Analysing Safety-Critical Systems and Security Protocols with Abstract State Machines

Thesis submitted in accordance with the requirements of the University of Liverpool for the degree of Doctor in Philosophy by

Farah Al-Shareefi

September 2019

Dedication

To my beloved family, especially my husband and sons.

List of Abbreviations

ASM	Abstract State Machines
ASM-STPA-SA	Abstract State Machines-System Theoretic Process Analysis-Safe Design
AsmetaL	ASMETA Language
AsmetaS	ASMETA Simulator
AsmetaSMV	ASMETA SMV
AsmetaV	ASMETA Validator
DoS_REPL	Denial of Service REPLay
INTRL	INTeRLeaving
IPCS	Insulin Pump Control System
MITM	Man In The Middle
REFL	REFLection
SASL	Simple Authentication and Security Layer
Simple_REPL	Simple REPLay
STPA	System Theoretic Process Analysis
TDC	Train Door Controller

Abstract

The research presented in this thesis is directed at the analysis of critical systems and protocols and the improvement of their safety and security aspects respectively, by combining formal and informal analysis methods. More specifically, we focus at combining Abstract State Machines (ASMs) as a formal method, with System Theoretic Process Analysis (STPA) as a safety analysis technique, with the aims of developing safer systems and more secure protocols. The ASM method was chosen due to both its generality in specifying any system at a convenient level of abstraction, and its specification which is supported by different formal analysis activities. While the reason for choosing STPA was its capability of eliciting safety requirements originated from inadequate control actions which affect the whole system functions.

The first contribution of this thesis is a methodology to analyse safety-critical systems by capturing both the formal representation of ASM and the safety requirements generated by the STPA. This has the advantages of verifying the STPA requirements in a formal way and giving insights to improve the ASM specification, depending on these requirements. We illustrated this methodology by applying it to a train door controller and an insulin pump control system case studies, showing what safety issues it highlighted.

The second contribution of the work presented in this thesis is a systematic methodology for analysing security protocols. This methodology was intended to provide a link between the formal simulation of external attack scenarios and protocol under analysis specified by the ASM method, and the analysis outcomes of a proposed technique called FATI method. The FATI (Flaws and Attack Types Identification) method is an inspired form of STPA that applies queries on each protocol action to determine the possible protocol flaws and their expected attack types. The identified attack types help to select the attack scenario specifications whose simulations are likely to produce attacks. Our methodology also minimized the number of the simulated protocol runs by reducing the number of intruder's messages through considering the receiver's expectations about the message type format and the content. Furthermore, we showed how to analyse protocols in the presence of an algebraic property for the commutative encryption. Our methodology for analysing security protocols was applied to several protocols. Moreover, within the security protocols area, we clarified the ambiguous requirements for simple authentication and security layer example depending on the ASM method.

Acknowledgements

My great full thanks to God, the Most Merciful and the Most Gracious, for giving me the strength and patience to overcome the challenges in my work and life while conducting my Ph.D study.

I would like to express my deepest gratitude to both supervisors Dr. Alexei Lisitsa and Dr. Clare Dixon for their ongoing support, patience, constructive criticism and encouragement throughout my Ph.D. journey. Their continuous guidance and valuable comments have made the completion of my Ph.D. possible and enjoyable. It has been a great privilege to have worked with you both.

I would also like to extend my deepest gratitude to the Higher Committee for Education Development in Iraq (HCED) for their generous financial support that has given me the opportunity to complete my Ph.D. study.

I am also thankful to my advisors Prof. Sven Schewe and Prof. Boris Konev for their advice and helpful comments.

Thanks also to all staff members and colleagues in the Department of Computer Science at the University of Liverpool who have been helpful whenever necessary.

A special thanks go to my family: father, mother, father-in-law, mother-in-law, sisters, brother, sister-in-law, and brothers-in-law, for their prayers, support, and encouragement.

Last but not least, my warmest thanks go to my sons and my husband who have given me constant love during the completion of the thesis. I apologize for every moment I missed to care about you all or to share your ups and downs. My husband, you are fabulous. I would never be able to get where I am without you and your help. Please accept my deepest thanks.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

This thesis considers analysing safety-critical systems and security protocols by combining the ASM method with STPA technique for critical systems, and with STPA-like technique for protocols.

1.1 Safety-Critical Systems

An increasing number of embedded systems are applied everywhere and are designed for specific purposes. Some of these systems are designed to operate in situations at which failure can lead to a safety issue. These are called *safety-critical systems* [134], which are considered in the context of this thesis. Safety-critical systems have a crucial contribution to many sectors in human life, such as medical care, transportation, automotive, aerospace, avionics, military, chemical applications, etc. Along with their contributions to the prosperity of the community, different accidents yield from critical system failures, for instance, the explosion of Chernobyl nuclear reactor as a result of flawed reactor design coupled with human mistake led to the death of many people and thousands of cancer cases among children afterward [241], overdoses from Therac-25 radiation therapy machine used to treat cancer resulted in eight people died and six injured [153], when Patriot missile defence system failed to destroy the incoming SCUD missiles from Iraqi army during the second Gulf war, due to imprecise arithmetic calculations, twenty-eight American soldiers were killed and ninety-eight were injured [210], and the crash of Ethiopian and Indonesian Boeing 737 MAX 8 planes caused by errors in position sensors leading to 346 people being killed [176]. Such accidents raise awareness of ensuring safety aspects for these systems

when they are developed, where *safety* means the system's ability to function without endangering its users and environment [220].

Moreover, modern systems have become more complex, both architecturally and functionally, as well as becoming more reliant on software that controls and monitors their critical functions. Hence, analysing and developing safe designs for safety-critical systems are challenging processes.

There is a great emphasis on using formal methods to address the analysis and development challenges. Formal methods are applied at the early stages of system development life cycle to detect the design faults as early as possible and to increase confidence in the correctness of the system behaviour. These methods are mathematically grounded languages and analysis tools for performing the following [80]:

System modelling which is a conceptual abstraction describing and representing the system behaviour [236].

Model Simulation is a process of carrying out experiments with a designed model to assess the performance of the system under different operational conditions [236].

Model Validation is a process of ensuring that the system model adequately satisfies the user requirements and expectations [44].

Model Verification is a rigorous evaluation process to check whether a system model fully satisfies the systems and users requirements [44].

In this thesis, all of the above activities are used for safety-critical systems, while for security protocols only the first two are applied. Furthermore, from the available techniques for conducting the model verification for critical systems, such as theorem-proving and model checking, this thesis is concerned with using the model-checking technique. This technique is an automatic method for exhaustive checking whether a finite state model of a system meets the desired requirements [79].

The model-checking helps to deliver the model correctness that is adherent to functional requirements. However, adherence to the functional requirements is insufficient to ensure system safety. Therefore, we need to verify satisfaction of safety requirements to ensure as safe as possible system behaviour.

System safety can be assessed by obtaining safety requirements through applying informal safety analysis techniques from system engineering domain. Safety analysis techniques

enable analysts to identify the system hazards and to elicit the safety requirements that mitigate or prevent those hazards [96]. Therefore, combining the formal verification with the safety analysis outcomes is beneficial for safety-critical systems, though, this combination faces the difficulty of expressing the outcomes of the safety analysis techniques in a formal way. The work in this thesis is directed at dealing with this difficulty to achieve the required combination.

In the following section we state the second topic that has been considered by this thesis.

1.2 Security Protocols

Security protocols form the building blocks of infrastructures required for secure communication over public networks. A security protocol is essentially a predefined sequence of actions performed by multiple communicating participants to achieve security-critical function(s), such as authentication, confidentiality, integrity, etc. [212]. These functions are accomplished by using cryptographic mechanisms, and due to this, security protocols are also called cryptographic protocols.

Security protocols are increasingly employed in our daily life online applications, such as banking, shopping, commerce, election, etc. Failure of such protocols can have negative financial, political, and social impacts, so the security of these protocols is a major concern. By *security* we mean the protocol ability to operate without harmfully influencing the performance of its intended functions. Therefore, developing a secure protocol is a vital and highly desirable task. Unfortunately, security protocols are notoriously difficult to analyse due to the implicit assumptions or sometimes unclear details about the environment in which the protocol operates, the intruder capabilities, the features of the employed cryptographic mechanism, etc. The work in this thesis considers analysing security protocols and specifying clear design.

A great deal of research work in formal methods has been conducted towards the development and analysis of security protocols. These methods operate under Dolev-yao intruder [91] assumptions which state the following: (1) unbounded number of messages can be generated by the intruder based on its knowledge; (2) an encrypted message cannot be broken without knowing the decryption key; and (3) unbounded number of protocol sessions(executions) can be launched by the intruder.

The first assumption may lead to some superfluous messages with wrong format or

type, are likely to be rejected by the honest participants. The second assumption may be too strong in some contexts. In particular, some encryption mechanisms with certain algebraic properties can be broken [141], for instance commutative encryption [166]. The third assumption together with the first one generally lead to unbounded state space to be analysed. In this case, methods that use model-checker will face the state space explosion problem, but they usually deal with this problem by drastically simplifying these assumptions to keep the state space small and finite. Some tools, e.g., ProVerif [51], and Tamarin [168], have addressed infinite state space problem by utilizing a finite characterization of the infinite state space whenever possible; but this is at the cost of possible non-termination of the analysis, particularly in the presence of non-trivial algebraic properties, such as the one of commutative encryption. The work in [129] copes with the infinite state space problem by simulating security protocols using scenarios designed for external attacks. A scenario is a protocol run composed of multiple sessions in which the pattern of ordering the protocol steps for the involved sessions is specified. This approach is an effective way of reducing the number of protocol runs while looking for an attack by simulation. However, the intruder in work [129] composes messages using only type matching restriction, which may still increase the message space with unacceptable messages. Furthermore, in work [129] all the attack scenarios are simulated even those whose simulation will definitely not produce attacks. Moreover, analysing security protocols in the presence of commutative encryption algebraic property has not been shown in [129]. This thesis deals with these three limitations in [129].

Besides works in formal methods, there are two attempts to develop informal techniques for analysing security protocols based on safety-critical system hazard analysis techniques [107, 108]. These attempts help to identify and address elementary vulnerabilities and to elicit requirements to reduce the likelihood of these vulnerabilities occurrence. However, in these attempts, it has not been shown how to utilize the output of their techniques in the next development phases which utilize formal methods.

Like its beneficial role in safety-critical systems, the collaboration between formal and informal analysis methods can play a similar contribution to security protocols. The work in this thesis is directed at this collaboration.

Another point that needs more attention is the silent assumptions and unclear details in the described requirements for such protocols. Most security protocols are described by RFCs or standards using natural language which has inherent imprecision and ambiguity, such the Simple Authentication and Security Layer (SASL) [169, 173]. Therefore, it is

necessary to make certain that security protocols requirements are clear before performing the analysis task.

As our work focuses on combining a formal method and an informal safety analysis technique to analyse safety-critical systems and security protocols, we looked for the best such combination that can be applied to both topics. So, we chose the Abstract State Machine (ASM) method [64, 60] from formal methods, and from informal safety analysis techniques we selected System Theoretic Process Analysis (STPA) technique [149].

