how to model web applications using a real time model checker.

Chapter 4

Modelling in UPPAAL

This chapter describes the second model checking tool UPPAAL used in our research. Firstly, background information about the UPPAAL model checker is given. This will assist later in describing how it can be used in the verification and analysis of web applications. Then the model of a web application is designed using UPPAAL. Finally, the results of the verification will be presented.

4.1 The Uppaal Model Checker

UPPAAL is a model checking tool suite for the verification of real-time systems. The UPPAAL modelling language extends the basic timed automata as defined in Chapter 2 with bounded integer variables and binary blocking synchronisation. Systems are modelled as a set of communicating timed automata. UPPAAL consists of a graphical user interface that allows systems descriptions to be defined graphically and a model-checker that combines on-the-fly verification with a symbolic technique reducing the verification problem to that of solving simple constraint systems. Furthermore,

UPPAAL also supports bounded-range integer and boolean data variables, which can be used in the guards, assignment and location invariants [Behrmann et al., 2004, Bengtsson and Yi, 2004].

UPPAAL is considered in the area of model checking tools as a fast and usable tool. The usability is due to the possibility of specifying the automata graphically and the existence of a graphical simulator on which some runs could be simulated. The efficiency is due to the fact that it restricts the type of properties checked to a reachability test. Hence, the verification engine can be better optimised for the task of reachability.

4.1.1 The Modelling Language

In this section we present a brief description of the UPPAAL modelling language from the main tool reference in Behrmann et al. [2004]. The expressions in UPPAAL range over clocks and integer variables and are used with the following labels:

- **Select:** To represent variables that are accessible on a specific edge and they will take a non-deterministic value in the range of their respective types.
- **Guard:** A guard that evaluates a boolean on clocks, integer variables, and constants.
- **Synchronisation:** A synchronisation label that represents the channels between processes. It uses the expression (!) to represent the action of sending or the expression (?) to represent the action of receiving.
- **Update:** An update label is used to update clocks integer variables, and constants and, can only be used to assign integer values to clocks.

- **Invariant:** An expression on locations to satisfy conditions on clocks, integer variables, and constants.

The main purpose of a model-checker is to verify the model specifications and to determine whether a process reaches a deadlock. UPPAAL's response is either "The property is satisfied" or "The property is not satisfied". When the verifier cannot determine the truth value of the property it responses with "The property is maybe satisfied". UPPAAL uses a simplified version of CTL. Figure 4.1 the relationship between CTL syntax and UPPAAL's query language syntax.

<i>CTL Formula</i>	<i>UPPAAL Form</i>
$\mathcal{A} \Box \varphi$	$\mathbf{A} \Box \varphi$
$\mathcal{A} \Diamond \varphi$	$\mathbf{A} <> \varphi$
$\mathcal{E} \Box \varphi$	$\mathbf{E} \Box \varphi$
$\mathcal{E} \Diamond \varphi$	$\mathbf{E} <> \varphi$
$\varphi \rightsquigarrow \psi$	$\varphi \text{ --> } \psi$
$\neg \varphi$	$\mathbf{not} \ \varphi$
$\varphi \wedge \psi$	$\varphi \ \mathbf{and} \ \psi$
$\varphi \vee \psi$	$\varphi \ \mathbf{or} \ \psi$
$\varphi \Rightarrow \psi$	$\varphi \ \mathbf{imply} \ \psi$

TABLE 4.1: CTL Syntax in UPPAAL

The query language in UPPAAL consists of path formulas and state formulas. In summary, the formula queries available in UPPAAL are:

- $\mathbf{E} <> p$: meaning that there exists a path where p will eventually hold.
- $\mathbf{A} \Box p$: meaning that for all paths p will always hold.
- $\mathbf{E} \Box p$: means that there exists a path where p will always hold.
- $\mathbf{A} <> p$: it means that for all paths p will eventually hold.
- $p \text{ --> } q$: meaning that whenever p holds q will eventually hold.

Path formulas can be classified into reachability, safety and liveness. Figure 4.1 illustrates the different path formulas supported by UPPAAL. Each type is described below.

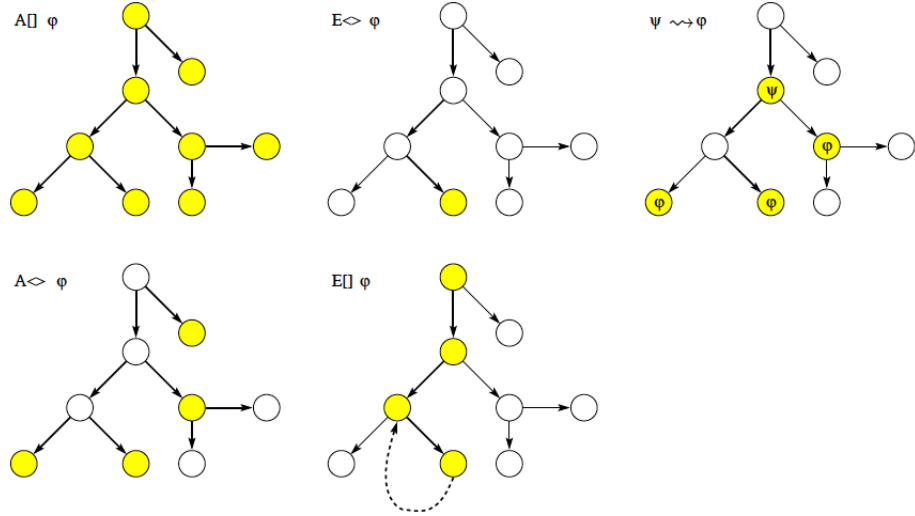


FIGURE 4.1: Path Formulas Supported in UPPAAL.

State Formula: Is defined as an expression that can be checked for a particular state without looking at the behaviour of the model of [Behrmann et al., 2004]. For example, $i == 5$, is true in a state whenever i equals 5. Moreover, we can verify that a process is in a given location using an expression in the form $M.1$, where M is a process and 1 is a location.

To check *deadlocks* in UPPAAL, we use the state formula ($A[]$ not deadlock).

For verifying *Reachability properties* we check whether a given state formula, φ , can possibly be satisfied by any reachable state. We use the path formula ($E \Diamond \varphi$).

In UPPAAL we verify *Safety properties* formulated positively, e.g., something good is invariantly true. Let φ be a state formula. We express that φ should be true in all reachable states with the path formula ($A \Box \varphi$), whereas ($E \Box \varphi$) says that there should exist a path such that φ is always true.

Finally, for verifying *Liveness Properties* we use the path formula $(A \Diamond \varphi)$, which means that φ is eventually satisfied. The *leads to formula* can be used, and it is expressed as $(\varphi \rightarrow \psi)$ which is read whenever φ is satisfied, then eventually ψ will be satisfied.

4.1.2 Modelling Time in UPPAAL

This section will explain in detail the concept of time in UPPAAL. In addition, the main concepts of the modelling language will be presented. The time model in UPPAAL is represented as continuous time. It is implemented as regions and the states are thus symbolic, which means that a state does not have any concrete value for the time, but rather has differences [Alur and Dill, 1994]. In order to illustrate how time is modelled in UPPAAL, a simple model will be used. The model uses an observer to show the differences. An observer is an add-on automaton that takes control of detecting events without interfering with the observed system. Figure 4.2 shows the first model with the observer. The reset of the clock ($x := 0$) is assigned to the observer in order to make it work. Time is used through clocks, in this example, x is a clock declared as `clock x`; in the global declarations section. A channel is declared (`reset`) for synchronization with the observer.

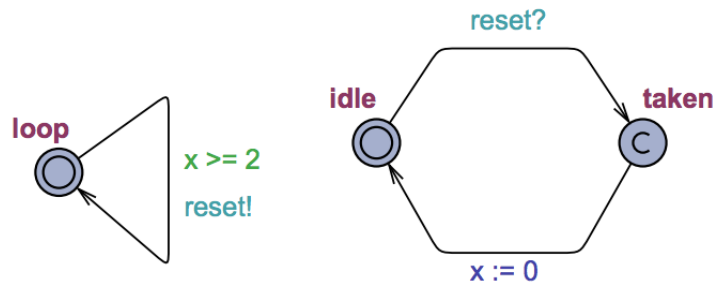


FIGURE 4.2: The Automata $P1$ with Obs Observer.

The channel synchronization is a hand-shaking between `reset!` and `reset?` in the example. The clock may be reset after 2 time units. The observer detects this and

performs the reset action. The state `taken` of the observer is of type `committed`, and this will be explained later in this section.

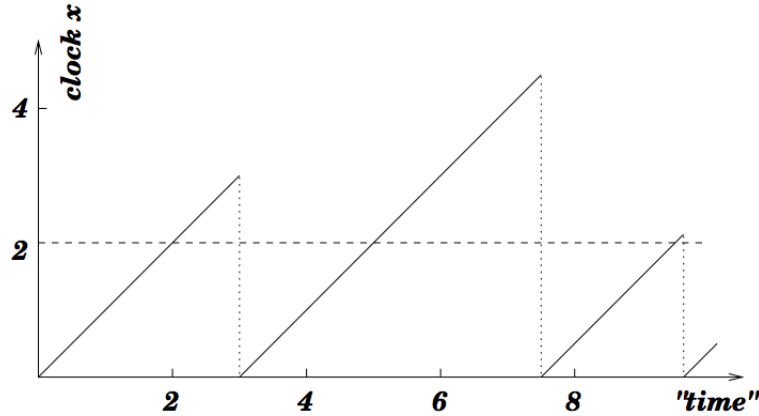


FIGURE 4.3: Possible Behaviour of the First Example.

Figure 4.3 shows the expected behaviour of the first run. As in the previous chapter, a verification is needed to understand the correctness of the model. At this stage, two properties are checked as follows:

- $A[] \text{ Obs.taken} \text{ imply } x \geq 2$ checks that for all states, being in the location *Obs.taken* implies that $x \geq 2$.
- $E<> \text{ Obs.idle and } x > 3$ this property checks is it possible to reach a state where *Obs* is in the location *idle* and $x > 3$.

When adding the properties to UPPAAL, the verifier checks whether it is correct or not, as shown in Figure 4.4. When the first property was checked, UPPAAL returned a result (Property is satisfied). When the value of the clock was changed to 3, UPPAAL returned a negative result (Property is not satisfied). The figure also shows the second property.

Another method of modelling and controlling time in UPPAAL, is by using an invariant as a *progress condition* [Behrmann et al., 2004]. This means that the system is not

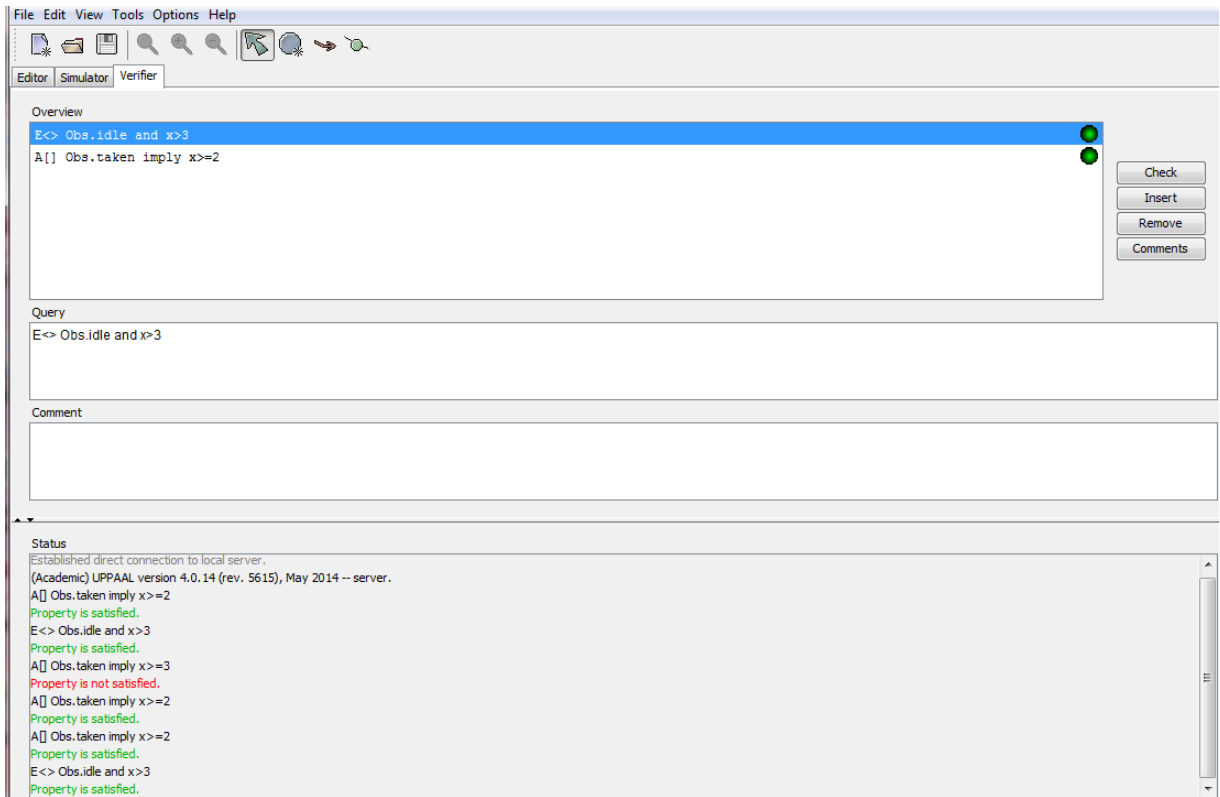


FIGURE 4.4: UPPAAL Verification Example.

allowed to stay in a specific state for more than a fixed number of time units. After that time, an action needs to be taken. As an example, when the invariant is added to the sample example as shown in Figure 4.5, the system is not allowed to stay in the `loop` location for more than 3 time units, and then the clock is reset.