The reasons for choosing the ASM method are the following. First, the generality feature which makes this method flexible enough to specify any system, whether critical or protocol, at a required level of abstraction. Second, the ASM method has simple and rich syntax along with accurate semantics which will together help to specify high fidelity models balancing between abstraction and preciseness. Third, the ASM method underpins the analysis of complex systems, and this satisfies our need to support formal analysis activities to improve system safety and protocol security. Fourth, ASM method uses two concepts: *ground model*, for capturing the informal requirements in an abstract, precise, and understandable way, and *stepwise refinement*, for gradual refining of the ground model into a more detailed and concrete model. These notions together are useful to clarify the requirements for complex protocols, such as SASL.

While we chose the STPA technique because it identifies safety requirements to mitigate or avoid hazards resulting from interaction faults among system components, component failures, requirement flaws, human mistakes, software errors, or inadequate control in system design. The STPA views the system as a control loop consisting of a controller, actuator, sensor, and a controlled process. Eliciting the safety requirements by STPA is based on some queries guided by time provided conditions on every control action issued by the controller. Recently, the STPA technique has been used, in [17, 19], as an integrated tool with verification activity by supplying it with the formulated safety requirements. However, the formalization process of safety requirements does not accurately capture some of the temporal aspects of the requirements. To handle this situation, the work in this thesis focuses on presenting adequate formalization to STPA requirements. Concerning security protocols, the STPA queries are not that helpful because typical attacks against protocols are not necessarily about timing, rather, they are about some other security-relevant aspects for each protocol action. Therefore, part of the work in the context of this thesis concerns with developing the STPA technique in a way that serves security protocols..

1.3 Research Questions and Contributions

Analysing safety-critical systems and security protocols is a challenging problem. This problem is dealt with in our thesis by combining a formal method, which is Abstract State Machines (ASMs), with an informal analysis technique, which is System Theoretic Process Analysis (STPA), to assess both safety and security of critical systems and protocols. Therefore, our overall goal is to investigate the ability of this combination to develop safe systems and secure protocols. This goal can be formulated in a research question as follows:

Can ASM be combined with STPA to analyse safety aspects of critical systems and security aspects of protocols?

To provide an answer to this research question, the following research sub-questions need to be addressed:

1) Can the STPA technique help to improve the ASM specifications concerning safety issues? How can we formulate the STPA safety requirements to be used in further verification?

2) Can the STPA technique be developed to analyse security protocols?

3) Given the answer to the above, in which analysis activity (simulation, verification), should the outcomes of the developed STPA technique be considered to serve the analysis of security protocols?

4) How to clarify ambiguous requirements for security protocols using the ASM method?

In the context of addressing the above research questions, the following list summarizes the main contributions of the work presented in this thesis.

1) A systematic methodology for analysing and developing safety-critical systems. This work is based on the idea of combining ASM with STPA, to develop safe specifications, and to provide an adequate and concise temporal formalization of the STPA requirements. In this methodology, we formalized the outcome of the STPA technique as LTL safety properties, and we utilized the validation and verification tools developed around ASMs to guide the modeller to redesign the ASM model. Guiding the modeller is attained through violation detection of the functional and safety requirements. This work is presented in Chapter 5.

2) A well-guided analysis method, called Flaws and Attack Types Identification (FATI), for determining the possible protocol flaws and attack types. This method is STPA-like, where it employs on every protocol action a set of queries whose answers facilitate recognizing potential flaws, which then will be assessed to determine the possible attack types that can exploit these flaws. The employed queries are essentially concerned with participants' liveness and messages freshness. This is because most of the attacks upon security protocols exploit flaws resulting from the falsely alive participant in the current run, and/or the message or part of it is not indeed fresh. The applicability of this method is shown to two security protocols: AS_RPC and NSPK protocols. This work is presented in Chapter 6.

3) A methodology for analysing security protocols by considering the combination of formal and informal analysis methods. This methodology is based on simulating security protocols using attack scenarios [129]. Typically, our methodology consists of two phases: the manual and the automatic analysis phases. In the manual analysis phase, we avoid wasting efforts in simulating all the specified scenarios through the application of FATI method to identify only the possible attack types whose corresponding scenarios will be simulated. While in the automatic analysis phase, we implement the idea of attack pattern scenarios in the ASM methodology. We also formulate a principle for generating messages by the intruder according to the receiver's expectations that are related to the message content and type format. This principle helps to further reduce the number of protocol runs during the simulation process. Furthermore, we show how to analyse protocols in the presence of an algebraic property for commutative encryption. This work is detailed in Chapter 6.

4) Comprehensible specifications for the Simple Authentication and Security Layer (SASL) protocol. By these specifications, we explicated ambiguities of the SASL informal descriptions in RFCs and Oracle implementation documents based on two ASM notions: ground model and stepwise refinement. The ground model enabled us to reflect the desired behaviour, which is explained in RFCs. While the stepwise refinement helped us to illustrate the ambiguous part of the desired behaviour accurately, using other document sources. This work is stated in Chapter 7.

1.4 Publications

Three peer-reviewed publications have been arisen out of the work presented in this thesis. These are listed below together with a brief description of each.

1) **Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon. "Abstract state machines and system theoretic process analysis for safety-critical systems". In Proceedings of 20^{th} International Conference on Formal Methods: Foundations and Applications (SBMF 2017), vol. 10623, pp. 15-32, Springer, 2017**.

   This was the first paper resulting from the work presented in this thesis. In this paper, what was described is a methodology for analysing safety-critical systems based on the combination between the ASM method and the STPA technique. This methodology captures both the formal representation of ASM with the ability to generate safety properties from the STPA hazard analysis. This has the advantages of verifying the STPA requirements formally and giving insights for the improvement of the ASM specification, depending on these requirements. The methodology in this paper has been applied to an insulin pump control system case study, to show what safety issues it highlights. The content of this paper is included in Chapter 5.

2) **Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon. "Clarification of ambiguity for the simple authentication and security layer". In Proceedings of 6^{th} International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ 2018), vol. 10817, pp. 189-203, Springer, 2018**.

   This paper was intended to show how can the ASM method be employed to deal with the problem that the silent assumptions and unclear details in the security protocol requirements can sometimes result in the flawed protocol design. This was realized by choosing the Simple Authentication and Security Layer (SASL) as an example to clarify its behaviour in terms of ASMs. This example is informally described in RFCs and Oracle implementation documents. The SASL behaviour has been clarified by starting with capturing the informal description of the RFC document, via ASM ground model. After that, the potential ambiguous description is explicated depending on other document sources through ASM refinement. The content of this paper is covered in the context of Chapter 7.

3) **Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon. "Analysing security protocols Using scenario based simulation". In Proceedings of 13^{th} International Conference on Verification and Evaluation of Computer and Communication Systems (VECoS 2019), Springer, 2019**.

   This paper presents a method for analysing security protocols. Typically, the idea presented in this paper was to extend the method of security protocols simulation based on attack scenarios in three ways. First, further reducing the number of protocol's runs by minimizing the number of intruder's generated messages; this achieved by limiting the intruder's ability to generate messages through considering: the expected message content and type matching. Second, specifying the attack scenarios in AsmetaL and simulating them using AsmetaS tool. Third, analysing protocols in the presence of the commutative algebraic property. The content of this paper is covered in Chapter 6.

1.5 Thesis Outline

The rest of this thesis is organized as follows:

Chapter 2: This chapter introduces background knowledge on the ASM method, starting by defining the main terminology used in this method, then reviewing various tools used for executing and analysing the ASM specifications.

Chapter 3: This chapter presents an overview of the common and well-known techniques that are used for safety and security analysis. Each technique is reviewed according to the following facets: analysis process, application areas, timing use, distinct features, and shortcomings. A comparison between these techniques is also illustrated in this chapter.

Chapter 4: This chapter provides background information on security protocols area, where basic notations, several external attacks, and flaws on such protocols are detailed. Besides, this chapter reviews the literature on formal methods for analysing security protocols. The reviewed methods include those applying logical inference rules, model-checking, term-rewriting rules, and simulation, to detect attacks upon protocols.

Chapter 5: This chapter details the methodology for analysing and developing safe designs for safety-critical systems by combining the ASM method with the STPA technique. This chapter also manifests how the informal STPA requirements are translated into the formal ones to be used in further verification. Furthermore, in this chapter, an illustration of applying this methodology to two case studies: the train door controller and the insulin pump control system, is shown.

Chapter 6: This chapter details a methodology for analysing security protocols. This methodology is based on developing queries that enable us to identify the possible flaws and the expected attack types; which are, in turn, used to determine the attack scenario that must be simulated to discover attacks upon the protocol under analysis.

Chapter 7: This chapter shows how can the ASM method be employed to clarify the informal requirements for simple authentication and security layer case study that is presented in the RFC document. This depends on using two ASM notions: a ground model and stepwise refinement, to build clarified specification for the behaviour of this case study.

Chapter 8: This chapter concludes this thesis by summarising the main findings regarding the research question and the associated subsidiary questions. It also discusses some suggested directions for future work.

Appendix A: This appendix illustrates the processes of the safety and security analysis techniques by applying them to an illustrative example.

Appendix B: This appendix states the functional requirements that are specified by validating scenarios used to analyse the insulin pump control system.

Appendix C: This appendix shows the ground model-control state for performing the rest steps in the authentication negotiation phase for simple authentication and security layer example.

Chapter 2

Abstract State Machines Method

2.1 Introduction

Formal methods play an effective role in the development process of computing systems. The effectiveness of formal methods comes from their mathematical basis, which is required for constituting well-formed notations for system specifications, and for allowing formal analysis. One of these methods is Abstract State Machines (ASMs) method [64].

This chapter provides an overview of the ASM method, starting with its theoretical background and moving down to its formal definition. In addition, a variety of tools for executing and/or analysing ASM specifications are reviewed and evaluated. This chapter also explains why ASM method has been used in this work for specifying and analysing computing systems.

2.2 Abstract State Machines Theory

Abstract State Machines (ASMs) were initially introduced by Yuri Gurevich [114, 115] as a general state machine formalism for modelling any algorithm at a convenient level of abstraction. He formulated the basic theory of ASMs as "every algorithm, no matter how abstract, is step-for-step emulated by an appropriate ASM". The generality of ASMs originates from the notion of abstract states. In traditional state machines, like Turing machines and finite state machines, states are represented symbolically by a collection of symbols. In contrast, abstract states are represented syntactically and semantically by mathematical structures of elements from domains equipped with functions and predicates

defined on them. In addition, ASMs have transition relations defined by *rules*, which specify how function interpretations are updated from one state into another. ASM specifications describe how the state of the system under specification evolves depending on transition rules. ASMs are further developed into a practical and mathematically well-grounded method for high-level system design and analysis [64, 60]. This method bridges the gaps between human understanding, formalization, and executable machines of a real world problem. ASMs improves the system development process by building an accurate high level modelling paradigm which is oriented toward executable code. ASMs have a number of successful applications in diverse areas, such as: specifying sequential, parallel, and distributed systems [64] [Chap. 4, Chap. 5, Chap. 6], modelling dynamic databases [106], defining the specification for programming and design languages like Prolog, Java, and UML [62, 38, 63, 70, 82], proving compiler correctness [244], specifying and analysing security protocols [45, 46, 240], modelling and verifying safety-critical systems [33, 30, 35], and so on.
The ASM method is constructed from three essential concepts:

- **basic abstract state machines** are transition systems which are based on abstract states, to model the systems structure, and on transition rules, to model the systems dynamic behavior;

- **ground model** is a technique for capturing system requirements through a concise and precise conceptual model;

- **stepwise refinement** is a general concept for constituting a "hierarchy of refined model" through a chain of steps starting from the abstract ground model and leading to a more and more detailed model which is implementation-linked, depending on the design decisions.

2.3 Basic Abstract State Machines

The notion of ASMs, or *basic* ASMs, were originally defined to capture the case where a single agent can execute simultaneous parallel actions. Later, this notion is generalized to distributed multi-agents acting and interacting in a synchronous or asynchronous paradigm, see Section 2.3.2.7.

Formally, basic ASMs are finite sets of *transition rules* with the form:

If *Condition* **then** *Updates*

which is the basic form for specifying the transition between abstract states. The *Condition* (or *guard*) is a formula in first-order predicate logic with no free variables, whose interpretation can be evaluated to *true* or *false*. *Updates* denotes a finite set of update functions of the form

$$f(t_1, \ ... \ , t_n) := t$$

where f is an n-ary function name, $t_1, \ ... \ , t_n$ are function arguments which are first-order terms, and t is the updated function value. In other words, the transition from one state into another one takes place when a finite set of functions modify their values. A basic ASM consists of four components:

- **Signature** (Σ): the required declaration of functions and domains.
- **Initial states** ($\mathcal{I}$): a set of states defined by specified constraints imposed on the signature.
- **Transition rules** ($\mathcal{TR}$): a set of transition relations identifies update sets over abstract states.
- **Main rule** or **program** ($\mathcal{R}$): a main transition rule of the machine with zero arity.

2.3.1 Simple Example

This section presents a very simple example, known as *Train Door Controller* (TDC) [227], with a view to provide the reader with a basic familiarity of ASMs formalism.

An automated TDC System receives sensitive data and transmits commands controlling the physical door of the train. Typically, the TDC system consists of the following components: *a*) computerized controller; *b*) door sensor; *c*) train sensor; *d*) emergency sensor; *e*) actuator; and *f*) door. The operation of the TDC system is best understood in terms of its components. The computerized controller manipulating the inputs coming from the sensor components to issue *open* or *close* the door command. The door sensor sends information about the door position and the obstacle existence. The train sensor generates signals showing the train motion and it's position according to the platform.

The emergency sensor produces an emergency signal when there is an emergency condition, such as fire or toxic gas. The actuator executes the issued control command on the door. The description of the requirements is that: the door should not be closed on a person in the doorway, the door should not be opened while the train is moving or when it is not aligned with a platform, and the travellers should be able to exist during the emergency.

2.3.2 Terminology Used in Abstract State Machines

This section provides some abstract state machine terminology used to describe the machine's execution.

2.3.2.1 Domain

Domain (also called *universe*) is a finite or infinite set of elements employed for the machine. The superdomain or superuniverse $\mathfrak{D}$ of an ASM state $\mathfrak{A}$ is divided into smaller domains related to specific category.

2.3.2.2 Function

A *function* in ASMs is defined by its name and arguments through the following form:

$$f(t_1, ..., t_n)$$

where f is a function name, and $t_1, ..., t_n$ are n arguments of a function; n is called an *arity* (a number of function's arguments). Zero arity functions, or *nullary* functions, are called *constants*. A function name is firmed in the signature. A function f is updated to a new value v in the next state, if in the current state the arguments t_i of f are evaluated to their values, say v_i, and a $f(v_1, ..., v_n)$ is evaluated to v value (the value of the function). A pair of function name and its associated parameter values constitute a *location*. A location represents a memory unit. The value of the function with its indicated argument values is the value of that location.

Functions can be pragmatically classified into: *derived* functions and *basic* functions.

Derived functions are auxiliary functions which come with a specification or computation scheme to yields values for the given read-only arguments. The *basic* functions are

differentiated from the derived functions in that they do not come with certain specification to update their values, instead, they are updated either by the machine (its rules), the environment (user), or both of them. Basic functions can be further classified, based on the updating way of their values, into: *static* functions and *dynamic* functions.

Static functions are functions with values that never change at any machine execution, i.e. their values are constant.

Dynamic functions are functions with changeable values from one state into another, so that they are similar to variables in programming. They are also categorized into three types: *controlled*, *monitored*, and *shared*.

Controlled functions are dynamic functions whose values can be read and modified only by the machine.

Monitored functions have values that are read by the machine, but written by the external environment (user). Whereas the dynamic *shared* functions are read and updated by both the rules and the environment of the machine.

2.3.2.3 Signature

A *Signature* Σ, (also called *vocabulary*), is a finite set of declarations of function names, and domain names. Each function name has a fixed arity. The zero arity functions, including: *true, false* and *undef* (undefined) are always assumed in the signature.

The signatures are declared when defining an ASM. The signature for TDC example is listed in Figure 2.1. In Figure 2.1, the domains *Status*[1], *DoorStatus*, *PositionStatus*, *MotionStatus*, and *Availability* are introduced to represent the operation states of the controller (sensing or executing), the set of potential door's states (opened, closed, opening, or closing), the set of possible train's positions (aligned and not aligned with a platform), the nature of train's motion (moving or stopped), and the availability status (exist or not) for the emergency and the obstacle, respectively. The functions *trainMotionSensor*, *trainPostionSensor*, *emergencySensor*, *obstacleSensor* denote monitored predicate, which represent an external environment input for train's motion and position, and existence of emergency and door's obstacles. The dynamic functions *doorStatus*, *trainMotion*, *trainPosition*, *emergency*, *obstacleStatus*, and *state*, show the current state for the door, the train's motion, the train's position, the emergency, and the obstacle at the doorway, and the op-

[1]Note that, we declare the *Status* domain as an abstracted domain, instead of enumerated one, to show how to deal with an abstracted domain

eration mode, respectively. The 0-ary functions, which comprises *sensing* and *executing*, refer to the atomic objects belonging to the *Status*. The *safeSituation* is a derived function that checks that the door is opened during the emergency situation. The *safeSituation* is defined such that it is *true*, if the door is opened or it is at an opening position.

EXAMPLE (Signature):
Domains
Status
DoorStatus={OPENED, CLOSED, OPENING, CLOSING}
PositionStatus={NOT_ALIGNED, ALIGNED}
MotionStatus={STOPPED, MOVING}
Availability={EXIST, NOT_EXIST}
Monitored Functions
trainMotionSensor: *MotionStatus*
trainPositionSensor: *PositionStatus*
emergencySensor: *Availability*
obstacleSensor: *Availability*
Controlled Functions
doorStatus: *DoorStatus*
trainMotion: *MotionStatus*
trainPosition: *PositionStatus*
emergency: *Availability*
obstacleStatus: *Availability*
state: *Status*
Static Functions
sensing: *Status*
executing: *Status*
Derived Functions
safeSituation: *Boolean*
where
safeSituation $\Longleftrightarrow$
$\exists s \in$ {OPENED, OPENING}: *doorStatus*= s

Figure 2.1: The train door controller example's signature

2.3.2.4 State and Update Set

A *state* $\mathfrak{A}$ for Σ is a mathematical structure consisting of the *superdomain* $\mathfrak{D}$, a non-empty set, together with an *interpretation* $f^{\mathfrak{A}}$, evaluation of terms and formulae, for every function name of Σ. When f has n arities, then it is interpreted as a function from $\mathfrak{D}^n$ into $\mathfrak{D}$.

An *update* of $\mathfrak{A}$ is expressed in a form

$$(loc, v)$$

where the *loc* is a location of $\mathfrak{A}$, and v is an element from the superdomain $\mathfrak{D}$. A set of updates of that form is called *update set* U.

An update set is called inconsistent if it has clashing updates that assign two different values to one location i.e (loc, v_1) and $(loc, v_2) \in U$, but $v_1 \neq v_2$; otherwise it is called *consistent* update set.

2.3.2.5 Transition Rules

A *rule* in ASMs specifies an update set over transition states, to describe the system's behaviour. It can be one of the following basic rules, or one complex rule constructed from more than one basic rule. Basic transition rules $\mathcal{TR}$ are as follows:

- *Skip rule*: **skip**.
  It causes an empty update set.

- *Update rule*: $\boldsymbol{f(t_1, ..., t_n) := t}$.
  It updates the value of $\boldsymbol{f(t_1, ..., t_n)}$ to $\boldsymbol{t}$. Where $\boldsymbol{t_1, ..., t_n, t}$ are first-order terms.

- *Block rule*: X **par** Y.
  It evaluates X and Y rules simultaneously and produces unified update sets computed by X and Y. The X and Y rules are syntactic expression.

- *Sequence rule*: X **seq** Y.
  It executes the rules X and Y in a sequential way, starting with X.

- *Conditional rule*: **If** $\boldsymbol{\psi}$ **then** X **else** Y.
  It checks if $\boldsymbol{\psi}$, a boolean expression, is true then it executes the X rule, otherwise executes the Y rule.

- *Let rule*: **let** $\boldsymbol{a = t}$ **in** X.
  It allocates the $\boldsymbol{t}$ value to $\boldsymbol{a}$, a logical variable, and executes X. The resulting update set is the set computed by X.

- *Forall rule*: **forall** $\boldsymbol{a}$ **with** $\boldsymbol{\psi}$ **do** X.
  It executes the X rule in parallel for every $\boldsymbol{a}$ meeting $\boldsymbol{\psi}$. The computed update set from this rule is the union of all the update sets yielded by the parallel execution of X over various values of $\boldsymbol{a}$.

- *Choose rule*: **choose** $\boldsymbol{a}$ **with** $\boldsymbol{\psi}$ **do** X **ifnone** Y.
  It non-deterministically chooses $\boldsymbol{a}$ satisfying a given condition $\boldsymbol{\psi}$ and executes the X rule. In case that no such $\boldsymbol{a}$ exists, it executes the Y rule.

- *Call rule*: $\boldsymbol{r(t_1, ..., t_n)}$.
  It executes the formerly specified transition rule $\boldsymbol{r}$ with the given $\boldsymbol{t_1, ..., t_n}$ arguments.

The *main rule*, or *program*, of the machine is "a distinguished rule name of arity zero", that describes single step of the machine. It is executed repeatedly. A main rule may either successfully terminate if no new update set is produced or no rule is viable, or it may fail to terminate when inconsistent update set is produced.

To illustrate, the TDC example has main rule and eight textual complex rules. Four of the complex rules, including *ClosedToOpening*, *OpeningToOpened*, *OpenedToClosing*, and *ClosingToClosedOrOpened*, are used to update the *doorState*, which is initially CLOSED. Three of the complex rules, which are *EmergencySensing*, *TrainMotionSensing*, and *Obstacle Sensing*, are specified to sense the sensors' information. The main rule, named *TrainDoorController*, is defined to model the behavior of the system.

In the main rule for TDC example, see Figure 2.2, if the current state of the system is SENSING, then three rules are called for sensing the current state of the emergency, train motion, train position, and the door's obstacle. While if the current system's state in EXECUTING, the emergency is checked. If it exists, the emergency is handled by opening the door; otherwise, four rules are called in parallel, one per door state changing. Note that, the **par** construct is rarely written, as the synchronous parallelism is the execution mechanism of ASMs by default [60].