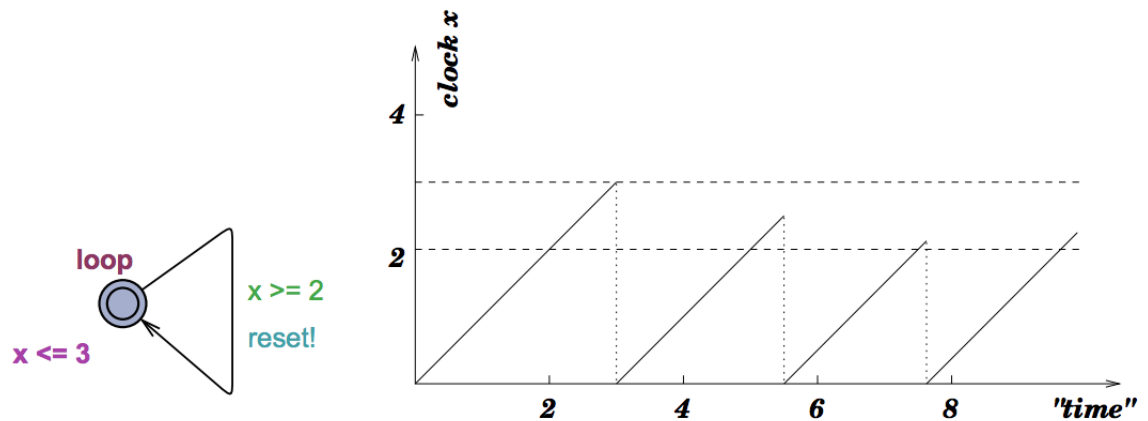


FIGURE 4.5: UPPAAL Behaviour with Invariant.

By adding an invariant to locations, more verification options are possible. For example, in contrast to the first example, now it is possible to check the following properties:

- $A[] \text{ Obs.taken} \text{ imply } (x \geq 2 \text{ and } x \leq 3)$: here the property checks that the transition is taken when the value of the clock `x` is in the interval $[2,3]$.
- $E<> \text{ Obs.idle and } x > 2$: meaning that it is possible to take an action when the value of the clock `x` is in the interval $[2,3]$.
- $A[] \text{ Obs.idle imply } x \leq 3$: This property ensures that the upper bound is respected.

Another method of controlling time in UPPAAL is through adding a guard to the channel. Figure 4.6 is the same example as before but the invariant is removed and the guard (`x >= 2 and x <= 3`) is added. By removing the progress condition and adding a guard, the behaviour of the system is different. At this stage, the system may reach a deadlock if the transition is not taken after 3 time units. The property $A[] x > 3 \text{ imply not Obs.taken}$ shows the deadlock scenario.

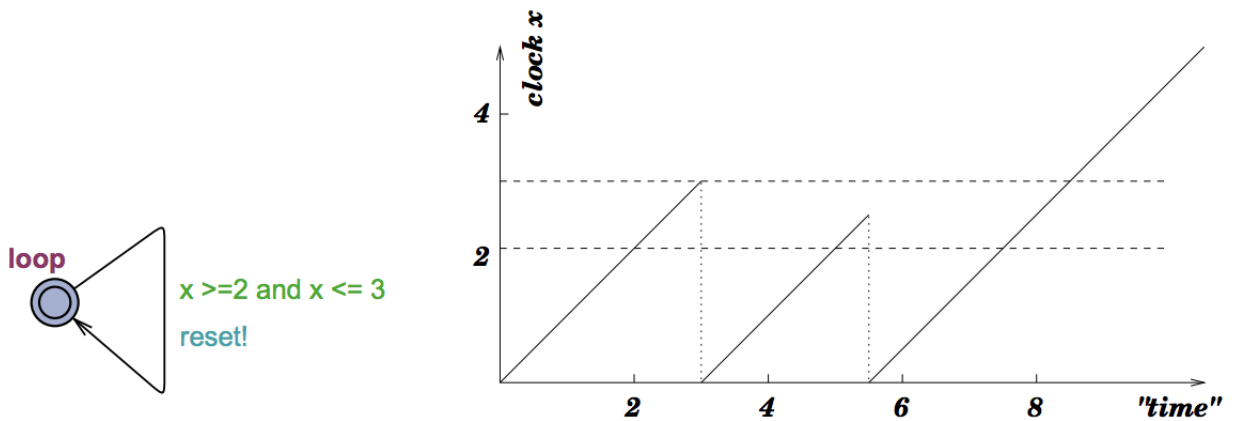


FIGURE 4.6: UPPAAL Behaviour with Guard.

Locations in UPPAAL : There are three different locations in UPPAAL, normal locations with or without invariants, urgent locations and committed locations. Each has a different effect on time. Figure 4.7 illustrates the three types as parallel automata with different clocks. P0 represents a normal location, P1 has the urgent location marked by *u*, while P2 is committed location marked by *c*.

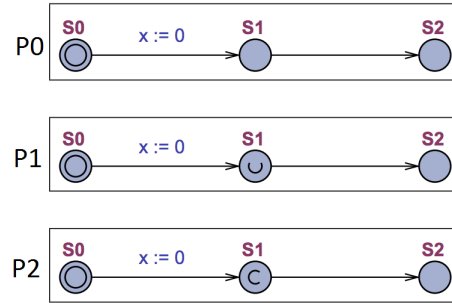


FIGURE 4.7: Location Types in UPPAAL.

In order to understand the different normal urgent locations, a property check will be illustrated as follows:

- $E \langle \rangle P0.S1 \text{ and } P0.x > 0$: In a normal location it is possible to wait in S1 of P0.
- $A [] P1.S1 \text{ imply } P1.x == 0$: In an urgent location it is not possible to wait in S1 of P1.

A committed location is more restrict as the delay at a location is not allowed and the committed location must be left in the successor state.

4.2 Modelling Web Applications in Uppaal

In this section we model a simple online banking application using UPPAAL. As in Chapter 3, we develop the model gradually in order to control the behaviour of the

model and to have clear results. The first model will be without clocks. The second model will contain dynamic navigation transactions. In the third model we add clocks to the model and verify time related properties. In the final model we add an intruder to the model and compare between a secure and compromised model. Modelling in UPPAAL is different than in SPIN. The graphical design method can be more complex to understand than using the PROMELA language in SPIN. We will compare the differences further in Chapter 5.

4.2.1 Model without Time Constraints

The first model in UPPAAL will only contain the basic properties as we did in the SPIN model in Chapter 3. We examine how to model properties without adding clocks at this stage. The main advantage of this method of modelling is to compare the differences later with the timed model. We consider the logical order of pages and actions without time constraints.

The global variables used in the first model are shown in listing 3.1. We have four main variables; page, status, Id and session. Since UPPAAL does not support the symbolic name `mtype` definition as in SPIN we develop another method. By declaring each page as constant variable and giving it a number, the variable page can take any value from the constant pages when it is as a state condition between locations later in modelling. We also created boolean variables to represent each page, which will assist later in the verification stage. The same method is also used also for the rest of the main variables. Status can have two values either login or error which represents the client logging out of his account. The session variable can be valid or invalid when the session expires.

We declare one channel (network) between both parties. We use the update on channels to deliver each page request from the client side. For example, after the authentication, the client updates the global variable page with the constant variable account (`page = account`). The advantage in this model is that the server first receives the message on the network and uses the condition (`page == account`) before moving to another page. Since we assume that the session will continue at this stage, the condition will be true. In order to keep the order of pages correct, after each page the client updates the global variable page to the current page request. The server then makes the same condition for each page. We use the conditions later at the verification stage.

The global variables session and status are used by the server to control the session. After a successful login the server updates the status to login and the session to valid. When the session expires then the variables are updated.

```
1 chan network;  
2 //Pages  
3 int Page;  
4 const int home = 0;  
5 const int account = 1;  
6 const int payment = 2;  
7 const int confirm = 3;  
8 const int logout = 4;  
9 bool A,B,C,D,E;  
  
11 //Status  
12 int Status;  
13 const int login = 0;  
14 const int error = 1;
```

```
16 //Id
17 int partnerA;
18 int partnerB;
19 const int client = 0;
20 const int server=1;

22 //Session
23 int Session;
24 const int valid = 0;
25 const int invalid = 1;
26 }
```

The second step is to model the client by creating a new template in UPPAAL. The client template has 5 locations that represent the pages in the web application model. We added intermediate committed locations between the main locations. The committed locations enforce that the synchronisation is atomic. We used the invariants in the intermediate committed locations as conditions. Figure 4.8 shows the locations and the transactions between them. From the home location to the first page we add an extra location for the following reasons:

- The client sets the **Page** variable to account, representing a message content. and uses the command **network!** to represent the send action.
- The **partnerA** variable is set to client as an id for the authentication properties.
- The middle location has an invariant that only the **partnerB** can receive the message. At this point the server will set the variable **partnerB** to server id.

- The next action is the receive action from the server side **network?**. The client checks that he is in communication with the server with the condition **PartnerB==server** and then proceeds to the account page.

The advantage of using an extra location between one location and another is that we can update the global variables and add conditions during the exchange of messages with the server side.

In the rest of the model, the client updates the **Page** variable with the page request along with the send action **network!**.

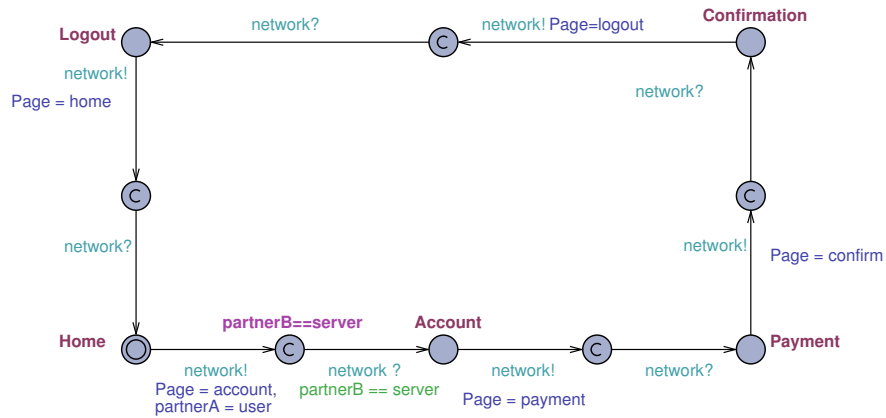


FIGURE 4.8: Client Automaton.

The server side has the same locations as the client template but with more conditions on edges and locations. Figure 4.9 shows the server automaton. The server has the opposite part of the communication. The communication starts with the sever receiving a message from the client. In the first part the server sets his id **partnerB** to server, and based on the page update from the client, the server updates the page boolean variables. A here represents the home page and it is set to false by the server. B represents the account page and it is set to true. The middle location has an invariant **partnerA==client** that states that only the partner with the id client can login. After that, the server replies and sets the **Status** variable to login and

the **Session** to valid. The rest of the server automaton has the same structure of updating the pages' boolean variables and in the logout location, the server changes the variables to reset the session.

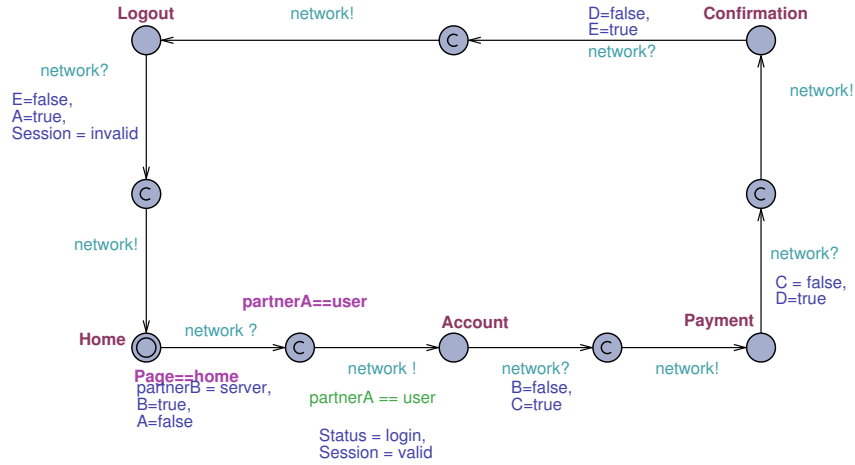


FIGURE 4.9: Server Automaton.

4.2.1.1 Simulation and Verification Results of Model without Time Constraints

In the next stage we run the simulation to gain insights into the behaviour of the model. Figure 4.10 shows the simulator screen displayer by UPPAAL. From the transitions chart we find that the middle location made it possible to represent the send and receive action between both parties before moving to the next page. The variables section shows the value of the variable at each stage of the communication. The verifier does not show the number of states or the verification time. It only shows whether the property is satisfied or not. We use UPPAAL's command line (`verifyta`) to show the number of states at each property.

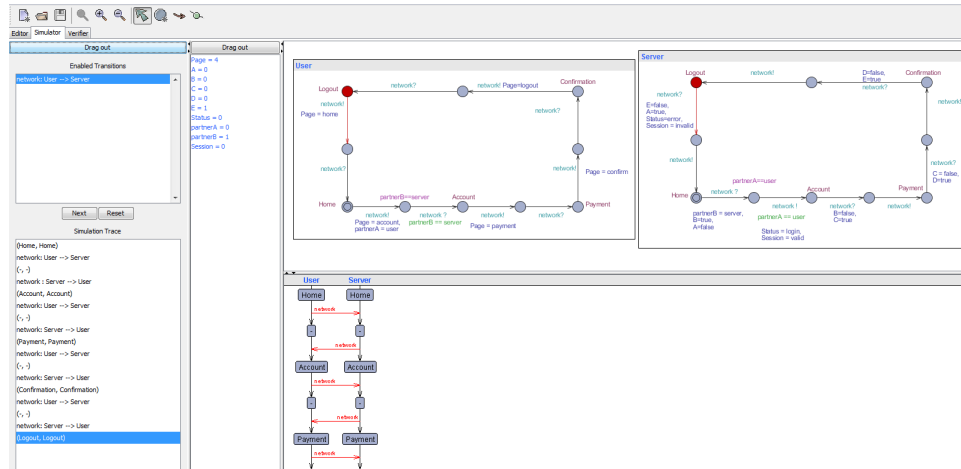


FIGURE 4.10: Simulation Result of Model without Time Constraints.

For the first model we verify the following properties:

The page reachable from the top page always has a next page in the transition. This can be checked by verifying that there is no deadlock in the model using the formula in 4.1.

$$A[] \text{ not deadlock} \quad (4.1)$$

We then use the command line (`verifyta -u -o1 -t0 -f tracefile First.xml First.q`) to run the verifier and show the number of states. Where `-u` shows a summary after verification, `-o1` sets the search to depth first, `-t0` generates diagnostic information and `-f` writes it to the trace file. The results in 4.11 show that property is satisfied, and the number of states is 12.

Verifying property 1 at line 2

-- Property is satisfied.

-- States stored : 12 states

-- States explored : 12 states

FIGURE 4.11: Verification Result of CTL Formula 4.1

We then check that every page is reachable from the initial state. This can be done by using the query in 4.2

$$E[] \text{ (} Server.Home \text{ imply } Server.Account \text{)} \quad (4.2)$$

In query 4.3 we check that the home page is reachable from all pages.

$$E[] \text{ (} Server.Account \text{ imply } Server.Home \text{)}. \quad (4.3)$$

Query 4.4 verifies that the client cannot bypass the home page without the authentication process.

$$A[] \text{ (} Server.Home \text{ imply not } Client.Account \text{)} \quad (4.4)$$

We use the global variables **Status** and **Session** to check that when the client is on the account page the status is login and the session is valid.

$$E <> \text{ (} Client.Account \text{ and } Status == login \text{ and } Session == valid \text{)} \quad (4.5)$$

4.2.2 Modelling Dynamic Navigation

In this model of the web application we consider the dynamic behaviour of the transitions. We add edges at each page location to model different possibilities. This step might be considered as a straightforward step in contrast to the SPIN model in

section 3.2.2. However, in the first model in section 4.2.1 we only used one channel for the whole session as each location only had only one option. In the case of multi options from each location, we need to declare more channels as UPPAAL has a non-deterministic choice of edges that can affect the sequence of communication between the server and the client model. If there is only one channel and multi options the model behaviour will not be correct, for example we may find that the server is on the account page while the client is on the payment page during the simulation run. Even with the use of conditions on edges we could not control the correct sequence of pages as UPPAAL do not run the communication simultaneously.

In the global variables we declare another three channels for the following connection:

- channel (`accountToLogout`) from the account page to logout page.
- channel (`paymentToLogout`) from the payment page to logout page.
- channel (`paymentToAccount`) from the payment page back to the account page.

Figure 4.12 shows the client automata with added locations.

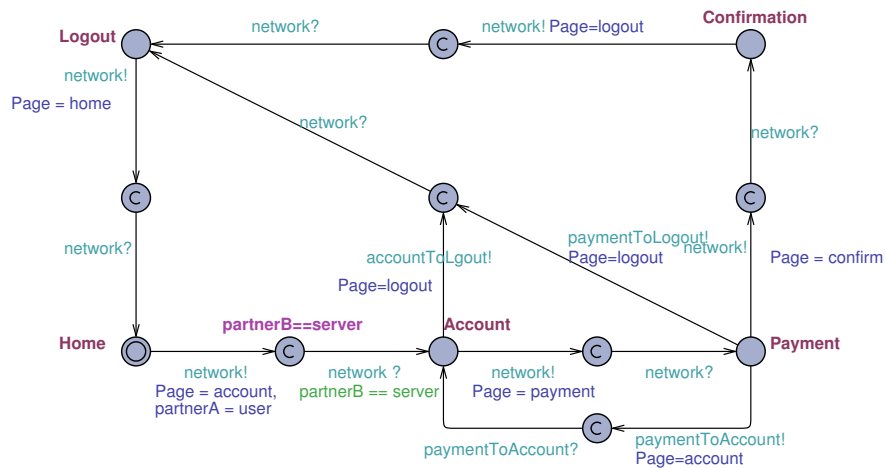


FIGURE 4.12: Dynamic Client Automaton.

In Figure 4.13 the server controls the session and updates the global variables after each change from the client side.

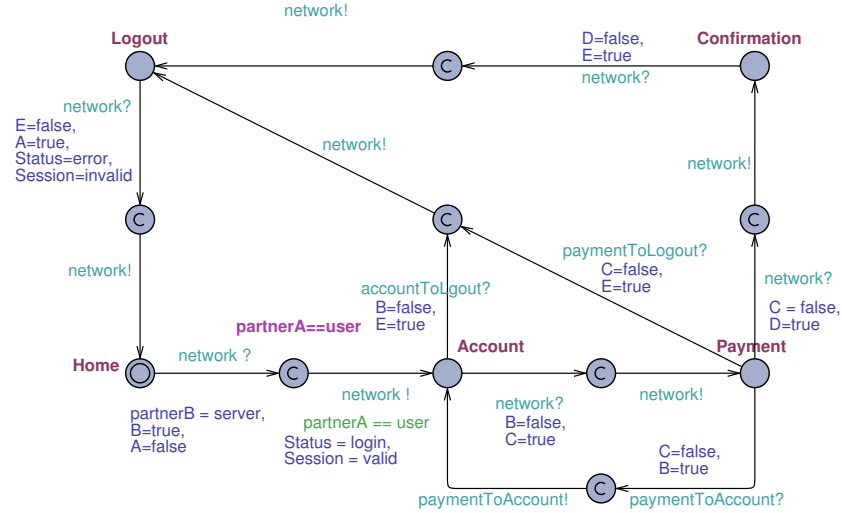


FIGURE 4.13: Dynamic Server Automaton.

4.2.2.1 Simulation and Verification Results of Dynamic Navigation Model

We use the simulation to analyse the behaviour in the first stages of the modelling. We first verify that there is no deadlock in the model using the query in 4.6. The results in 4.14 show that there is no deadlock and the number of states has increased compared to the first model in Section 4.2.1.

$$A[] \text{ not deadlock} \quad (4.6)$$

```
Verifying property 1 at line 11
-- Property is satisfied.
-- States stored: 14 states
-- States explored: 14 states
```

FIGURE 4.14: Verification Results of CTL Formula 4.6

After adding the new channels the sequence of actions were correct. To verify our approach we use the following query in 4.7 to verify that the client cannot bypass the home page without the authentication process.

$$A[] \text{ (Server.Home imply not Client.Account)} \quad (4.7)$$

4.2.3 Modelling with Time Constraints

Including time in the model assists in representing realistic properties such as the session management time-out mechanism. Furthermore, by adding time we can analyse the sequence of events as done in Chapter 3. In UPPAAL it is possible to use real time clocks. However, the time is symbolic and represented as clock constraints, and we cannot capture the value of time at each step. We only can compare the value to integer constraints.

To add time constraints to the model, we declare a global clock that is used by the server side as shown in listing 3.2. A constant integer MAX is declared with the value 10. We use it at the edge conditions when there is a time-out.

```

27 clock time;
28 const int MAX = 10;
```

Figure 4.15 shows the client side. We added three edges from the main pages to represent a time-out message by the server side. The server will use the network and send an error message. To avoid complicating the model, we did not add the dynamic navigation properties to the current model.

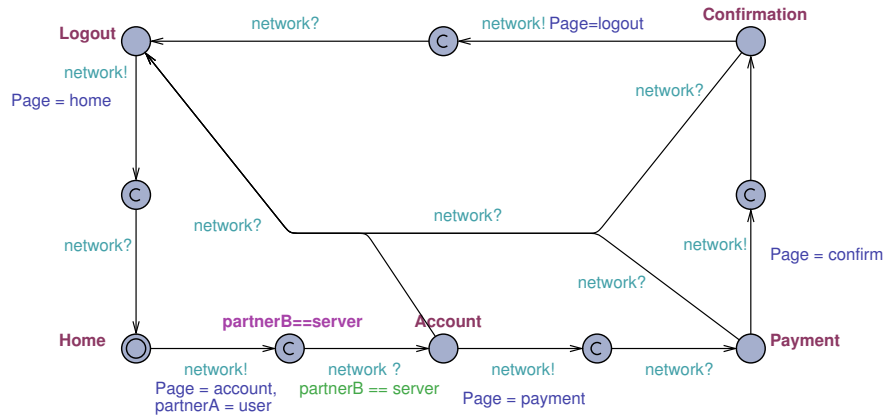


FIGURE 4.15: Timed Client Automaton.

In Figure 4.16 the server uses the clock to control the session. At the start the time value is 0. When the client is on the account page there will be two conditions: $\text{time} < \text{MAX}$, then the client could proceed to the next page or; $\text{time} \geq \text{MAX}$ here the server will send a message that the session has expired. And then the client will be redirected to the logout page.

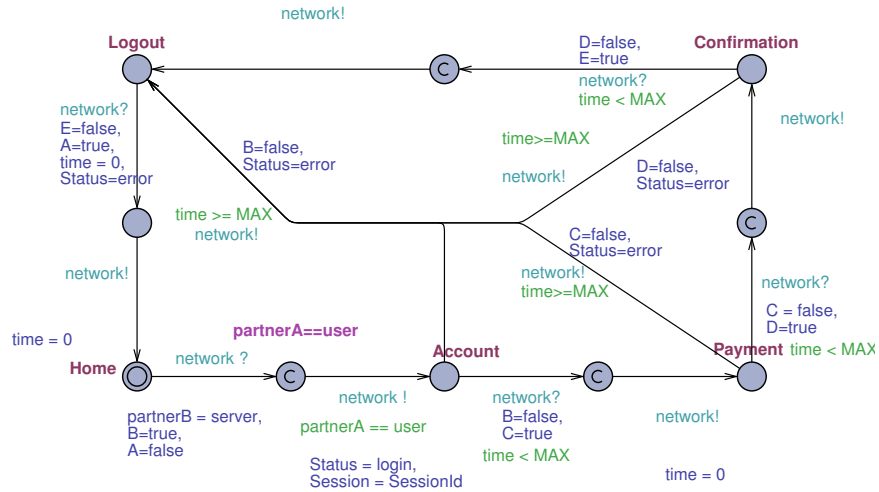


FIGURE 4.16: Timed Server Automaton.