In the *ClosedToOpening*, see Figure 2.3, the controller checks whether the state of the door is CLOSED and the train is STOPPED such that it is ALIGNED with the platform, to update the door state to OPENING. While in the *OpeningToOpened* rule, see Figure 2.4, the controller updates the state of the door into OPENED, if it is at OPENING state. The *OpenedToClosing*, and *ClosingToClosedOrOpended* rules have the same changing the door state activity, but they check the input from the obstacle sensor before taking a decision

about changing the door state.

```
TrainDoorController =
    if state = SENSING then
      EmergencySensing
      TrainMotionSensing
      ObstacleSensing
      state := EXECUTING
      emergency := NOT_EXIST
    else if emergency=EXIST then
              HandleEmergency
          else
              ClosedToOpening
              OpeningToOpened
              OpenedToClosing
              ClosingToClosedOrOpened
              state:= SENSING
    where
      EmeregencySensing =
          emergency:= emeregencySensor
      HandleEmergency =
          if not safeSituation then
             doorStatus = OPENED
```

Figure 2.2: The main rule for Train Door Controller example

```
ClosedToOpening =
     if doorStatus = CLOSED and trainMotion = STOPPED and
       trainPosition = ALIGNED then
          doorStatus := OPENING
```

Figure 2.3: The ClosedToOpening rule for Train Door Controller example

```
OpeningToOpened =
     if doorStatus = OPENING then
        doorStatus = OPENED
```

Figure 2.4: The OpeningToOpened rule for Train Door Controller example

2.3.2.6 Step and Run

A computation *step* for an ASM is the process of simultaneous executing, firing, all updates for all transition rules in a given state to yield the next state.

A *run* is a finite or infinite sequence of consecutive states, resulting from executed *steps*.

2.3.2.7 Distributed Abstract State Machines

Distributed Abstract State Machines (DASM) are extended basic ASMs that cover the formalization of multi-agents acting and reacting in a synchronous or asynchronous manner [64].

DASM has a finite set (possible dynamic) of agents *Agents* where each agent executes its own basic ASMs. DASM introduces two notions: *local state*, a $\mathfrak{A}$ state for an agent's machine, and *global state*, a $\mathfrak{B}$ state for DASM. Using these states is supported by the **self** function, which is a *reserved 0-ary function* of type *Agents* for denoting the agent which is running its basic machine. The relation between these states, which is required to determine the run of DASM, relies on the agents nature: synchronous or asynchronous.

In *synchronous* DASM, every agent operates in parallel and synchronously by running their own machine using an implicit global system clock, and contributes to the global states over the union of the signatures of each machine. The runs for the synchronous machine is a totally ordered set of steps.

Asynchronous DASM provide a coherent global system view for concurrent sequential computations of single agents, where each individual agent has it's own pace without any global clock to execute its basic ASM in its local state. In DASM, a single computation step achieved by an agent is replaced by the notion *move*, which can be atomic or durative. A run for DASM is a partial order set of moves for *Agents*. It is formally defined as a triple $(\mathcal{M}, \phi, \gamma)$, such that the following conditions are attained [114]:

1. $\mathcal{M}$ is a partially ordered set of moves for *Agents*, such that each move has only finitely many predecessors;

2. ϕ is a function on $\mathcal{M}$ for associating agents with the set of moves in such a way that the set of moves computed by every agent is linearly ordered;

3. γ is a function for assigning a state of $\mathcal{M}$ to each initial segment $\mathcal{X}$ of *Agents* by performing all moves in $\mathcal{X}$;

4. If m is a maximal element belonging to a finite initial segment $\mathcal{X}$ of $\mathcal{M}$ and $\mathcal{Y} = \mathcal{X} - m$, then $Agents(m)$ is an agent in $\gamma(\mathcal{Y})$, and $\gamma(\mathcal{X})$ is gained from $\gamma(\mathcal{Y})$ by firing $\phi(m)$ at $\gamma(\mathcal{Y})$.

2.3.3 Ground Model and Stepwise Refinement

Ground model is a conceptual model that represent an accurate formulation (also called *system blueprint* or *system contract*) of a real-world problem for a system to capture it's informal requirements in a transparent way. The ground model has no general mathematical definition, but it has the following intrinsic features:

- *precision*: it means the ground model must be precise at a convenient level of detailing that satisfies the desired accuracy literally, without adding inessential exactness;
- *simplicity* and *conciseness*: constructing a simple and concise ground model needs to avoid any extraneous encoding and reflect just the structure of the system to provide an understandable model and manageable for the analysis;
- *abstractness*: presenting every semantically relevant features and abstracting from irrelevant ones which are required for later refinements;
- *consistency*: eliminating misleading details and ambiguities in the textual requirements;
- *semantical foundation*: the ground model is equipped with a precise semantical foundation which is the basis for the model analysis process.

The ground model can be constructed graphically using *control state ASMs*. Control state ASMs are a class of ASMs representing a normal form for UML activity diagrams, and enabling the designer to model machines which under the main control structure of Finite State Machines (FSM). Control state ASMs enrich the FSM control structure by synchronous parallelism and/or manipulating data structures. A control state ASM can be used at an early stage of the system development process, to capture the system's requirements due to its graphical form which is associated with a precise semantics.

A control state ASM is an ASM whose rules are all of the form textually and graphically presented in Figure 2.5, such that in a given control state i, the machine remains at it if no condition $condition_j$ is satisfied.

Typically, in a control state ASM, each rule can be defined as a generalized FSM with the following representation:

FSM$(i,$ **if** *condition* **then** $rule,\ j) \equiv$

if $ctl\_state = i$ **and** *condition* **then**

$rule$

$ctl\_state := j$

where each rule updates the control state *ctl_state* value, from i for example to the next value j relying on the guard *condition*.

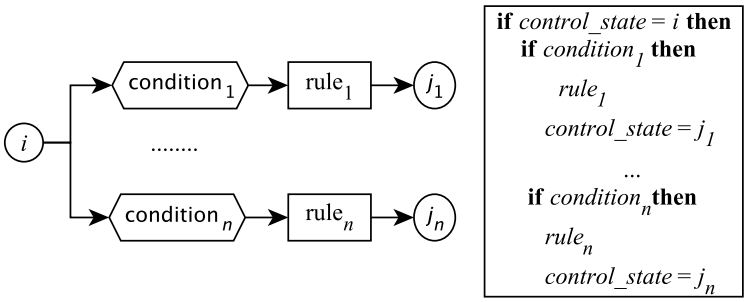

Figure 2.5: Control state ASMs

Stepwise Refinement is a successive refinement method that enables to obtain a more detailed model from the abstract one. This can be achieved by either refining the signature, the flow of operations, or both of them. At each refinement step, an obtained model must be proved correct with respect to former upper one, i.e, the more concrete model implements the exact functionality of abstract model plus its one, while keeping the main features of the system, e.g., safety. The first abstract model (the ground mode) is close to the system's description view (often informal), while the lowest refined model is close to the programmer's view (executable code). Such stepwise refinement facilitates changing the system design in an incremental way, and enables to find silent assumptions and ambiguities in the system's requirements while crossing abstraction levels.

2.4 The Abstract State Machines Tools

For the last two decades, different tools have been developed around ASMs to offer executable models and to support one or more of the following analysis tasks: simulation, validation, and verification. Among all the available tools, this work has chosen the ASMETA framework. As a result, this section surveys a number of the more popular ASM tools and specification languages, and compares their distinctive attributes and potential shortcomings, see Table 2.2. Table 2.2 summarizes the main features for each tool, hence, we can conclude from this table that the ASMETA framework is the most convenient tool for this work, as it is available, it uses notation close to the ASM formalism, and it supports a variety of analysis tools that serves the aim of this work.

2.4.1 The Dynamic Algebra Specification Language

The Dynamic Algebra Specification Language (DASL) is a formal language for an earliest ASM simulator [135]. DASL is inspired by an approach for a Dynamic Algebra (DA) Specification of the full Prolog language [58, 57]. Typically, within a Prolog framework, a concrete language for DA specifications was defined, and a prototype compiler was designed [135].

The DASL is a typed language that extends the ASMs by a specific form for a set of multi-sorted equational specifications which defines the static part of the specifications. Generally, a signature in DASL can be declared according to two types: *dynamic* and *static*. The dynamic or evolving part is a finite part that includes the domains and functions declaration. The static part is infinite part consisting of a sequence of constructors and a set of conditional and unconditional equations defined on the declared functions. In DASL, updating states occurrs regularly at each step of the simulation depending on the evaluated conditional rules. The initial state for DASL is specified explicitly using the *start* rule. While final states are defined implicitly by transitions originating from the initial state. However, in DASL a distinguish between regular and error final states must be defined to characterize the termination situation. Where, the final states imply no further updates of algebra are permitted.

The DASL specification is associated with a compiler for compiling its specification into intermediate program, known as an Algebraic Target Machine (ALMA). ALMA is a single-sorted abstract state machine adapted for DA computations. It presents three types

of control statements: a simple and two conditional statements (*if*- and *case*- statements). An ALMA program is obtained by a decision tree. This tree has *if*- or *case*- nodes, with either *update*- or *error*- leaves. A computation for the ALMA is the execution of a simple statement (initialization) followed by a recursive walk over the decision tree. Both of the ALMA and the compiler are implemented in the Prolog language.

Although the DASL has a nice features, like simplicity and executability, it is tailored only for a small subset of sequential deterministic ASMs, with slightly different syntax and semantics from the actual ones of ASMs.

2.4.2 The ASM Workbench

The ASM Workbench (ASM-WB) is a development framework of modelling and analysis tools for ASMs [86, 87]. It aims to overcome the main shortcomings of the existing ASM tools at that time, including incompatibility between the available ASM tools, and lack of extensibility with other analysis tools. It was designed to constitute a basis for the development of further ASM tools and transformations to other modelling languages.

The ASM Workbench provides an executable specification, which is the ASM-based Specification Language (ASM-SL). The ASM-SL notation extends the basic language of ASMs [114] by introducing several structural and pragmatical additions. In particular, defining a simple and pliable type system, identifying the states' structure based on transition rules, providing mechanisms to construct interfaces to the system's environment.

The type system of the ASM-SL adopts the same one of the Standard ML [171]. This type system may allow for detecting trivial errors and inconsistencies at a very early development stage, before conducting any simulation or verification. The data structures and notation include a set of predefined types (booleans, integers, floating-point numbers, and strings), data structural constructors (tuples, lists, finite sets, finite maps), and definitions for recursive and mutually recursive functions.

The ASM Workbench architecture consists of a *kernel* and a set of *exchange formats* that allows the Workbench to be extensible. The kernel is a set of program modules written in the functional language Standard ML [171], which is responsible of the main functionalities, such as parsing, type-checking, and interpreting-based evaluation. A kernel is formed of a collection of data structures representing syntactic (e.g., transition rules and terms) and semantic (e.g., update sets and runs) objects of ASM specifications, and a collection of functions to treat ASM objects. According to the exchange format feature,

any tool can be added to the ASM Workbench, without caring about the internal work of the ASM Workbench, through either the *tight coupling*, writing other ML programs that employ the kernel's functionalities, or through the *loose coupling*, writing other programs in any language that compatibles with the ASM Workbench kernel.

The key tools that have been involved with the ASM Workbench are a type checker, an interpreter, and a graphical user interface for controlling the simulation results and debugging. The planned tools were a code generator for the Java Virtual Machine, and an interface to the SMV model checker [167] relying on the translation of the ASM-SL specification into the SMV language.

The development for the ASM Workbench ceased in 2001. The ASM Workbench is not available on the Web [2], at the present time.

2.4.3 The ASM Gofer

The AsmGofer is a programming system based on abstract state machines [208], aiming at providing an ASM interpreter, which is embedded in the functional programming language Gofer [235] (more precisely TkGofer [233]), a dialect of Haskell language [228]. The AsmGofer is compatible with the Unix, MS, and Windows 95/NT operating systems. This system has been used in several substantial applications. For example, *Java and the Java Virtual Machine* [223], *the Light Control Case Study* [61], and *Simulating UML Statecharts* [71].