4.2.3.1 Simulation and Verification Results of Timed Model

After adding the time, we can now check the value of the clock on each page with the following queries.

In query 4.8 we check that at a particular location a time-out can occur. We use the same query for all the pages in our model.

$$E <> Client1.Account \ \&\& \ Server1.Account \&\& \ time > MAX \quad (4.8)$$

In query 4.9 we verify that a time-out will not occur when the client is active.

$$E[] \ not \ (Client.Account \ and \ time > MAX) \quad (4.9)$$

We can also use the global variables to verify that while the login status holds, the time will not be more than **MAX** as shown in 4.10.

$$E[] \ not \ (Status == login \ and \ time > MAX) \quad (4.10)$$

4.2.4 Adding an Intruder to the Model

At this stage we add an intruder process to our model. The model process is based on the intruder model designed by [Dolev and Yao \[1983\]](#). The intruder collects information about both parties during the session. The knowledge is used during a session to create and send messages or to tamper with other agent communication. The intruder at first overhears and intercepts all the messages over the network. We use the location invariant to represent that the attacker has the knowledge of the identity of the session and participates in the communication. After receiving all the information, the intruder becomes active and sends a new message. Figure 4.17 shows the intruder automaton at the first location until the session id is validated. Then the intruder uses the local variable **data** to store the value of the session.

The next step is to get the communication parties' identities and to store them in the local variables `id1` and `id2`. If the `id2` is identified as the client, the intruder then starts by receiving a message from either of the parties and then replies.

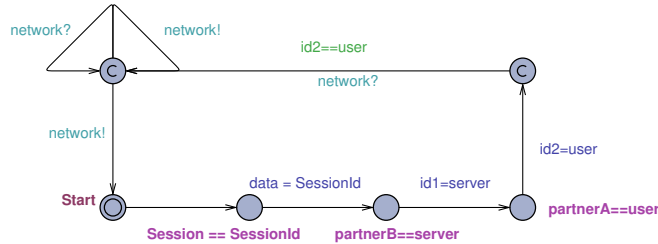


FIGURE 4.17: Intruder Automaton.

We used the same client and server automaton in the timed model in section 4.2.3. The next stage is to verify the sequence of actions and timing.

4.2.4.1 Simulation and Verification Results of Model with Intruder

To understand the model behaviour we first perform a simulation check. In UPPAAL the moment any process becomes active it can affect the other processes. When the intruder becomes active and sends or receives messages, the sequence of events changes. In Table 4.2 we show the differences between the two models when we did the safety check for deadlocks. The number of states increased dramatically from 15 states to 749 states. The reason for this increase is that the communication in the secure model was running simultaneously but after we added the intruder process the other processes were affected and the transitions increased.

Model	States Stored
Secure	15
Intruder	749

TABLE 4.2: Verification Results of the Secure Model and Compromised Model in UPPAAL.

4.3 Summary

This chapter presented the second model checker used in our research UPPAAL. We first presented the tool modelling language. We then modelled a web application in four different scenarios based on the requirements of verification. Concluding remarks can be summarised as follows:

- In the model without timer [4.2.1](#), we presented how to model a web application in UPPAAL. We built a model that represents a client and server communicating simultaneously by adding an intermediate committed location between the main locations.
- In the model with dynamic navigation [4.2.2](#) we modelled the dynamic exchange of messages based on different input from the client of the server status. However, we needed to create additional channels to represent the new actions. In the case of the location having two edges, each process will choose a different path.
- In the model with timed constraints [4.2.3](#), we declared a clock in the global variables. We then verified session management properties that rely on time constraints.
- In the model with the intruder [4.2.4](#) we added an intruder process. We found that the number of states has increased. When we used the simulation we noticed that the communication between the server and client became different and with the presence of an intruder in the middle of the session there will be a difference in the locations of both parties at the same time.

In the next chapter we present a comparison between modelling web applications in SPIN and UPPAAL. We discuss the differences between the results obtained from both model checking tools. We also present the challenges of modelling web applications.

Chapter 5

Comparison

In this chapter we compare between the results from SPIN and UPPAAL and highlight final remarks from Chapter 4 and Chapter 5. Firstly, we introduce the SPIN modelling challenges and compare it with UPPAAL. We then compare between the different modelling properties in both tools. We then discuss the intruder model. Finally we list a summary of the chapter.

5.1 Modelling Web Applications in Spin

The SPIN model checker [Holzmann, 2004] was developed to verify communication protocols. It has been used in successful examples [D'Argenio et al., 1997, Joesang, 1995]. SPIN has two main sections, the simulator and the verifier. Firstly in the simulator we can gain insights into the model under development, eliminating design mistakes and analysing the system behaviour. On the other hand, the verifier has an expressive number of verification options beside the LTL reasoning. We presented a full description of the SPIN model checker in Chapter 3

We modelled a simple web application in Chapter 3. We used most of SPIN's features to simulate and verify web applications' properties. The model was developed gradually to ensure that the behaviour is easy to understand and also that the results are correct. We list each model below and discuss the challenges.

The first model in section 3.2.1 had one session of the client logging into his/her online banking account to make a simple transaction and the model then terminates at the end of the run. We examined the basic properties of authentication, navigation and session management at this stage of the model. Since we started with a basic model, evolving and adding functions was considered easy after understanding the basics of the modelling input language PROMELA. The first challenge in the first model is understanding how not to reach a deadlock. When modelling with channels it is possible that there will be a model timeout during the simulation if there is no further action from the processes, and this is not considered a termination of the processes. When using the verifier the result will show an error meaning there is a deadlock. The way to solve this is by examining the channel's send and receive operations. Each send transition must have a receive in the other process in order for the model to be correct.

Understanding the repetition and selection loops could also be a challenge. The selection construct `if-fi` ends when there are no other options to select in the construct, while the repetition loop `do-od` keeps going back to the start and choosing another statement. Both constructs are equally important, for example we mostly used the repetition loop `do-od` client process and we only needed the client to keep sending messages to navigate in the session. The server side mostly used the selection loop `if-fi` to examine and check the statements and conditions before replying.

In the first model simulation and verification section we verified the following properties as listed in the OWASP Application Security Verification Standard [Stock et al.,

2014] as Table 5.1 shows:

Property	Description
1	The page is reachable from the top page and always has a next page in the transition.
2	Every page is reachable from the top page.
3	The top page is reachable from all pages.
4	Eventually a chosen-page is visited.
5	The first page is the login-page and the next page is either the login-error-page or the home-page.
6	Whenever the login-page is visited, the next page is either the login-error-page or the login-success-page.
7	Verify that sessions are invalidated when the user logs out.
8	Verify that all pages and resources require authentication except those specifically intended to be public
9	Verify that all authentication controls are enforced on the server side.
10	Verify that re-authentication is required before any application-specific sensitive operations are permitted as per the risk profile of the application.

TABLE 5.1: Web Applications' Properties [Stock et al. \[2014\]](#).

In the second model in section 3.2.2 we modelled a dynamic page navigation where different input leads to different pages. We kept the model simple and verified dynamic navigation properties. The model illustrates two main principles of modelling; firstly, understanding how the model checker executes the processes improves the accuracy of the results. The statements' execution behaviour in PROMELA is conditional on its "enabledness" [[Holzmann, 2004](#)] and statements are either blocked or enabled based on the condition. Since we use an unbuffered channel we had to ensure that the communication is executed simultaneously, otherwise if any of the parties receives an unexpected message the process will block. We designed a method in which it is possible to include all the possibilities under each page block, and in PROMELA we used the `labels` identifiers to represent pages.

The second modelling principle is deciding whether the model should terminate or have an infinite number of sequences. During the modelling we need to ensure that

the model is correct for the intended verification formula. As stated by [Clarke, 2008]:

“ We used the term Model Checking because we wanted to determine if the temporal formula f was true in the Kripke structure \mathcal{M} , i.e., whether the structure \mathcal{M} was a model for the formula f . Some people believe erroneously that the use of the term model refers to the dictionary meaning of this word (e.g., a miniature representation of something or a pattern of something to be made) and indicates that we are dealing with an abstraction of the actual system under study.”

By ensuring the model will be correct for the formula, we checked more properties such as that the session expires after the client logs out by ensuring that the model is infinite. Furthermore, we varied the sequence of pages on different properties by using LTL.

In the third model we introduce a novel approach for modelling discrete time in SPIN. We add an extra process that controls the time, and we then verify properties that rely on time constraints, such as session management properties. In the method presented in section 3.2.3 we designed a discrete timer to suit the verification requirements of web applications. The properties we verified related to security and session management as listed in The OWASP Application Security Verification Standard Stock et al. [2014]. The assertions added to the model ensured that during an active session the timing will not exceed a fixed period. Though the number of states and transitions has increased, we did not reach a state explosion problem as the model was kept simple to verify the properties efficiently.

The final model in section 3.2.4 includes an intruder that becomes active in the middle of the communication. We examine the difference in the sequence of actions and the timing between a secure model and a compromised model. We use the timer process from model 3.2.3 and we add the intruder process.

The first result to be noticed is that the number of states increased dramatically during the execution of the compromised model in contrast to the secure model. Furthermore, the verification run time increased to 14 seconds in the compromised model and the depth reach its limit of 9999.

The results verifies that the presence of an intruder in the communication increases the number of transitions and states. We used the default safety verification for deadlocks to obtain the results.

The next stage analyses the order of events and timing. We added the macro definition `# define MSG printf("Time is %d n ", timer)` to print the value of the time during the run of the model. After each message received from the server side, SPIN prints out the value of the time, and this simulates a time stamp for the message exchange process. The simulation charts identified that the sequence of transactions were different between the models. The timing values after the first exchange of messages were different.

5.2 Modelling Web Applications in Uppaal

We used UPPAAL to model a simple web application in Chapter 5. The model was developed gradually as we did in the SPIN chapter to ensure that the behaviour is easy to understand and also that the results are correct. We list each model bellow and discuss the challenges.

At the first model in section 4.2.1 we focused on developing an accurate model with the basic properties. We did not include the time constraints at this stage. The first challenge we found is how to model a realistic channel that can pass values with each message. In UPPAAL the update on the edge from the sender is always evaluated

before the update of the edge with the receiver, and the receiver can access the data written by the sender in the same transition. But in our model we needed a condition to check the value of the message, and the guards of the edges are evaluated before the updates are executed, i.e., before the receiver in the communication has access to the value. We modified the conditional two-way synchronous value passing in the 4.2.1 method to suit our model. We added intermediate committed locations between the main locations. The committed locations enforce that the synchronisation is atomic. We used the invariants in the intermediate committed locations as conditions.

In the second model in section 4.2.2 we verified dynamic navigation properties. We used the same value passing method as in the first model 4.2.1. The challenge was that when a location has more than one option both processes will act differently if the same channel is used for both actions. We then added channels for each extra action so the server and client will execute simultaneously. We then verified our approach by verifying that the client cannot bypass the authentication page.

In the third model in section 4.2.3 we added a global clock that we used to control the session. With the time constraints we verified properties such as the session is active and the session does not reach a timeout. We also used locations to verify session management properties such as if the client is on the account page and we checked the time constraints.

In the last model in section 4.2.4, we introduced an intruder process to check the differences between a secure and a compromised model as done in the SPIN model. The number of states increased dramatically from 15 states to 749 states. The reason for this increase is that the communication in the secure model was running simultaneously but after the intruder started receiving and sending messages each process was affected and the transitions increased.

5.3 Comparison

In this section we make comparison between the SPIN and UPPAAL model checkers in verifying web applications. In Table 5.2 we show the difference between the number of states from both model checkers. The number of states increased with the increase in the complexity of each model. Each of the model checkers has a different verification background language. In SPIN each change in the integer values, conditions, and jump statements is stored as a state, whereas in UPPAAL only the changes between locations are stored. In SPIN the number of states increased from 53 states in the *model without time* to 92 states in the *dynamic navigation model* as we added more transactions between the client and the server.

In the models without the intruder in UPPAAL; the server and client exchanged the messages simultaneously. When we added the intruder to the UPPAAL model, there was a change in the execution behaviour of the model as each of the communication parties were in a different location at a specific time unit, hence the increase in the number of states from 15 states to 749 states.

<i>Model</i>	<i>Model without Time</i>	<i>Dynamic Navigation Model</i>	<i>Timed Model</i>	<i>Model with Intruder</i>
SPIN	53	92	1182	14666773
UPPAAL	12	14	15	749

TABLE 5.2: Comparison of Number of States between SPIN and UPPAAL.

In the following part of this section we answer the questions we posed in Chapter 1.6.

Q1 How complex and expressive is the input language of the model checking tool to cover the properties of the web applications?

In SPIN's input language PROMELA becoming familiar with the basis of the language was not complex. The basic processes and channels made it suitable for our research. We benefited from the symbolic names `mtype` that represented the messages in the communication. From the first stages of modelling we managed to have a sequence chart of the client and server communicating during the simulation stage. This advantage assisted in understanding the behaviour of the model and also the PROMELA language itself. The complexity was from the verification using LTL, as it was difficult to understand how to formalize the logic of the model in order to suit the formula during the verification. In some of the LTL formulas we needed to make some changes to the model in order to get the correct results. The process of changes improved the final model, and also developed an understanding of how the verification runs. The PROMELA language is expressive enough to cover most of the web application properties if the modelling process is carried out correctly.

In UPPAAL developing the model using the graphical editor was easy to understand. However, we faced the challenge of having a correct representation of the web application. At the first stages we assumed that both processes in the model would run simultaneously by exchanging messages and moving to the next state. This is not true in UPPAAL, as for modelling web application we needed to add conditions on locations, and value passing through channels as we did in SPIN. The process of creating the channels, messages and conditions in UPPAAL is more complex than in SPIN. The guards of the edges are evaluated before the updates are executed from both processes, So we found that we need to add extra intermediate committed locations the between main locations. When we added the intruder process to the model, we faced the problem of the processes not running simultaneously, hence the increase in the number

of states in the compromised model. A solution to the problem was to create a broadcast channel, but we cannot add time conditions to edges as it is not allowed in broadcast channels. UPPAAL can assist in verifying basic properties such as reachability and invariants, but the verification temporal logic used is a simplified subset of CTL.

Q2 To what extent can property specification language be adaptable to specify web application properties?

In SPIN we found that using the LTL patterns assisted in specifying all the proposed properties in the background chapter. However, in UPPAAL we could not verify properties such as precedence and response formulas to verify the sequence of actions that we used in the SPIN chapter. We only verified in UPPAAL the basic properties along with the clock constraints.

Q3 How capable is the model checker verifying the model without resulting in a state explosion problem?

The approach we used in modelling assisted in avoiding over modelling the design. We started by modelling the simplest possible description of the communication between the client and the server. We then focused on capturing the essential system characteristics to be analysed. We developed the model gradually and used the tools simulation and verification after each addition to the model. Since the model was small we did not face any issues in the verification stage. The advantage of this approach is managing the complexity of modelling and avoiding errors in the early stages.

Q4 How are the results different when integrating a simple timing constraint to SPIN, in contrast to UPPAAL which is based on timed automata specifications?

Adding discrete time to SPIN was a challenging task, The timer process should run simultaneously with other processes in the model. If there was a deadlock in any process, the timer process would reach an infinite loop resulting in a failure execution of the model. Correspondingly, a timer could be integrated to suit the verification requirements. The main advantage of modelling time in SPIN is that we were able to have the value of time in each step of the model. In UPPAAL the value of the clock is treated symbolically and the clock values are not represented in real numbers, but rather as clock constraints.

5.4 Summary

This chapter presented the model results summary obtained from the SPIN and UPPAAL modelling chapters. We then discussed the differences between both model checkers to verify web applications. In the next chapter we summarise the research results, highlight the research limitations and provide directions and areas for future research.

Chapter 6

Conclusion

This chapter summarises the work done in this thesis. In section [6.1](#) we summarise the main results of this research. Section [6.2](#) presents a review of the thesis contributions and outstanding issues. In section [6.3](#) we discuss the directions of future research.

6.1 An Overview of the Research

We started the thesis in Chapter [1](#) with a brief description of web applications challenges, properties, formal methods and the advantages of using model checking for the verification of web applications.

In Chapter [2](#) we summarised the background on web application fundamentals and properties, and provided a comparison of the analysis and verification methods found in the literature. The model checking theory used in this research was then described. We then compared the temporal logics LTL and CTL. We then showed the temporal logic patterns that were used in expressing the web application properties in Chapter [3](#).

In Chapter 3 we presented the SPIN and its input language PROMELA. Firstly, we presented an overview of the tool language and advantages and our justification for using it for modelling web applications. Secondly, as SPIN does not support the modelling time concept, the integration of time into PROMELA is discussed, along with examples from the existing literature. We then used a detailed approach for modelling a web application example (on-line banking web application). We presented discussion of the verification options in SPIN. The result shows that it is possible to model basic web applications properties using either assertions or LTL formulas. We proposed a novel approach for modelling the dynamic behaviour of web applications. We then designed a discrete-time process to verify realistic properties, The verification results proved that it is possible to model time in SPIN and to verify the properties without causing a state explosion problem. Finally, we used the timing process to study the sequence of actions and timestamps when an intruder is added to the model. The results showed a difference in the sequence of actions and timing, along with an increase in the number of states, memory and verification time.

Chapter 4 presented the second model checking tool UPPAAL. We first gave an overview of the tool modelling language and verification options. We then showed a simple example to understand modelling with real time. We then used the same approach shown in 3 to gradually model a web application and to analyse the behaviour of the model after each stage. We then verified the properties using the verifier in UPPAAL.

In Chapter 5 we presented the results from SPIN and UPPAAL. We then compared between both tools in the context of verifying web applications.

6.2 Summary of Thesis Contributions to Research Areas

In this section we review the main contributions achieved in this thesis. We list the contributions in three categories as follows:

6.2.1 Contributions to Model Checking Web Applications

Firstly, the challenges in adopting model checking for the analysis and verification of web applications were critically reviewed. The usage of model checking is examined for the critical properties of web applications, such as security and navigation properties. After providing a sound background on the current challenges in verifying web applications, methods are devised to develop more secure and easy-to-maintain web applications.

Secondly, a novel approach for time constraints was integrated into the SPIN model checker to enable the modelling of web application properties. We used SPIN's existing abilities to design realistic web application models. This enabled the expression of issues, such as modelling session management properties and dynamic navigation properties, where a time-out can lead to different pages. Chapter 3 describes the steps in modelling time constraints in Spin.

In addition to modelling the static properties of web applications, a novel approach was developed for modelling the dynamic properties of web applications [Alalfi et al., 2009], in which a single input can lead to different pages based on time constraints and server state. As highlighted in the literature review, there is a gap in modelling the dynamic navigation properties of web applications. Our research shows how it is possible to

model web applications using the model checking tool's existing capabilities, which is detailed in Chapter 3 and Chapter 4.

6.2.2 Contributions to Model Checking Timed Models of Web Applications

The model checker UPPAAL is presented in the literature [Ben-Ari, 2008, Ruys and Holzmann, 2004] as an alternative in SPIN when modelling real-time systems. By integrating discrete time to SPIN we had the advantage of analysing and understanding the behaviour of the model using the timestamps of actions and messages. Whereas in UPPAAL the value of time was treated symbolically and the clock values are not represented by real numbers, but rather as clock constraints. The differences were analysed in Chapter 5.

6.2.3 Contributions to Modelling Security Properties of Applications

By using our approach of analysing the time sequence and action sequence of different web application models. The difference between a secure session and a compromised session with the presence of an intruder can be identified by the differences between time, number of states and behaviour complexity. Understanding the web application behaviour in different scenarios leads to a better understanding of the behaviour, improved security and more stable development. In addition, analysing different sequences of model runs can assist in developing security detection methods [Carl et al., 2006, Kruegel et al., 2005] that rely on quantitative measurements such as difference in time and memory as presented in Chapter 3.

6.3 Future Work

Web applications are evolving rapidly nowadays, and finding new methods to ensure design correctness is important. Web applications are different to other applications and systems due to their complex structure and technologies [Alalfi et al., 2009, Li and Xue, 2014]. The web applications models we created in our approach can be adapted to other communication protocols such as security protocols and ad-hoc routing protocols. In addition, the security and navigation properties we verified in this work can be used in applications that share the same specifications. Below we discuss future direction:

- To find a link between the research and industry in using model checking. We believe that developing model templates and frameworks for the verification of web applications could assist in simplifying the process. In addition, verification options should be added to each template.
- The implementation used in our research is applicable to other types of applications, such as mobile applications, ad-hoc routing protocols and multi-agent systems.
- Our approach of analysing the sequence of actions with an intruder is added to the model, which could assist in understanding other different types of attack in the area of the quantitative detection of web applications attacks.

Bibliography

- Alalfi, M., Cordy, J., and Dean, T. (2009). Modelling methods for web application verification and testing: state of the art. *Software Testing, Verification and Reliability*, 19(4):265–296.
- Alpuente, M., Ballis, D., Espert, J., and Romero, D. (2010). Model-checking web applications with web-tlr. In Bouajjani, A. and Chin, W.-N., editors, *Automated Technology for Verification and Analysis*, Lecture Notes in Computer Science, pages 341–346. Springer Berlin Heidelberg.
- Alur, R. and Dill, D. L. (1994). A theory of timed automata. *Theoretical computer science*, 126(2):183–235.
- Amnell, T., Behrmann, G., Bengtsson, J., Dargenio, P. R., David, A., Fehnker, A., Hune, T., Jeannet, B., Larsen, K. G., Möller, M. O., et al. (2001). UPPAAL-now, next, and future. In *Modeling and Verification of Parallel Processes*, volume 2067, pages 99–124. Springer.
- Armando, A., Carbone, R., Compagna, L., Li, K., and Pellegrino, G. (2010). Model-checking driven security testing of web-based applications. In *Third International Conference on Software Testing, Verification, and Validation Workshops*, pages 361–370. IEEE.
- Baier, C., Katoen, J.-P., et al. (2008). *Principles of model checking*. MIT press Cambridge.
- Balduzzi, M., Gimenez, C. T., Balzarotti, D., and Kirda, E. (2011). Automated discovery of parameter pollution vulnerabilities in web applications. In *The 18th Annual Network and Distributed System Security Symposium*. Internet Society (ISOC).

- Behrmann, G., David, A., and Larsen, K. (2004). A tutorial on UPPAAL. In *Formal methods for the design of real-time systems*, pages 33–35. Springer.
- Ben-Ari, M. (2008). *Principles of the Spin model checker*. Springer Science & Business Media.
- Bengtsson, J. and Yi, W. (2004). Timed automata: Semantics, algorithms and tools. In *Lectures on Concurrency and Petri Nets*, pages 87–124. Springer.
- Blanchet, B. (2001). An efficient cryptographic protocol verifier based on prolog rules. In *Computer Security Foundations Workshop. Proceedings. 14th IEEE*, pages 82–96. IEEE Computer Society.
- Bosnacki, D. Dams, D. (1998). Integrating real time into spin: A prototype implementation. In *FORTE XI / PSTV XVIII*, pages 423–438. Kluwer, B.V.
- Brinksma, E., Mader, A., and Fehnker, A. (2002). Verification and optimization of a plc control schedule. *International Journal on Software Tools for Technology Transfer*, 4(1):21–33.
- Burstall, R. (1974). Program proving as hand simulation with a little induction. *Information processing*, 74:308–312.
- Carl, G., Kesidis, G., Brooks, R. R., and Rai, S. (2006). Denial-of-service attack-detection techniques. *Internet Computing, IEEE*, 10(1):82–89.
- Casteleyn, S., Daniel, F., Dolog, P., and Matera, M. (2009). *Engineering Web Applications*. Springer-Verlag New York Inc.
- Castelluccia, D., Mongiello, M., Ruta, M., and Totaro, R. (2006). Waver: A model checking-based tool to verify web application design. *Electronic Notes in Theoretical Computer Science*, 157(1):61–76.
- Cenzic (2014). Application vulnerability trends report. Technical report, Cenzic.
- Cimatti, A., Clarke, E., Giunchiglia, F., and Roveri, M. (2000). Nusmv: a new symbolic model checker. *International Journal on Software Tools for Technology Transfer*, 2(4):410–425.