An AsmGofer program, or *script*, is a combination of signatures, rules, functions and data structures which are given in any order. AsmGofer is strongly typed but its signatures are non compulsory nevertheless.

The AsmGofer conservatively extends Gofer by introducing the concepts of state and parallel updates. Supporting these concepts is performed by allowing the AsmGofer to modify the evaluation machine in the run-time system for Gofer, and to use the *IO* actions [191] for achieving the input-output-operations. By this way, encoding ASMs in Gofer is enabled while maintaining the ASMs feature of having side effects on the global state. Typically, AsmGofer does not provide any ASM syntax. Instead, it retains the Gofer syntax and represents the ASM features as expressions. As a consequence, the AsmGofer language is Gofer-based, but it is not ASM-based.

AsmGofer defines constructs for sequential and distributed ASMs. For sequential ASMs, the *seq* construct is used for executing a set of rules sequentially in one step.

For distributed ASMs, the special function *multi* is employed for firing rules of multiple agents. This function takes a bounded set of agents as an input, and it is implemented by non-deterministically choosing a subset of those agents to execute their rules simultaneously at every step. The *multi* function does not choose a subset that leads to an inconsistent update.

The AsmGofer interpreter underpins a command line interface. With this interface, the user can issue several commands, such as evaluating an expression, printing the type of an expression, loading and editing files containing an AsmGofer script, etc.

A practical feature of AsmGofer is an automatic generation of a graphical user interface (GUI) which is fairly useful for validating and debugging the specifications. The GUI generator, which is written in AsmGofer, reads information from a special configuration file, such as information about expressions to display, and rules that can be selected in the GUI to execute. By this information, the GUI generator shows the evaluation of AsmGofer program at each step.

The AsmGofer is available for academic use on the system website [13].

2.4.4 Extensible Abstract State Machines

One of the noteable implementation for ASMs is the Extensible Abstract State Machines (XASM) project [28]. This project considers the sequential abstract state machines to generate efficient executable programs and simulate the run of the specified machines. Generally, the main design goals are upgrading the ASMs formalization into a programming language, and managing large ASM specifications by the added features, including modularity and reusabilty. The XASM project comprises the executable XASM-language, XASM-compiler, the runtime system, and the graphical debugging and animation interface.

The XASM-language underpins the concept of the constructs modularization, which is based on the component notion (presented in [225]), for structuring and reusing the ASM specifications. A modularization construct or a component in XASM machine can be reused either as a *sub-asm*, a rule called in its parent-asm to compute a step of the parent-asm, or as a *function*, a term or normal machine in the body of the parent-asm used to compute its internal steps. Each construct includes a list of declared functions that is enriched with *accesses* and *updates* functionalities for reading and writing, respectively, the locations of the included functions.

The XASM is a typed language. It supports the regular expression based pattern matching on strings, to match the left operand (string data) against the right operand (pattern matching variables). In addition, this language supports the grammar definitions to generate a parser of the specified syntax and semantics for this language.

The XASM project is implemented in C language. At this basis, the XASM source files are translated by the XASM-compiler into C source code to implement the ASM version, which is specified in XASM files. To this end, the runtime system fulfills the update and the access functionality for the ASMs functions. The graphical animation and debugging tools have been employed for tracing the updates and viewing the function values at each step.

The XASM project defines an external language interface for interconnecting the XASM programs with C programs. The interaction is realized in two alternative ways. First, specifying external C functions in XASM specifications. In this case, the arguments and the returned values of C functions must be characterized in a specific C-type which are represented elements of the XASM super-domain. Second, the main XASM machine and all its sub-machines can be embedded in the main C-code, when they are compiled properly using the XASM-compiler. An interface to the Java language has been considered as a future work.

At the present time, the XASM is not available on the Web [6].

2.4.5 Abstract State Machine Language

In 2000, the Foundations of Software Engineering (FSE) team at Microsoft Research developed an executable specification language called Abstract State Machine Language (AsmL) [40, 117, 4]. AsmL is fully integrated into the Microsoft .Net framework. It is inspired by the ASM theory but it supports some object-oriented and programming features required for the .Net integration. Typically, the AsmL combines the intrinsic features of ASMs, including synchronous parallelism and finite choice, with the programming language merits, such as interfaces, classes, methods, and exception handling. AsmL provides mathematical types and declaration for sets, bags, tuples, maps and sequences. It also upholds some mathematical set operations which are necessary for editing high-level specifications, such as comprehension and quantification. The AsmL strongly enforces the data typing at a compile time. The essential characteristic of the AsmL is being executable specification which can be used at the design and test development stages.

According to the implementation side, the Microsoft's AsmL tool originally includes a compiler for compiling the AsmL specification into executable .Net files. Subsequently, it has been added a plug-in for the Microsoft Word that facilitates the literate specification, i.e., the AsmL specification can be encoded into Microsoft Word file, using a specific formatting style, and then the resulted file is compiled by the AsmL compiler. The integration with the Microsoft environment and supporting some programming features cause the AsmL specification to be more closer to the detailed code universe than to the concise problem description universe. Furthermore, it can not specify the distributed systems.

From the analysis side, there is an approach for model checking AsmL specification, without encoding the AsmL into another notation accepted by the existing model checker tools [136]. This approach employs the on-the-fly verifying to achieve a direct exploration for the state space of a system specification. However, the performance for this approach is considerably slower than the other explicit model checking tools, like Spin and NuSMV, as it runs in the .Net platform [136]. The efficiency for this approach has not been shown for a complex case study [136].

In addition to the verification activity, the AsmL is integrated with a model-based testing tool, known as Spec Explorer, within the Microsoft .NET architecture [230]. The integration with this tool provides an environment for developing models through generating test cases from models, and executing them against an implementation under test. The model can be written either in MS Word that is embedded in Spec Explorer, or in Spec# language. The Spec# is an extended version of C# through presenting most of AsmL features with C# syntax. The new version of Spec Explorer: Spec Explorer 10 Visual Studio, does not directly support the implementation of ASM parallel update semantics [118]. Furthermore, the official language for Spec Explorer is Spec# which is slightly different from the AsmL.

2.4.6 The Timed ASM Language and Toolset

The Timed Abstract State Machine (TASM) language [184] and its associated toolset [183] are developed for engineering the reactive real-time systems. The TASM language extends the theory of ASMs by integrating the functional and non-functional specifications, in particular: function, time, resource consumption [184, 181]. In other words, the TASM language differs from ASMs in two syntactical aspects to capture the physical behaviour of the real-time systems. First, the machine steps are instantaneous in ASMs, while they are

durative in TASM. Second, in TASM, the durative steps can consume a finite quantity of resource, such as power, memory, and communication bandwidth. The syntactical sugar addition of TASM language considers the specification of the sequential and concurrent behaviours. However, it seems to be that the TASM language deals with time in some limited form, where the continuous dynamic behaviour is not introduced. Furthermore, the syntax of the TASM language is presented in a concise way, which does not has the definition of arrays and data structures.

TASM language also providess the *hierarchical composition* concept, which is based on the component concept for XASM language [28], for structuring large specifications.

The TASM toolset has three essential components for implementing the features of the TASM language, including an editor, an analyzer, and a simulator. The editor enables the TASM specification to be created and edited using formal expressed syntax and semantics of the TASM language [181]. The simulator allows the dynamic behaviour of the specifications to be visualized graphically through a step-by-step fashion. The simulator's application results are illustrated in [180]. The analyzer component achieves the specifications analysis task.

For the formal analysis of the TASM specification, two tools, including SAT solver [172] and UPPAAL [43, 143], have been used to perform functional and static types of analysis, respectively.

During the functional analysis, an automatic verification of completeness and consistency properties of TASM specifications is performed [182]. In the TASM context, for a given machine, the completeness of the specification means that there is an enabled rule for every possible combination of it's monitored variables. While the consistency means that there are no two enabled rules at the same time with two different values. The analysis of these properties is achieved by translating the TASM specifications into the input specifications for SAT solver and by formulating both completeness and consistency properties as a Boolean satisfiability problem [217]. The TASM toolset employs the SAT4J solver, which is an open source SAT solver [48].

Whereas for the static analysis, the timing properties [142] of TASM specifications are verified using UPPAAL tool [179]. The TASM specifications are mapped into the timed automata of UPPAAL, and the timing properties, in terms of safety and liveness, are formulated in the temporal logic of UPPAAL.

The main limitations with the TASM toolset are as follows. The analysis toolset is inapplicable to large case studies. The translation to the input specification of UPPAAL

and SAT solvers supports only finite TASM model with allowed datatypes, e.g., decimal numerical values from the Reals domain are not acceptable. Furthermore, there is no a generic translation way to be adopted if a new tool is desired to be added.

The TASM toolset is fully written in Java programming language, using a number of the freely available Eclipse libraries [1]. Unfortunately, this toolset is not available at the website [10], contrary to what is claimed in its paper [183].

2.4.7 Simulator for Real-Time ASMs

The Simulator for Real-Time ASMs is designed for simulating reactive timed abstract state machines [218]. The reactive ASMs mean that the ASMs should sufficiently respond to infinite input. In particular, ASMs have a special input that represents the time value, which is given by the external function CT (derived from current time as in [116]). The time for ASMs, that the simulator deals with, can be continuous or discrete. The time constraints, or the guards of the rules, are linear inequalities.

The simulator is equipped with a language to describe the timed semantics for ASMs. Essentially, for this simulator, two semantics have been defined: one with instantaneous actions, and the another one is with no-deterministic, bounded, and delayed actions. These semantics are characterized for timed basic ASMs with a syntax similar to AsmL [40]. The denoted syntax is based on the if-then updates, parallel, and sequential composition.

The simulator is applied for verifying whether the generated run from the reactive timed machine satisfies the requirements property for this run. The properties of real-time ASMs are expressed as First Order Timed Logic (FOTL) formulas [41], which is a predicate logic with arithmetics. It's main concept is to select a decidable theory, e.g., real addition, to manipulate arithmetics or other concrete mathematical functions, and then extend it by abstract functions of time, which are required to specify the considered problems.

Unfortunately, a complicated case study that entails multi-agent construction is inapplicable using the simulator, and some rule constructors, such as: *extend, import*, are not defined in the presented timed ASM's syntax [218].

The simulator is an independent tool implemented in Java programming language. It's architectural components include a **kernel** and a **graphical user interface**. The kernel is a java-application for accomplishing the necessary tasks of processing the ASM specifications and checking the FOTL properties. The results of the simulation, which are runs of the simulated ASM, are stored in trace repository. These results are accessible

by the user, the **verifier** component of the kernel, and the graphical user interface. The verifier verifies the timed ASM properties. The graphical user interface represents the results graphically. Currently, a version of the simulator is not available at the mentioned website [11] in [218].

2.4.8 Core Abstract State Machine

The Core Abstract State Machine (CoreASM) is an open development project that includes tools for modelling and simulating ASM specifications [99, 97, 100, 98]. It was designed with the aims of providing an executable language that supports rapid prototyping for ASM models, and developing an extensible tool architecture to accommodate different application domains.

The CoreASM project provides the following facilities [101]:

- A lean and executable specification language, called CoreASM.
- A multi-agent simulation engine for simulating the CoreASM specification language.
- A library of optional *plug-ins* that provides extra characteristics and syntactical sugar, which are not de facto part of the ASMs, for example *case*, and the *plug-ins* concept facilitates the extensibility aim and it is in charge of the most of the engine's functionalities.
- An Eclipse user interface with a command-line interaction plus dynamic syntax highlighting.