- Clarke, E. M. (2008). The birth of model checking. In *25 Years of Model Checking*, volume 5000 of *Lecture Notes in Computer Science*, pages 1–26.
- Clarke, E. M. and Emerson, E. A. (1982). *Design and synthesis of synchronization skeletons using branching time temporal logic*. Springer.
- Clarke, E. M., Grumberg, O., and Peled, D. (1999). *Model checking*. MIT press.
- Clarke, E. M. and Wing, J. M. (1996). Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643.
- Conallen, J. (1999). Modeling web application architectures with UML. *Communications of the ACM*, 42(10):63–70.
- Corbett, J. C., Dwyer, M. B., Hatcliff, J., et al. (2000). A language framework for expressing checkable properties of dynamic software. In *SPIN Model Checking and Software Verification*, pages 205–223. Springer.
- Corin, R., Etalle, S., Hartel, P., and Mader, A. (2003). On modelling real-time and security properties of distributed systems (extended abstract). *University of Twente, Centre for Telematics and Information Technology*.
- Cremers, C. J. (2008). The scyther tool: Verification, falsification, and analysis of security protocols. In *Computer Aided Verification*, pages 414–418. Springer.
- D’Argenio, P. R., Katoen, J.-P., Ruys, T. C., and Tretmans, J. (1997). The bounded retransmission protocol must be on time! In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 416–431. Springer.
- Davis, J. (2000). System verification technologies. Technical report, Institute for Software Integrated Systems, Vanderbilt University.
- Di Sciascio, E., Donini, F., Mongiello, M., and Piscitelli, G. (2003). Web applications design and maintenance using symbolic model checking. In *Seventh European Conference on Software Maintenance and Reengineering. Proceedings.*, pages 63–72. IEEE.

- Dolev, D. and Yao, A. C. (1983). On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208.
- Donini, F., Mongiello, M., Ruta, M., and Totaro, R. (2006). A model checking-based method for verifying web application design. *Electronic Notes in Theoretical Computer Science*, 151(2):19–32.
- Dwyer, M. B., Avrunin, G. S., and Corbett, J. C. (1999). Patterns in property specifications for finite-state verification. In *Software Engineering, 1999. Proceedings of the 1999 International Conference on*, pages 411–420. IEEE.
- Elks, J. (2011). Man in the middle attack: Focus on SSLStrip. Online <http://www.grin.com/en/e-book/170676/man-in-the-middle-attack-focus-on-sslstrip>. Accessed 2015-01-20.
- Engels, G., Kuster, J., Heckel, R., and Lohmann, M. (2003). Model-based verification and validation of properties. *Electronic Notes in Theoretical Computer Science*, 82(7):133–150.
- Falk, L., Prakash, A., and Borders, K. (2008). Analyzing websites for user-visible security design flaws. In *Proceedings of the 4th symposium on Usable privacy and security*, pages 117–126, New York, USA. ACM.
- Fenton, N. and Bieman, J. (2014). *Software metrics: a rigorous and practical approach*. CRC Press.
- Fisher, M. (2011). *An Introduction to Practical Formal Methods Using Temporal Logic*. John Wiley & Sons.
- Fraternali, P. (1999). Tools and approaches for developing data-intensive web applications: a survey. *ACM Computing Surveys (CSUR)*, 31(3):227–263.
- Frier, A., Karlton, P., and Kocher, P. (1996). The ssl 3.0 protocol. *Netscape Communications Corp*, 18:2780.
- Fundation, O. S. (2010). Data loss database - 2010 yearly report. Online <http://datalossdb.org/reports>. Accessed 2013-05-10.

- Gangan, S. (2015). A review of man-in-the-middle attacks. *Computing Research Repository*.
- Ginige, A. (2002). Web engineering: managing the complexity of web systems development. In *Proceedings of the 14th international conference on Software engineering and knowledge engineering*, pages 721–729. ACM.
- Ginige, A. and Murugesan, S. (2001). Web engineering: An introduction. *MultiMedia, IEEE*, 8(1):14–18.
- Google (2015). Google vulnerability reward program (VRP). Online <http://www.google.co.uk/about/appsecurity/reward-program/>. Accessed 2015-01-15.
- Han, M. and Hofmeister, C. (2006). Modeling and verification of adaptive navigation in web applications. In *Proceedings of the 6th international conference on Web engineering*, pages 329–336. ACM.
- Haydar, M., Boroday, S., Petrenko, A., and Sahraoui, H. (2005). Properties and scopes in web model checking. In *Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 400–404. ACM.
- Haydar, M., Petrenko, A., and Sahraoui, H. (2004). Formal verification of web applications modeled by communicating automata. In *Formal Techniques for Networked and Distributed Systems - FORTE*, pages 115–132. Springer.
- Hesseldahl, A. (2015). Cyber crime still on the rise, using nine basic attack methods. Online <http://recode.net/2015/04/13/cyber-crime-still-on-the-rise-using-nine-basic-attack-methods/>. Accessed: 2015-04-21.
- Hoff, J. (2013). A strategic approach to web application security. Technical report, WhiteHat Security.
- Holzmann, G. (2004). *The SPIN model checker: Primer and reference manual*. Addison-Wesley Professional.
- Holzmann, G. J. (1997). The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295.

- Homma, K., Izumi, S., Takahashi, K., and Togashi, A. (2011). Modeling, verification and testing of web applications using model checker. *IEICE Transactions on Information and Systems*, 94(5):989–999.
- Huang, Y. and Lee, D. (2005). Web application security past, present, and future. In Lee, D., Shieh, S., and Tygar, J., editors, *Computer Security in the 21st Century*, pages 183–227. Springer.
- Hughes, G. E. and Cresswell, M. J. (1996). *A new introduction to modal logic*. Psychology Press.
- Huth, M. and Ryan, M. (2006). *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge University Press.
- Joesang, A. (1995). Security protocol verification using SPIN. In Gregoire, J.-C., editor, *Proceedings of the First SPIN Workshop*, pages 1–9, Montreal, Canada. INRS-Telecommunications.
- Jovanovic, N., Kruegel, C., and Kirda, E. (2006). Pixy: A static analysis tool for detecting web application vulnerabilities. In *IEEE Symposium on Security and Privacy*, pages 258–263. IEEE Computer Society.
- Kappel, G., Proll, B., Reich, S., and Retschitzegger, W. (2006). *Web engineering*. John Wiley & Sons.
- Kourkouli, M. and Hassapis, G. (2005). Application of the timed automata abstraction to the performance evaluation of the architecture of a bank on-line transaction processing system. In *Proceedings of the 2nd South-East European Workshop on Formal Methods SEEFM*, pages 142–153. South East European Research Centre SEERC.
- Kripke, S. (2007). Semantical considerations of the modal logic. *Studia Philosophica, Studia Universitatis Babe-Bolyai*, 1.
- Krishnamurthi, S. (2006). Web verification: Perspective and challenges. *Electronic Notes in Theoretical Computer Science*, 157(2):41–46.

- Kröger, F. (1977). Lar: A logic of algorithmic reasoning. *Acta Informatica*, 8(3):243–266.
- Kruegel, C., Vigna, G., and Robertson, W. (2005). A multi-model approach to the detection of web-based attacks. *Computer Networks*, 48(5):717–738.
- Lee, S. C. and Shirani, A. I. (2004). A component based methodology for web application development. *Journal of systems and software*, 71(1):177–187.
- Li, X. and Xue, Y. (2014). A survey on server-side approaches to securing web applications. *ACM Computing Surveys (CSUR)*, 46(4):54.
- McClure, S., Shah, S., and Shah, S. (2003). *Web hacking: attacks and defense*. Addison-Wesley Professional.
- McMillan, K. (1992). Symbolic model checking: an approach to the state explosion problem. Technical report, DTIC Document.
- Meier, S., Schmidt, B., Cremers, C., and Basin, D. (2013). The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification*, pages 696–701. Springer.
- Mendes, E. and Mosley, N. (2006). *Web engineering*. Springer Science & Business Media.
- Miao, H. and Zeng, H. (2007). Model checking-based verification of web application. In *12th IEEE International Conference on Engineering Complex Computer Systems*, pages 47–55. IEEE Computer Society.
- Microsoft (2011). Microsoft .net framework. Online <http://www.microsoft.com/net>. Accessed 2014-05-13.
- MSDN (2011). Improving web application security: Threats and countermeasures roadmap. Online <http://msdn.microsoft.com/en-us/library/ff649874.aspx>. Accessed 2015-05-12.

- Murugesan, S. and Deshpande, Y. (2002). Meeting the challenges of web application development: the web engineering approach. In *Proceedings of the 24th International Conference on Software Engineering*, pages 687–688. ACM.
- Offutt, J. (2002). Quality attributes of web software applications. *IEEE Software*, 19(2):25–32.
- OWASP (2013). OWASP top ten web applicatins attacks. Online <https://www.owasp.org/>. Accessed 2013-05-16.
- Pnueli, A. (1981). The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13(1):45–60.
- Rescorla, E. (2001). *SSL and TLS: designing and building secure systems*, volume 1. Addison-Wesley Reading.
- Ricca, F. and Tonella, P. (2000). Web site analysis: Structure and evolution. In *Software Maintenance, 2000. Proceedings. International Conference on*, pages 76–86. IEEE.
- Ricca, F. and Tonella, P. (2001). Analysis and testing of web applications. In *ICSE*, pages 25–34. IEEE Computer Society.
- Ruys, T. C. (2003). Optimal scheduling using branch and bound with SPIN 4.0. In *Model Checking Software*, pages 1–17. Springer.
- Ruys, T. C. and Holzmann, G. J. (2004). Advanced SPIN tutorial. In *Model Checking Software*, pages 304–305. Springer.
- Salamah, S., Gates, A., Roach, S., and Mondragon, O. (2005). Verifying pattern-generated LTL formulas: a case study. In *Model Checking Software*, pages 200–220. Springer.
- Schätz, B. (2004). Model-based development: combining engineering approaches and formal techniques. In Davies, J., Schulte, W., and Barnett, M., editors, *Formal Methods and Software Engineering*, volume 3308 of *Lecture Notes in Computer Science*, pages 1–2. Springer Berlin Heidelberg.

- Solutions, V. E. (2015). 2015 data breach investigations report. Online http://www.verizonenterprise.com/DBIR/2015/?utm_source=pr&utm_medium=pr&utm_campaign=dbir2015. Accessed 2015-01-10.
- Stock, A., Kazerooni, S., Cuthbert, D., and Raja, K. (2014). Application security verification standard. Technical report, The OWASP Foundation.
- Taylor, M. J., McWilliam, J., Forsyth, H., and Wade, S. (2002). Methodologies and website development: a survey of practice. *Information and Software Technology*, 44(6):381–391.
- Tracy, M., Jansen, W., and McLarnon, M. (2002). Guidelines on securing public web servers. *NIST Special Publication*, 800:44.
- Tripakis, S. and Courcoubetis, C. (1996). Extending PROMELA and SPIN for real time. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 329–348. Springer.
- Valmari, A. (1998). The state explosion problem. In Reisig, W. and Rozenberg, G., editors, *Lectures on Petri Nets I: Basic Models*, volume 1491 of *Lecture Notes in Computer Science*, pages 429–528. Springer Berlin Heidelberg.
- Vardi, M. Y. (2001). Branching vs. linear time: Final showdown. In *Tools and Algorithms for the Construction and Analysis of Systems*, pages 1–22. Springer.
- Venema, Y. (2001). Temporal logic. *The Blackwell Guide to Philosophical Logic*, pages 203–223.
- WASC (2011). Web application security consortium. Online <http://www.webappsec.org/>. Accessed 2015-01-15.
- Yuen, S., Kato, K., Kato, D., and Agusa, K. (2006). Web automata: A behavioral model of web applications based on the mvc model. *Information and Media Technologies*, 1(1):66–79.

Appendix A

Promela code is listed here as mentioned in Chapter 3.