The CoreASM language has textual notation very close to the mathematical definition of ASMs. CoreASM language serves as an untyped modeling language, to enable rapid prototyping with ASMs. Unfortunately, this feature is at the price that unpredictable type errors will be caught only at the execution time, and handling the large erroneous specification will be slightly difficult. The design of sequential, parallel, and distributed systems is feasible through CoreASM language.

The structure of the specification language consists of a *header block* for declaring the signature depending on the used plug-ins, and *rule declaration block*, where the specifications of the rules take place, including the *init rule* for constructing the initial state. The engine's architecture is made up of four constituents: the *Parser*, the *Interpreter*, the

Abstract Storage, and the *Scheduler*. To execute the specification, the Parser firstly produces a parser for the specification, with a language grammar based on the used plug-ins in the header block. Then, the Interpreter evaluates all the specified rules and generates update sets. The Abstract Storage saves the current state of the machine and the history of its previous states as well. The Interpreter gets the value for the current state from the abstract storage and produces updates for the next state. The Scheduler is used for scheduling the agents.

There are several attempts for making CoreASM supports model-checking activity, such as: [102, 161, 42]. In [102], an approach for translating the CoreASM specifications into Promela specifications, which can later be verified by Spin model checker [123], is illustrated. This approach is extended in [161], to support the n-ary functions and the *extend* rule which can potentially lead into models with infinite state space. Contrary to these approaches, the method in [42] aims at keeping the expressiveness of the CoreASM language, by not performing any translation from CoreASM language into another input model checking language. This is achieved through employing the CoreASM simulator to generate the state space, which can then be model checked by the Model Checking Micro-Controller, or for short [MC] SQUARE model checker [207]. This method is nevertheless has limited capability and has been applied to only small case studies [94]. Despite all of these attempts, till now there is no an automatic tool integrated within CoreASM project to support them.

The CoreASM engine is implemented using the Java programming language. The CoreASM is an open source project available at [5]

2.4.9 The ASM mETAmodelling Framework

The ASM mETAmodelling Framework, ASMETA for short, is one of the most comprehensive modeling and analysis environments developed for ASMs [111, 36]. Realising the drawbacks of available tools around ASMs, such as limited coverage of the aspects in the whole development process, encoding ASMs into a language with syntax rigidly relying on the implementation environment, and the impracticality of the integration of the ASM tools, the ASMETA framework was born to provide a unified notation and interoperable tools environment for ASMs.

ASMETA is based on the guidelines of the Model-Driven Engineering (MDE) concepts and technologies [209], which facilitate the practical integration of a set of tools supporting

various tasks of the development process. The MDE technologies include:

- *domain-specific modelling languages* for describing the system's behaviour and design intent. The model-driven language consists of: an *abstract syntax* (*metamodel*) that abstractly represents the concepts and constructs of a specific domain (e.g., ASMs), a *concrete syntax*, which can be textual, visual, or both, and it is driven from the metamodel to be the language used by the designer, and *sematics* that find the meaning of the user language.

- *transformation engines and generators* for analysing specific aspects of the models and synthesizing different artifacts, like simulation , source code, or alternative model representations.

Regarding this technology, the ASMETA development defines a metamodel for ASMs, known as *Abstract State Machine Metamodel*(*AsmM*), using the OMG metamodelling framework [7], and it develops a set of software artefacts (concrete syntax, parser, API, etc.) which is used for model editing and it is also helped for developing new complex tools integrated within this framework.

ASMETA is implemented using the Java programming language. The ASMETA framework is available for academic use on the website [3].

As one of the analysis tools for the ASMETA framework is a model checker, we provided some background on model-checking and its input temporal logic, before introducing the ASMETA tools.

2.4.9.1 Model-Checking

Model-checking [79] is a common technique that emerged with the goal of automatic verification for the desired behavioural properties of computational systems. It entails a finite model of the system and a property of interest which is a formula expressed usually in temporal logic. The model-checking technique depends on an efficient algorithm to exhaustively traverse the state space of the system in order to determine whether the given model satisfies the desired property. When a model fails to satisfy a required property, the model checker generates a faulty trace called counter-example that manifests an incorrect system's behaviour. The model checker has become a prominent verification technique as it does not require the construction of a manual proof, and it produces diagnostic counterexamples. Several practical model checking techniques have been developed, such as

SPIN [123], UPPAL [143], NuSMV [75], etc. In our work for analysing safety-critical systems, we use the AsmetaSMV model checker that verifies temporal properties of ASM models based on the capabilities of the NuSMV model checker.

2.4.9.2 Temporal Logic

Temporal logic is an approach for describing the desired properties of a system behaviour as logical formulae based on the concept of time [104]. Different forms of temporal logic have been constructed depending on the adopted way of defining the time concept, such as Linear-time Temporal Logic (LTL) [193], Computation Tree Logic (CTL) [78], etc. In our work for analysing safety-critical systems, we consider the LTL form, which models the time as an infinite path of states.

The syntax and semantics of LTL are as follows:

Syntax. Any LTL formula is formally constructed from the following symbols:

- a set, PROP of propositional symbols or *literals*;
- the constants **true** and **false**;
- propositional connectives: $\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (implies), $\leftrightarrow$ (if and only if);
- temporal connectives: $\bigcirc$ (in the next moment in time), $\Box$ (always in the future), $\Diamond$ (eventually in the future), U (until), and W (weak until);
- two brackets symbols: '(' and ')'.

The set of well-formed LTL formulae (WFF), is inductively defined as the smallest set that satisfies the following two conditions:

- The **true**, **false**, and any element in PROP set are in WFF.
- If X and Y are in WFF, then $\{\neg X, X \vee Y, X \wedge Y, X \rightarrow Y, X \leftrightarrow Y, X \text{ U } Y, X \text{ W } Y, \bigcirc X, \Box X\}$ are in WFF too.

Semantic. The LTL models time in a manner similar to Natural Numbers, $\mathbb{N}$, in other words, a model of LTL, σ, can be described as a sequence of infinite states of the form $\sigma = \{s_0, s_1, s_2, ...\}$ where each state, s_i, is a set of proposition symbols. Those propositions are satisfied in the i^{th} moment in time. Owing to the fact that, any LTL formula is interpreted at a particular moment in time, the satisfaction of a formula

X is denoted by $(\sigma, i) \models X$, where σ denotes the model and i is a state index at which the temporal formula is interpreted. For any well-formed formula X, model σ, and state index $i \in \mathbb{N}$, then either $(\sigma, i) \models X$ holds, or not. Typically, the semantics for the symbols that construct the LTL formula are:

$(\sigma, i) \not\models \textbf{false}$
$(\sigma, i) \models prop$ iff $prop \in s_i$, where $prop \in$ PROP
$(\sigma, i) \models \neg X$ iff $(\sigma, i) \not\models X$
$(\sigma, i) \models X \vee Y$ iff $(\sigma, i) \models X$ or $(\sigma, i) \models Y$
$(\sigma, i) \models X \wedge Y$ iff $(\sigma, i) \models X$ and $(\sigma, i) \models Y$
$(\sigma, i) \models X \rightarrow Y$ iff $(\sigma, i) \models Y$ whenever $(\sigma, i) \models X$
$(\sigma, i) \models \bigcirc X$ iff $(\sigma, i + 1) \models X$
$(\sigma, i) \models \Box X$ iff $\forall j \in \mathbb{N}$ and $j \geqslant i$ then $(\sigma, i) \models X$
$(\sigma, i) \models \Diamond X$ iff $\exists i \geq 1$ such that $(\sigma, i) \models X$
$(\sigma, i) \models X$ U Y iff $\exists j \in \mathbb{N}$, such that $j \geqslant i$ and $(\sigma, j) \models Y$ and $\forall k \in \mathbb{N}$: if $i \leqslant k < j$ then $(\sigma, k) \models X$
$(\sigma, i) \models X$ W Y iff $(\sigma, i) \models X$ U Y or $(\sigma, i) \models \Box X$

2.4.9.3 ASMETA Framework Tools

The ASMETA framework comprises different tools, including but not limited to:

The ASMETA Language (AsmetaL) is a textual notation and concrete syntax for AsmM specified in terms of an EBNF (Extended Backus-Naur Form) grammar. It is strongly-typed language, encoded in a way very similar to the basic and multi-agents ASM [64]. AsmetaL is equipped with a *Standard Library*, a declaration set of predefined ASM domains (Integer, Boolean, String, etc.) and functions defined on those domains, and a *text-to-model compiler* (AsmetaLc) for parsing and translating the AsmetaL specifications into AsmM instances that can later be executed by the simulator.

The structure of AsmetaL consists of four sections: a *header* for importing the Standard Library and defining the signatures of domains and functions, *body* for inserting, in order, the implementation of static concrete domain, derived and static functions, and then the rules of the model, *main rule*, and *initialization* for initializing the controlled functions. For example, see the AsmetaL specification for the Train Door Controller example in Code 2.1.

The ASMETA Simulator (AsmetaS) is a tool for constructing a *run* (an update set at every step) of the specification under simulation [112]. In fact, the AsmetaS is an interpreter that makes computations for the stored ASMs specifications in the abstract storage or model repository. The AsmetaS enables specification validation process. Due to the direct integration of the AsmetaS in the ASMETA framework, AsmetaS simulates the model without the need for implementing a parser, a type checker, and an internal representation of the model to simulate. The AsmetaS supports several useful activities during the simulation, including *consistency checking* for updates set in order to detect inconsistent updates, *random simulation* for simulating the specification randomly, where the simulator chooses the input values by itself in an arbitrary manner, and *interactive simulation* for simulating arbitrary values provided interactively from the environment by the monitored functions, and *invariant checking* to inspect whether invariant expressed constrains over the functions and rules of the executed AsmetaL model are satisfied or not. The invariant constraint can be declared as follows:

$$\textbf{invariant}\ \mathit{constraint\_name}\ \textbf{over}\ \mathit{fun\_name}, ..., \mathit{rule\_name} : \mathit{term} \tag{2.1}$$

where **invariant** is a keyword related to a selected name *constraint_name* of the constraint, fun_name and *rule_name* are some functions and rules of the specification that are mentioned after the **over** keyword, and *term* denotes the boolean term that expresses the constraint.

The ASMETA Validator (AsmetaV) tool is a more effective and reliable validation approach for ASM models than the AsmetaS [69]. It does not depend on the user's judgment on the conformance of the actual outputs against the supposed ones. It is based on the *scenario construction* idea for validating the AsmetaL specifications. A *scenario* characterizes the system's behaviour globally through exploring the perceivable interactions between the system and its external environment in certain circumstances. A scenario is quite helpful for confirming that the informal requirements are captured correctly, and for indicating alternative design solutions while exploring system functionalities.