(A) Model without Time Constraints

```
1 #define p (Server@HomePage)
2 #define q (Server@AccountPage)
3 #define r (Server@PaymentPage)
4 #define s (Server@ConfirmPage)
5 #define z (Server@LogoutPage)
6 #define m (Status == login)
7 mtype = {ok, err, ack, msg1, msg2, msg3, msg4, msg5, msg6,
          sessionID, user, server, password};
8 mtype = {home, account, payment, confirm, logout, error,
          login};

10 typedef Crypt { mtype session, content1}

12 chan network = [0] of {mtype, /* msg# */
13                        mtype, /* receiver */
14                        Crypt};
15 /* global variables for verification*/
16 mtype partnerA, partnerB;
17 mtype statusA = err;
18 mtype statusB = err;
19 mtype page = err;
20 mtype Status = err;
21 active proctype Server() {
```

```

23  mtype sessionId; /* session id that we receive from the
      User */
24  Crypt messageSU; /* encrypted message to the user
      */
25  Crypt data; /* received encrypted message
      */
26  partnerB = user;
27  messageSU.session = sessionId;
28  messageSU.content1 = server;

30 HomePage: page = home;

32      network? msg1 (partnerB, data);

34      if:: (data.session == sessionId) && (data.content1
      == user) ->

36          network ! ack (partnerB, messageSU);

38          statusA = ok; Status = login; goto
      AccountPage;

40      ::else -> goto HomePage;
41  fi;
42  AccountPage:

44      network? msg2 (partnerB, data);
45      if::(data.session == sessionId) && (data.content1
      == account) -> page = account;network ! ack (partnerB,
      messageSU);goto PaymentPage;
46      ::else -> goto HomePage;
47  fi;
48  PaymentPage:

50      network? msg3 (partnerB, data);

```

```

52         if::(data.session == sessionId) && (data.content1
           == payment) -> page = payment; network ! ack (partnerB,
           messageSU); goto ConfirmPage;
53         ::else -> goto HomePage;
54     fi;

56     ConfirmPage:

58     network? msg4 (partnerB, data);
59     if::(data.session == sessionId) && (data.content1
           == confirm) -> page = confirm; network ! ack (partnerB,
           messageSU);
60     ::else -> goto HomePage;
61     fi;
62     Logoutpage: Status = logout

64     network? msg5 (partnerB, data);
65     if::(data.session == sessionId) && (data.content1
           == logout) -> page = home; statusA = err; network ! ack
           (partnerB, messageSU);
66     ::else -> goto HomePage;
67     fi;
68 }

69 active proctype User() {
70     mtype sessionId;      /* nonce that we receive from the
                           Server */
71     Crypt messageUS;      /* encrypted message to Server
                           */
72     Crypt data;           /* received encrypted message
                           */

74 A:
75     partnerA = server;

78 /* login messgae */

```

```
80 messageUS.session = sessionId;
81 messageUS.content1 = user;

84 network! msg1 (partnerA, messageUS);

86 network? ack (partnerA, data);

88 (data.session == sessionId) && (data.content1 == server)

90 statusB = ok;

92 B:
93 /* Second messgae */

95 messageUS.session = sessionId;
96 messageUS.content1 = account;

98 network! msg2 (partnerA, messageUS);
99 network? ack (partnerA, data);

101 C:
102 /* Third messgae */

104 messageUS.session = sessionId;
105 messageUS.content1 = payment;

107 network! msg3 (partnerA, messageUS);
108 network? ack (partnerA, data);

110 D:
111 /* forth messgae */
112 messageUS.session = sessionId;
113 messageUS.content1 = confirm;
114 network! msg4 (partnerA, messageUS);
```

```

115 network? ack (partnerA, data);

117 E:
118 /*  fifth messgae */
119 messageUS.session = sessionId;
120 messageUS.content1 = logout;
121 network! msg5 (partnerA, messageUS);
122 network? ack (partnerA, data);statusB = err;

124 }
125 /*ltl p0 { <> (partnerA==server && partnerB==user) -> <>(
      statusA == ok && statusB == ok) }*/
126 /*ltl p1 { <> (statusA == ok && statusB == ok) }*/
127 /* ltl p2 { []((page == account) -> <>(statusA == ok &&
      statusB == ok) ) }*/
128 /* ltl p3 { [] ((Server@PaymentPage) -> (Status == login))
      } */
129 /*ltl p4 { [] (P -> <>R ) }*/
130 /*ltl p5 { [] (!Q || <>(Q && <>P)) }*/
131 /*ltl m1 { !M W P}*/
132 /*ltl p7 { [] (p && !z -> (!z U (m && !z))) } */
133 /* ltl p8 { [] (z -> <>m) } */
134 /*      ltl p9 { <> (q && m) } */

```

(B) Dynamic Navigation Model

```

1      #define p (Server@HomePage)
2      #define q (Server@AccountPage)
3      #define r (Server@PaymentPage)
4      #define s (Server@ConfirmPage)
5      #define z (Server@Logoutpage)
6      #define m (Status == login)

8 mtype = { ok, err, ack,
9          msg1, msg2, msg3, msg4, msg5, msg6,
10         sessionId, user, server, password
11 }

```

```

13 mtype = {
14     home, account, payment, confirm, logout, error,
        login
15 }
16 typedef Crypt { mtype session, content1 }

18 chan network = [0] of { mtype, /* msg# */
19                          mtype, /* receiver */
20                          Crypt }
21 /* global variables for verification*/
22 mtype partnerA = server
23 mtype partnerB = user
24 mtype statusA = err
25 mtype statusB = err
26 mtype page = err
27 mtype Status = err
28 active proctype Server()
29 {
    mtype sessionId /* session that we receive from
        the User */
30     Crypt messageSU /* encrypted message to the user
        */
31     Crypt data /* received encrypted message
        */

33     messageSU.session = sessionId
34     messageSU.content1 = server

36 HomePage:
37     page = home /* LOGIN ONLY */

39     if
40         :: network ? msg1 (partnerB, data)->
41             (data.session == sessionId) && (data.
                content1 == user) ->
42         network ! ack (partnerB, messageSU)

```



```

43         statusA = ok
44         Status = login
45
46     fi
47
48 AccountPage: assert(Status == login)
49     if
50         :: network? msg2 (partnerB, data)->
51             /* Move to Payment Page*/
52             (data.session == sessionId) && (data.
53 content1 == account) ->
54                 page = account
55                 network ! ack (partnerB, messageSU)
56                 goto PaymentPage
57
58             :: network ? msg5 (partnerB, data) ->
59                 /* Client request logout*/
60
61                 (data.session == sessionId) && (data.
62 content1 == logout) ->
63                     page = home; network ! ack (partnerB,
64 messageSU)
65
66                     statusA = err
67                     Status = logout
68                     goto HomePage
69
70     fi
71
72 PaymentPage: assert(Status == login)
73     if
74         :: network? msg3 (partnerB, data) ->
75             /* Move to Confirmation Page*/
76             (data.session == sessionId) && (data.
77 content1 == payment) ->
78                 page = payment
79                 network ! ack (partnerB, messageSU)

```

```

72         goto ConfirmPage

74         :: network ? msg2 (partnerB, data) ->
           /* Go back to Account Page*/

76         (data.session == sessionId) && (data.
content1 == account) ->
77         page = account
78         network ! ack (partnerB, messageSU)
79         goto AccountPage;

81         :: network ? msg5 (partnerB, data) ->
           /* Client request logout*/

83         (data.session == sessionId) && (data.
content1 == logout) ->
84         page = home; network ! ack (partnerB,
messageSU)
85         statusA = err
86         Status = logout
87         goto HomePage

89     fi
90 ConfirmPage: assert(Status == login)

92     if
93         :: network? msg4 (partnerB, data) ->
           (data.session == sessionId) && (data.
content1 == confirm) ->
94         page = confirm
95         network ! ack (partnerB, messageSU)

96         :: network ? msg2 (partnerB, data) ->
           /* Go back to Account Page*/

```

```

100             (data.session == sessionId) && (data.
               content1 == account) ->
101             page = account
102             network ! ack (partnerB, messageSU)
103             goto AccountPage;

105         fi
106 Logoutpage: assert (Status == login); Status = logout
107             network? msg5 (partnerB, data)
108             if
109                 :: (data.session == sessionId) && (data.content1
               == logout) ->
110                 page = home; network ! ack (partnerB,
               messageSU)
111                 statusA = err
112                 goto HomePage
113             :: else ->
114                 goto HomePage
115         fi
116 }
117 active proctype User()
118 {
119     mtype sessionId /* nonce that we receive from the
               Server */
120     Crypt messageUS /* encrypted message to Server */
121     Crypt data      /* received encrypted message */

122 A:      /* Home Page */
123     messageUS.session = sessionId
124     messageUS.content1 = user
125     network! msg1 (partnerA, messageUS)
126     network? ack (partnerA, data)
127     (data.session == sessionId) && (data.content1 ==
               server)
128     statusB = ok

130 B:      /* Account Page */

```

```

131         do
132             ::messageUS.session = sessionId;messageUS.content1
              = account
133             network! msg2 (partnerA, messageUS)
134             network? ack (partnerA, data)->goto C;
135             ::messageUS.session = sessionId; messageUS.
content1 = logout;
136             network! msg5 (partnerA, messageUS);
137             network? ack (partnerA, data)->statusB = err;
goto A;
138         od;

140 C:      /* Payment Page */
141         do
142             ::messageUS.session = sessionId; messageUS.
content1 = payment;
143             network! msg3 (partnerA, messageUS);
144             network? ack (partnerA, data)->goto D;
145             ::messageUS.session = sessionId; messageUS.
content1 = account;
146             network! msg2 (partnerA, messageUS);
147             network? ack (partnerA, data)->goto B;
148             ::messageUS.session = sessionId; messageUS.
content1 = logout;
149             network! msg5 (partnerA, messageUS);
150             network? ack (partnerA, data)->statusB = err;
goto A;
151         od;

152 D:      /* Confirmation Page */
153         do
154             :: messageUS.session = sessionId
155             messageUS.content1 = confirm
156             network! msg4 (partnerA, messageUS)
157             network? ack (partnerA, data)->goto E;

```

```

159         ::messageUS.session = sessionId; messageUS.
            content1 = account;
160         network! msg2 (partnerA, messageUS);
161         network? ack (partnerA, data)->goto B;

163     od;
164 E:      /* Logout Page */
165         messageUS.session = sessionId
166         messageUS.content1 = logout
167         network! msg5 (partnerA, messageUS)
168         network? ack (partnerA, data)
169         statusB = err
170         goto A
171 }
172 ltl p1 { !r U q }
173 /*ltl r1 { [] (q -> <>r || <> z || <>p) }*/

```

(C) Model with Timed Constraints

```

1      #define p (Server@HomePage)
2      #define q (Server@AccountPage)
3      #define r (Server@PaymentPage)
4      #define s (Server@ConfirmPage)
5      #define z (Server@Logoutpage)
6      #define m (timer < MAX)
7      #define MSG printf("Status is %e\n", Status)

9      mtype = { ok, err, ack,
10              msg1, msg2, msg3, msg4, msg5, msg6,
11              sessionId, invalid, valid, user, server, password
12          }

14      mtype = {
15          home, account, payment, confirm, logout, error,
            login
16      }

```

```

18 typedef Crypt { mtype session, content1 }

20 chan network = [0] of { mtype, /* msg#      */
21                          mtype, /* receiver */
22                          Crypt }
23 /* global variables for verification*/
24 mtype partnerA = server
25 mtype partnerB = user
26 mtype statusA = err
27 mtype statusB = err
28 mtype page = err
29 mtype Status = err
30 mtype Session = err
31 byte timer ;
32 byte MAX = 2;

34 byte flag;
35 byte flag2;
36 byte flag3;
37 byte flag4;
38 proctype Clock()

40 {
41     do
42         ::atomic {timer < MAX -> timer ++} skip;
43         ::else -> atomic {timer == MAX -> timer = 0} skip;
44     od;
45 }

46 proctype Server()
47 {
48     mtype sessionId /* session that we receive from
the User */
49     Crypt messageSU /* encrypted message to the user
*/
50     Crypt data      /* received encrypted message
*/

```

```

53 HomePage:
54     page = home /* LOGIN ONLY */
55     messageSU.session = sessionId
56     messageSU.content1 = server

57
58     if
59         :: network ? msg1 (partnerB, data)-> atomic {
60             (data.session == sessionId) && (data.
content1 == user) ->
61                 network ! ack (partnerB, messageSU)
62                 statusA = ok
63                 Session = valid
64                 Status = login }
65     skip;
66     fi

67
68 AccountPage: /*assert (timer <= MAX && Session == valid &&
Status == login)*/

69
70     if
71         :: network? msg2 (partnerB, data)-> if
72
73             /*
Move to Payment Page*/
74             :: atomic {(data.session == sessionId) &&
(data.content1 == account)&&(timer < MAX) ->
75                 page = account ;timer = 0

76                 network ! ack (partnerB, messageSU)
77                 goto PaymentPage;} skip;
78             ::else-> atomic {Session = invalid;
79                 network!error (partnerB, messageSU);Status =
logout;flag ++;MSG;goto HomePage;} skip;
80
81                 fi;
82     fi