The scenario-based validation approach is inspired from the concept of UML use-cases description [56], to realize the dual aim of model validation and testing. In UML, a scenario is a diagrammatic description of the interaction sequences between actor actions and reactions of the analyzed system. In the AsmetaV context, a scenario is an algorithmic

```
asm TrainDoorController
import StandardLibrary
signature:
  // DOMAINS
  enum domain DoorStatus={OPENED | OPENING
                          | CLOSING | CLOSED}
  enum domain Availability={EXIST | NOTEXIST}
  enum domain PositionStatus = {NOTALIGNED
                                | ALIGNED}
  enum domain MotionStatus = {STOPPED | MOVING}
  enum domain Status = {SENSING | EXECUTING}
  // FUNCTIONS
  controlled state: Status
  controlled doorStatus: DoorStatus
  controlled trainMotion: MotionStatus
  controlled trainPosition: PositionStatus
  controlled emergency: Availability
  controlled obstacleStatus: Availability
  monitored trainMotionSensor: MotionStatus
  monitored trainPositionSensor: PositionStatus
  monitored obstacleSensor: Availability
  monitored emergencySensor: Availability
  derived allowToOpen: Boolean
  derived allowToClose: Boolean
definitions:
  function allowToOpen=
    if doorStatus=CLOSED  and
       trainMotion=STOPPED and
       trainPosition=ALIGNED then
           true
    else
           false
    endif
  function allowToClose=
    if doorStatus=OPENED and
       obstacleStatus=NOTEXIST then
           true
    else
           false
    endif
  rule r_closed_to_opening=
   if allowToOpen then
      doorStatus:=OPENING
   endif
  rule r_opening_to_opened=
    if doorStatus=OPENING then
       doorStatus:=OPENED
    endif
  rule r_opened_to_closing=
    if allowToClose then
       doorStatus:=CLOSING
    endif
  rule r_closing_to_closed_or_opened=
  if doorStatus=CLOSING then
    if obstacleStatus=NOTEXIST then
      par
        doorStatus:=CLOSED
        trainMotion:=MOVING
        trainPosition:=NOTALIGNED
      endpar
    else
       doorStatus:=OPENED
    endif
  endif
 rule r_EmergencySensing=
   emergency:=emergencySensor
 rule r_TrainMotionSensing=
   if doorStatus=CLOSED then
    if trainMotionSensor=STOPPED then
      par
        trainMotion:=STOPPED
        trainPosition:=trainPositionSensor
      endpar
    endif
   endif
 rule  r_ObstacleSensing=
   if doorStatus=CLOSING or
      doorStatus=OPENED then
     obstacleStatus:=obstacleSensor
   endif
// MAIN RULE
  main rule r_Main =
    if state=SENSING then
     par
       r_EmergencySensing[]
       r_TrainMotionSensing[]
       r_ObstacleSensing[]
       state:=EXECUTING
     endpar
    else
      if emergency=EXIST then
        doorStatus:=OPENED
      else
        par
          r_closed_to_opening[]
          r_opening_to_opened[]
          r_opened_to_closing[]
          r_closing_to_closed_or_opened[]
          state:=SENSING
        endpar
      endif
    endif
// INITIAL STATE
default init s0:
  function doorStatus = CLOSED
  function trainMotion = MOVING
  function trainPosition = NOTALIGNED
  function obstacleStatus = NOTEXIST
  function emergency= NOTEXIST
  function state = SENSING
```

Code 2.1: The AsmetaL specification for train door controller example

description of a specified path for the interaction sequences, that could include two actors: *user actor*, adopted UML actor, and *observer actor*, added actor. The user actor is capable of setting the external environment, i.e., determining the values of the monitored functions, and checking the machine outputs, i.e., the values of the updated controlled functions. The observer actor has further capabilities to check the internal configurations of the machine, to ask for the execution of the determined transition rule, and to oblige the machine to make one or more steps.

The AsmetaV supports a metamodel-based language, called **Avalla** (ASMETA Validation Language). This language has a list of syntactic commands to express executing scenarios as interaction sequences. The list includes: the **set** command for updating monitored or shared functions, the **step** for executing one ASM step, the **step until** for iterative executing more than one ASM steps until a specified condition is satisfied, **exec** for executing an ASM transition rule, and the **check** for inspecting the boolean value for internal property at the current state of the machine. Avalla has a well-defined semantics given in terms of ASM.

The AsmetaV is built around the Avalla scenario language, and the AsmetaS tool. AmetaV reads a scenario written in Avalla by the user, and invokes the AsmetaS to simulate this scenario. During simulation, the AsmetaV produces a *PASS/FAIL* verdict, to indicate the output results of the specified check actions. In this way, any check violation can be captured.

The ASMETA SMV (AsmetaSMV) model checker [31] is a formal verification tool of ASMs. It is designed to enrich he ASMETA framework with the NuSMV [75] model checking capabilities to verify temporal properties of ASM models. AsmetaSMV automatically maps the ASM model, which is written in AsmetaL, into NuSMV code, and runs the NuSMV tool on the translated code. The temporal properties, with AsmetaSMV, can be expressed as either Computation Tree Logic (CTL) or Linear-time Temporal Logic (LTL) formula. With AsmetaSMV, the user can directly add the temporal properties to the AsmetaL program, without the need to know the NuSMV syntax, but knowing the syntax of the AsmetaL language and the temporal operators is required. In our work for analysing safety-critical systems, we use the LTL formula, therefore we present just the syntax for LTL as shown in Table 2.1.

The advantages of using the AsmetaSMV, instead of NuSMV, is that the input language is easier to write and much more expressive, due to it's an extensive set of transition rules,

Table 2.1: The LTL operators and their corresponding function in AsmetaL

LTL operator	**AsmetaL LTL function**
◯ p	static x: Boolean → Boolean
□ p	static g: Boolean → Boolean
◊ p	static f: Boolean → Boolean
p $\mathcal{U}$ q	static u: Prod(Boolean, Boolean) → Boolean
p $\mathcal{R}$ q	static v: Prod(Boolean, Boolean) → Boolean

but the price is that the model checking is does not support every ASM element (domain, function, or rule construct). As a consequence, a user must be aware of the unsupported elements which as follows:

- the String, Char, any, and abstract domains are not supported. However, the String domains can be used in terms of numerated (enum) domains. The enum domains are finite user-named enumerations, e.g. one may define the following enumeration:
  enum domain MotionStatus = {MOVING | STOPPED}
  Note that the enumerated domain accepted an element consisting only of upper-case letters with length greater than or equal to two;
- the infinite Integer, and Natural domains are not accepted, although the concrete domains of Integer or Natural type are possible. The concrete domains are defined sub-domains of type-domains, e.g., consider the following domain:
  domain Numbers subsetof Integer
  domain Numbers={1 .. 3};
- declaring sequence Seq and set Powerset terms are disallowed;
- the constructs for sequential seq endseq, extension extend, iterative while while do, and turbo rules are not supported by the AsmetaSMV. The remaining rule constructs are allowable;
- using a function as an argument of another function to determine its location is not possible, as the tool cannot detect what NuSMV variable corresponds to this location;
- AsmetaSMV is not able to resolve the inconsistent updates problem for controlled functions. Where, the AsmetaSMV will assume that the controlled function has the first satisfied value, without giving any indication of the inconsistency. Therefore, before applying the AsmetaSMV, the AsmetaS must be run to discover any inconsistent updates for the specified controlled functions in the AsmetaL;

- not exhaustively defined derived and static functions, makes the AsmetaSMV gives an error signal stating that the conditions of the defined function are not exhaustive, see Code 2.2. In Code 2.2 a, the boolean derived function *allowToOpen* is not defined when the *doorStatus* is not equal to CLOSED;

```
function allowToOpen =
 if doorStatus=CLOSED then
   if trainMotionSensor=STOPPED and
        trainPositionSensor=ALIGNED then
      true
   else
     false
   endif
 endif
```

(a) The not exhaustively defined derived function

```
function allowToOpen =
 if doorStatus=CLOSED then
   if trainMotionSensor=STOPPED and
        trainPositionSensor=ALIGNED then
      true
   else
      false
   endif
 else
    false
 endif
```

(b) The exhaustively defined derived function

Code 2.2: Exhaustively and not exhaustively defined derived functions: AsmetaL model

- Updating the Integer controlled function to a value not belonging to the defined domain is impossible, see Code 2.3 a. This problem can be managed by adding a condition that states the allowed value, see Code 2.3 b.

```
asm update
import StandardLibrary
signature:
    domain Numbers subsetof Integer
    controlled incCon: Numbers

definitions:
    domain Numbers={1..3}

    main rule r_Main =
       inCon:= incCon + 1

default init s0:
   function incCon= 1
```

(a) The wrong update

```
asm update
import StandardLibrary
signature:
    domain Numbers subsetof Integer
    controlled incCon: Numbers

definitions:
    domain Numbers={1..3}

    main rule r_Main =
       if (incCon + 1) in Numbers then
          inCon:= incCon + 1
       else
          inCon:=1
       endif

default init s0:
   function incCon= 1
```

(b) The correct update

Code 2.3: Update the Integer controlled function: AsmetaL model

Other tools the ASMETA framework also includes additional tools (we do not present them, as they are not utilized in this work), such as: the *ASMETA Model Advisor* (AsmetaMA) [32] for identifying defects of AsmetaL models, the *ASM Test Generation Tool*

(ATGT) [110] for test cases generation, Asm2c++ [55] for automatic generation of C++ code from AsmetaL, etc.

As this work chooses ASMETA framework, we are grateful for the ASMETA teamwork assistance and active communication in several issues that we detected, such as:

- the validator wrongly parses the AsmetaL specifications that contain *extend* rule, but the ASMETA teamwork released a new version of the validator;
- a bug in handling the monitored strings during simulation, i.e. reading a string value from the environment. The ASMETA teamwork updated the plug-ins to solve this issue;
- the simulator does not implement some functions, such as toInteger("80"), toChar("f"), lt('h', 'g'), the ASMETA teamwork rebuilt the plug-ins and implemented them.
- the validator does not correctly validate specifications that include several dependant monitored functions. We highlight an example of reading the length and the elements of a sequence of integer numbers. Using AsmetaV to validate an ASM specification for this example by a scenario, that checks for example the value of the third element, does not give a right output. In such output, every element in the sequence is repeated several times equal to the entered length. The ASMETA teamwork is still working to find a correct way to handle this scenario of such example.

Language/Tool	Feature							
	Uses notation closed to ASMs	**Supports Basic ASMs**	**Supports Distributed ASMs**	**Avaliability**	**Incorporates time**	**Provides simulation**	**Facilitates validation task**	**Supports model checking activity**
DASL	No	Yes	No	No	No	Yes	No	No
ASM-WB	No	Yes	No	No	No	Yes	No	Yes
AsmGofer	No	Yes	Yes	Yes	No	Yes	No	No
XASM	No	Yes	No	No	No	Yes	No	No
AsmL	No	Yes	No	Yes	No	Yes	No	No
TASM	No	Yes	Yes	No	Yes	Yes	No	Only theoretically
RealTime ASMs Simulator	No	Yes	No	No	Yes	Yes	No	No
CoreASM	Yes	Yes	Yes	Yes	Only global system time is abstractly represented by a monitored function called *now*	Yes	No	Only theoretically
ASMETA	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes

Table 2.2: ASM languages and tool comparison

2.5 Why Abstract State Machines

There are a variety of formal methods, coming with their own merits and drawbacks. A well-known example is *Petri Nets* (PNs) [190], which is a graphical and mathematical modelling method. PNs are emerged for modelling and analyzing the behavior of asynchronous distributed systems depending on bi-partite graphs. However, it has been observed that PNs are inadequate for modelling distributed systems, due to providing insufficient abstraction and introducing algorithmically extraneous details, resulting from the token-based transition view of objects, make the refinement and analysis processes of the specified model a more complicated than necessary, failing to support convenient modeling paradigm of communication among agents and/or the environment, and PNs are dedicated for only small and medium size systems [59].