```

```

83 PaymentPage:      /*assert (timer <= MAX && Session ==
                        valid && Status == login)*/

85         if
86         :: network? msg3 (partnerB, data) -> if
87
88             /* Move to Confirmation Page*/
89             ::atomic {(data.session == sessionId) && (
                        data.content1 == payment) &&(timer < MAX)->
89                 page = payment;timer = 0
90                 network ! ack (partnerB, messageSU)
91                 goto ConfirmedPage;} skip;

93         :: else -> atomic {Session = invalid;
94                 network!error (partnerB, messageSU);
                        Status = logout;flag2 ++;MSG;goto HomePage;} skip;
95         fi
96     fi

98 ConfirmedPage:    /*assert (timer <= MAX && Session == valid
                        && Status == login)*/

100         if
101         :: network? msg4 (partnerB, data) -> if
102             :: atomic {(data.session == sessionId) && (
                        data.content1 == confirm) &&(timer < MAX)->
103                 page = confirm;timer = 0
104                 network ! ack (partnerB, messageSU);}skip;

106         :: else -> atomic {Session = invalid;
107                 network!error (partnerB, messageSU);
                        Status = logout;flag3 ++;MSG;goto HomePage;}skip;
108         fi

110     fi

```



```

112 Logoutpage: /*assert (timer <= MAX && Session == valid &&
    Status == login)*/
113     network? msg5 (partnerB, data)
114     if
115         :: atomic {(data.session == sessionId) && (data.
    content1 == logout) &&(timer < MAX)->
116         page = logout;network ! ack (partnerB,
    messageSU)
117         statusA = err
118         Status = logout
119         timer = 0
120         Session = invalid;
121         goto HomePage;}skip;

123         :: else -> atomic {Session = invalid;
124             network!error (partnerB, messageSU);
    Status = logout;flag4 ++;MSG;goto HomePage;} skip;
125     fi;
126 }

128 proctype User()
129 {
    mtype sessionId /* nonce that we receive from the
    Server */
130     Crypt messageUS /* encrypted message to Server */
131     Crypt data      /* received encrypted message */

133 A:      /* Home Page */
134     messageUS.session = sessionId
135     messageUS.content1 = user

137     network! msg1 (partnerA, messageUS)
138     network? ack (partnerA, data)
139     if ::
140         (data.session == sessionId) && (data.
    content1 == server) ->

```

```

141         statusB = ok ;skip; fi;

143 B:      /* Account Page */
144         do
145             ::messageUS.session = sessionId;messageUS.content1
               = account
146             network! msg2 (partnerA, messageUS)
147             if
148                 :: network? ack (partnerA, data)->goto C
               ;skip;
149                 :: if ::network? error (partnerA, data)
               -> statusB = err; goto A; skip; fi;
150             fi;
151         od;

153 C:      /* Payment Page */
154         do
155             ::messageUS.session = sessionId; messageUS.
               content1 = payment;
156             network! msg3 (partnerA, messageUS);
157             if
158                 ::network? ack (partnerA, data)->goto D;
               skip;
159                 ::if::network? error(partnerA, data)->
               statusB = err;goto A; skip;fi;
160             fi;

162         od;

163 D:      /* Confirmation Page */
164         do
165             :: messageUS.session = sessionId
166             messageUS.content1 = confirm
167             network! msg4 (partnerA, messageUS)
168             if
169                 ::network? ack (partnerA, data)->goto E;
               skip;

```

```

170             ::if::network? error (partnerA, data)->
                statusB = err;goto A; skip;fi;
171             fi;
172         od;

174 E:         /* Logout Page */
175         do
176             ::messageUS.session = sessionId
177             messageUS.content1 = logout
178             network! msg5 (partnerA, messageUS)
179             if

181             ::network? ack (partnerA, data)-> statusB
                = err; goto A;skip;
182             ::if::network? error(partnerA, data) ->
                statusB = err;goto A; fi; skip;

184             fi;
185         od;
186 }

187 /*ltl p1 { [] (q -> [] (!m)) }*/
188 /*ltl u5 { [] ((Session == valid) && !(Session == invalid)
    -> ((Status == logout) W (Session == invalid))) }*/
189 /*ltl e3 { [] (!Server@PaymentPage || <>(Server@PaymentPage
    && <> (!timer == MAX))) }*/
190 /*ltl a1 { <> (Session == valid) || (Session == invalid)
    || (Session == 0) }*/
191 /*ltl p1 { <> (statusA == err && statusB == err) }*/
192 /*ltl p2 { ![] ((Status == logout) -> <>(statusA == ok &&
    statusB == ok) ) }*/
193 /*ltl p3 { [] ((Server@PaymentPage) -> [] (!timer == MAX))
    }*/
194 init{
195     run User(); run Clock(); run Server()}

```

(D) Model with an Intruder A: Secure Model

```

1      #define MSG printf("Time is %d\n", timer)

3 mtype = { ok, err, ack,
4          msg1, msg2, msg3, msg4, msg5, msg6,
5          sessionId, user, server, intruder,
6      }

8 mtype = {
9     home, account, payment, confirm, logout, error,
    login
10 }

12 typedef Crypt { mtype session, content1 }

14 chan network = [0] of { mtype, /* msg#      */
15                        mtype, /* receiver */
16                        Crypt }

18 /* global variables for verification*/
19 mtype partnerA;
20 mtype partnerB;
21 mtype statusA = err
22 mtype statusB = err
23 mtype page     = err
24 mtype Status   = err
25 byte timer;
26 byte MAX = 20;

29 proctype Clock()
30 {

32     do
33         ::atomic {timer < MAX -> timer ++} skip;
34         ::else -> atomic {timer == MAX} skip;
35         break

```

```

36         od;

39     }

40     proctype Server()
41     {
42         Crypt messageSU /* encrypted message to the user
43         */
44         Crypt data      /* received encrypted message
45         */
46         HomePage:
47         page = home
48         atomic{network ? msg1, partnerB (data);MSG
49             network ! ack (partnerB, messageSU);
50             statusA = ok
51             Status = login} skip;

53     AccountPage:
54         atomic{network? msg2, partnerB (data);MSG

56             page = account
57             network ! ack (partnerB, messageSU);
58             goto PaymentPage} skip;

60     PaymentPage:
61         atomic{network? msg3, partnerB (data);MSG

63             page = payment
64             network ! ack (partnerB, messageSU);
65             goto ConfirmedPage} skip;

68     ConfirmedPage:
69         atomic{network? msg4, partnerB (data);MSG

```

```

71         page = confirm
72         network ! ack (partnerB, messageSU)} skip;

76 Logoutpage:
77     atomic{network? msg5, partnerB (data);MSG

79         page = home;network ! ack (partnerB,
messageSU);
80         statusA = err
81         Status = logout; goto HomePage} skip;

84 }
85 proctype User()
86 {
87     Crypt messageUS /* encrypted message to Server */
88     Crypt data      /* received encrypted message */

90     if
91     ::partnerA = intruder;
92     ::partnerA = server;
93     fi;

95 A:    /* login message */
96     atomic{messageUS.session = sessionId
97     messageUS.content1 = login
98     network! msg1 ,partnerA, messageUS;
99     network? ack (partnerA, data);
100     statusB = ok} skip;

102 B:    /* second message */
103     atomic{messageUS.session = sessionId
104     messageUS.content1 = account

```

```

105         network! msg2 (partnerA, messageUS);
106         network? ack (partnerA, data);} skip;

108 C:      /*  third message */
109         atomic{messageUS.session = sessionId
110         messageUS.content1 = payment
111         network! msg3 (partnerA, messageUS);
112         network? ack (partnerA, data);} skip;

114 D:      /*  fourth message */
115         atomic{messageUS.session = sessionId
116         messageUS.content1 = confirm
117         network! msg4 (partnerA, messageUS);
118         network? ack (partnerA, data);} skip;

120 E:      /*  fifth message */
121         atomic{messageUS.session = sessionId
122         messageUS.content1 = logout
123         network! msg5 (partnerA, messageUS);
124         network? ack (partnerA, data);
125         statusB = err; goto A} skip;

128 }

130 init {
131     atomic{
132     if
133         ::run User(); run Clock(); run Server();
134     fi;
135     }
136 }

```

B: Model with intruder

```

1      #define MSG printf("Time is %d\n", timer)

```

```

3 mtype = { ok, err, ack,
4           msg1, msg2, msg3, msg4, msg5, msg6,
5           sessionId, user, server, intruder,
6 }

8 mtype = {
9           home, account, payment, confirm, logout, error,
           login
10 }

12 typedef Crypt { mtype session, content1 }

14 chan network = [0] of { mtype, /* msg# */
15                          mtype, /* receiver */
16                          Crypt }

18 /* global variables for verification*/
19 mtype partnerA;
20 mtype partnerB;
21 mtype statusA = err
22 mtype statusB = err
23 mtype page     = err
24 mtype Status   = err
25 byte timer;
26 byte MAX = 10;
27 proctype Clock()

29 {

31           do
32           ::atomic {timer < MAX -> timer ++} skip;
33           ::else -> atomic {timer == MAX} skip;
34           break
35           od;

```



```

38 }
39 proctype Server()
40 {
41     Crypt messageSU /* encrypted message to the user
42                     */
43     Crypt data      /* received encrypted message
44                     */
45     messageSU.session = sessionId
46     messageSU.content1 = 0
47 HomePage:
48     page = home
49     atomic{network ? msg1, partnerB (data);MSG
50             network ! ack (partnerB, messageSU) ;
51             statusA = ok
52             Status = login} skip;
53
54 AccountPage:
55     atomic{network? msg2, partnerB (data);MSG
56
57             page = account
58             network ! ack (partnerB, messageSU);
59             goto PaymentPage} skip;
60
61 PaymentPage:
62     atomic{network? msg3, partnerB (data);MSG
63
64             page = payment
65             network ! ack (partnerB, messageSU);
66             goto ConfirmedPage} skip;
67
68 ConfirmedPage:
69     atomic{network? msg4, partnerB (data);MSG
70
71             page = confirm
72             network ! ack (partnerB, messageSU)} skip;

```

```

75 Logoutpage:
76     atomic{network? msg5, partnerB (data);MSG

78         page = home;network ! ack (partnerB,
        messageSU);MSG
79         statusA = err
80         Status = logout; goto HomePage;} skip;
81 }
82 proctype User()
83 {
84     Crypt messageUS /* encrypted message to Server */
85     Crypt data      /* received encrypted message */

87     if
88     ::partnerA = intruder;
89     ::partnerA = server;
90     fi;

92 A:    /* login message */
93     atomic{messageUS.session = sessionId
94     messageUS.content1 = login
95     network! msg1 ,partnerA, messageUS;
96     network? ack (partnerA, data);
97     statusB = ok} skip;

99 B:    /* second message */
100     atomic{messageUS.session = sessionId
101     messageUS.content1 = account
102     network! msg2 (partnerA, messageUS);
103     network? ack (partnerA, data);} skip;

105 C:    /* third message */
106     atomic{messageUS.session = sessionId

```

```

107         messageUS.content1 = payment
108         network! msg3 (partnerA, messageUS);
109         network? ack (partnerA, data);} skip;

111 D:      /* fourth message */
112         atomic{messageUS.session = sessionId
113         messageUS.content1 = confirm
114         network! msg4 (partnerA, messageUS);
115         network? ack (partnerA, data);} skip;

117 E:      /* fifth message */
118         atomic{messageUS.session = sessionId
119         messageUS.content1 = logout
120         network! msg5 (partnerA, messageUS);
121         network? ack (partnerA, data);
122         statusB = err; goto A} skip;

124 }

126 proctype Intruder() {
127     mtype msg, recpt;
128     Crypt data, intercepted;
129     do
130         :: network ? msg, _, data ->
131         atomic{ if /* store the message */
132             :: intercepted.session = data.session;
133             intercepted.content1 = data.content1;
134             :: skip;
135         fi; }

138         if /* choose a recipient */
139             :: recpt = user;
140             :: recpt = server;
141             :: skip;
142         fi;

```

```

143         if /* replay intercepted message or assemble it */
144             :: data.session      = intercepted.session;
145             data.content1 = intercepted.content1;

147         :: if /* assemble content1 */
148             :: data.content1 = server;
149             :: data.content1 = user;
150             :: data.content1 = login;
151             :: data.content1 = home;
152             :: data.content1 = account;
153             :: data.content1 = payment;
154             :: data.content1 = confirm;
155             :: data.content1 = logout;
156             :: skip;
157         fi;

159         :: if /* assemble SessionId */
160             :: data.session = sessionId;
161             :: skip;
162         fi;
163     fi;
164     network ! msg (recpt, data);
165 od
166 }

168 init {
169     atomic{
170         if
171             :: run User(); run Clock(); run Intruder(); run
Server();
172         fi;
173     }
174 }

```