Another example that concerns the behaviour of concurrent/distributed systems, is called *Communicating Sequential Processes* (CSP) [121]. CSP is a process algebra notation for describing the interactions between a system and it's environment in terms of processes accomplished events. CSP is equipped with well-defined semantics. The CSP is considered as a pen and paper language. As a result, the current implementation tools, which are developed to prove and analyse CSP notation, adopt an input language called *machine readable CSP* (CSP_M) [206], which incorporates functional programming language and it does not support some of the idioms of CSP. To the best of our knowledge, implementing a truly encoded CSP notation is still an open problem.

Other methods, like *Vienna Development Method* (VDM) [50], *Zermelo* (Z) [221], *Alloy* [127], and *Bourbaki* (B) [20], are considered similar to ASM method with respect to supporting first-order data structures, providing abstract notations, and underpinning automatic application tools. Among these methods, the B method, a specification-oriented method, is the most similar one to ASMs. The reasons for that are its basic block is abstract machine and it includes the refinement concept, that leads in increasingly way to the generation of the executable code. However, unlike ASMs, the data structures in B are not pure first-order predict logic, as they also incorporate set theory. Its modelling methodology is close to an object-oriented style, and its syntax seems to be more complicated and less understandable comparing with ASMs. In addition, B [2] and the other methods

[2]The B method has been extended to the event-B method with slightly different mathematical languages to support the concurrent/distributed algorithms [68]. With respect to the applications, ASMs method seemingly is more mature than event-B, due to its wide successful application case studies.

(VDM, Z, and Alloy) are designated for specifying sequential systems, i.e. executing more than one update simultaneously is not supported [187]. Furthermore, Z and VDM are primarily employed for specifying the requirements and they both are not executable, i.e. there is an issue for converting the specification into executable code. Differently, from Z and VDM, Alloy is an analysis-oriented method, and its main focus is model checking the system. The modelling language for Alloy has several restrictions to avoid the state space explosion issue.

Among all these methods, this work chooses the Abstract State Machine method for several reasons:

- ASMs come with a generality feature which makes these machines versatile enough to adapt as needed to model a variety of systems, including sequential, parallel, and distributed with synchronous or asynchronous paradigm; without the necessity of assuming a certain size of a system being modelled. This serves our two directions work for developing safe systems and secure protocols.
- The ASM method provides a link between human understanding and formulation of problems under consideration and the deployment of their developed solutions.
- The ASM specifications are expressed in a simple and rich syntax with a straightforward and accurate semantics. This makes the ASM specifications understandable by being able to read and write them in a transparent way.
- The specifications for ASMs are executable for their simulation and validation, as they are in fact pseudo-code programs with precisely well-defined semantics on abstract data structures. [64].
- The faithfulness translation from the concrete problem into an abstract specification, without superfluous coding details.
- ASMs can non-deterministically describe the behavior of the environment through two ways: the *monitored functions* for modifying the machine by it's environment, and the *choose* construct for selecting an arbitrarily value in a manner unaccompanied by scheduling restrictions.
- ASMs with the ground model and stepwise refinement concepts, can design any system at the necessary level of abstraction, starting from capturing the requirements to the executable code.

- The ASM method serves in supporting the design, simulation, validation, and verification activities, within the open source ASMETA framework.

2.6 Conclusion

The Abstract State Machines are general mathematical machines for modelling hardware and software systems at the natural abstraction level. These machines are further developed into a system engineering method that seamlessly directs the development process of computational systems, from capturing the requirements to practical implementation.

There is a wide spectrum of executable ASM languages and analysis tools that have been developed throughout the years. Some of these languages and their supported tools are still around, while others are not, as their originators are graduated Ph.D. or no longer active. Most of the existing tools either adopt different notations of ASM models or they underpin only one aspect of the whole system development life-cycle. This work employs the ASMETA framework, as it is maintained and feature-rich project, and it supports notation language very similar to ASMs and provides tools served the development process as well.

Chapter 3

Safety and Security Analysis Techniques

3.1 Introduction

Both safety and security are crucial aspects in the development process of computational systems [220]. Safety refers to a state being devoid of/protected from unintentional accidents, whereas security represents a state that is free from/resisted deliberate threats. Another distinctive feature between these two aspects is: while security is dealing with environmentally originated risks that might affect the system, safety is focusing on risks which stem from the system and has potential effects on the environment [192]. System safety/security can be realized through safety/security analysis techniques, respectively. As safety and security can not be ensured, these techniques help to convey the systems' behaviour within reasonable limits.

Safety analysis aims at identifying hazards that could jeopardise a system's environment, determining potential causes and effects for hazards, tracing hazards that could lead to an accident, and eliciting requirements that manage the accidents.

Security analysis aims at identifying threats which may endanger a system, exploring system's vulnerabilities (weaknesses) that could be exploited by these threats, evaluating the risks of the indicated threats, decomposing each threat into possible attacks on a system, showing how these attacks can be caused, and eliciting requirements which can tackle or mitigate security problems.

Safety and security analysis techniques can be applied from the beginning of the sys-

tem development process, or can integrated within any stage of this process. The common stages of the development process are: *requirements elicitation*, *design* (designing a system in an abstract way), *validation* (showing that the designed system satisfies the users' requirement), *verification* (showing that the designed system meets its requirements specification), and *Implementation* (converting the abstract design into executable code). These stages can be adopted for safety-critical systems and security protocols [108].

This chapter presents an overview of a number of safety and security analysis techniques. There are a considerable number of techniques from both fields, but we cover just the techniques that are well-known in the literature and widely practiced in industry. However, before reviewing these techniques, we briefly explain the system development process, to grasp the role of these techniques. After that, we look in detail at safety analysis techniques. In addition, we evaluate the safety analysis techniques according to their capability to ensure safety in modern critical systems. This is followed by a concise review of common security analysis techniques, which includes an illustration of their principal ideas. We also present an evaluation of security analysis techniques to examine whether they are sufficient to elicit the necessary requirements for security protocols. Appendix A shows the application of the analysis techniques to an illustrative example with respect to both safety and security fields.

3.2 Software System Development Process

The software system development process or the software development life-cycle is a structured process consisting of a set of activities-based stages that help to manage the development and the implementation of a system.

Different models have been formed and developed to represent the development process. In order to illustrate the role of analysis techniques reviewed here in the development process, a simple safety V model [196] is sufficient for this purpose. The V model, as shown in Figure 3.1, has the V shape that incorporates the main activities in a typical development process. The left branch of the V represents the analysis and design phase. This phase starts with system analysis stage through identifying the hazards and eliciting the requirements which are related to the system's functions, or they are just constraints on its behaviour. There are many safety and hazard analysis techniques established and designed to serve this stage. Following this stage, the design of the system should be gradually specified, until the detailed and completed design is obtained. The progressive design

can be directed by the recommendations resulted from the failure causes and consequences analysis. The analysis and design phase produces the system design plus the safety-related requirements and recommendations.

The right branch of the V is the testing phase. The testing phase should be performed on the software using the requirements and design specifications. Typically, this phase includes activities for demonstrating that the safety requirements taken place in the right branch of the V have been met. The activities encompass validation and verification of the requirements for the abstract and detailed design in an integrated process. An interesting feature of the safety V model is using the feedback from the earlier phase to the later phase in an integrated way.

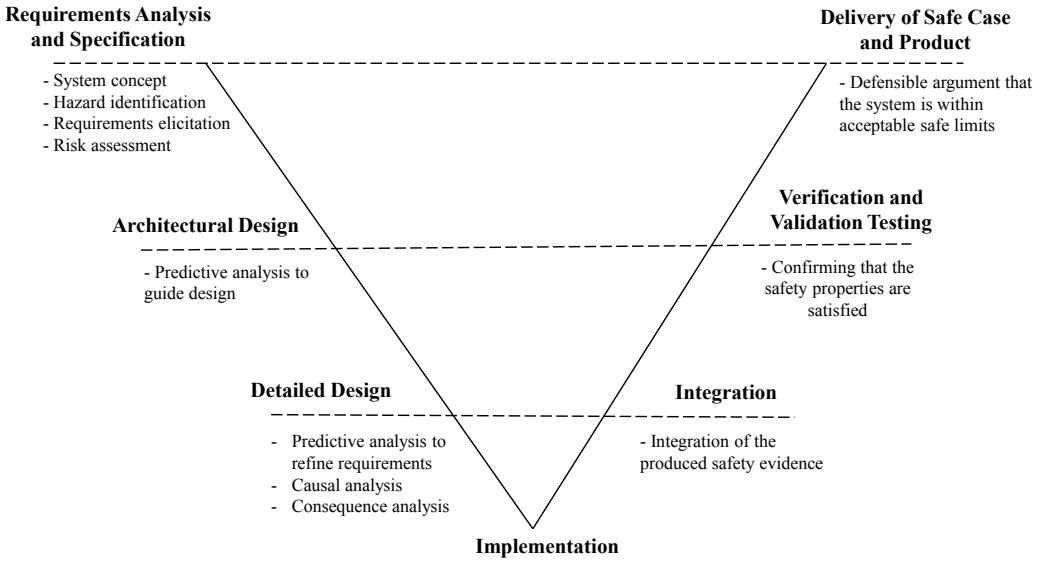

Figure 3.1: The safety V representation of the development process

3.3 Safety Analysis Techniques

There are a huge number of safety analysis techniques that have been developed to identify and analyse hazards. However, only some of these techniques are applied by safety analysts in practice [96], such as: Fault Tree Analysis, Failure Modes and Effect Analysis, and HAZards and OPerability Analysis. These techniques are known as traditional safety analysis techniques which are concerned with a system's component. On another hand,

there is a technique, called System Theoretic Process Analysis, whose concern is dynamic system behaviour. In this section each technique is described in a way that covers the following points: the *definition and application areas* where the techniques can be applied, the *distinctive features and advantages*, the procedure of *process* related to each technique, the suitable *time to use* these techniques with respect to the system development process, and finally the main *limitations* for these techniques.

3.3.1 Fault Tree Analysis

Definition and Application Areas: One of the well-known hazard analysis technique is the Fault Tree Analysis (FTA) [88]. This technique is firstly designed and developed, in the early 1960s, by H.A. Watson at Bell Laboratories, contracting with U.S. Air Force, to evaluate the Minuteman missile system [95]. A few years later, it was further developed, at Boeing Company, as a safety analysis tool applied in the commercial aviation [14]. After that, FTA has been successfully used for assessing the safety and reliability of various systems in different application areas, including but not limited to: transport [74], nuclear power plants [103], medical domain [39], related safety critical area [232], and so on. FTA is fully defined and described in the Fault Tree Handbook [231].

Distinctive Features and Advantages: FTA performs a top-down analysis starting with a given undesired event, which is hazard or accident, and tracing backward to determine the foreseeable and failure events that may give rise to the top level event. Typically, FTA is a deductive technique that diagnoses the critical failure combinations which likely cause a hazardous event, and it can calculate the probability for them [88]. The analysis by FTA is represented graphically using a logic tree-structured notation. The tree is built using the following two elements [88]:

Events are the nodes of the tree that comprises the hazardous event/root and its identified causes. The events are categorized according to their types, as follows:

- **Basic events** are the basic failure events that do not need to be expanded or investigated any more, as they explicitly defined. These events are the leaves of the tree and they represented by a circle shape.
- **Undeveloped events** are also the leaves failure events, but they can not be further developed, due to inadequate information is available about them, or

because they are considered unsuitable in this analysis. They have a diamond shape.

- **Intermediate events** are faulty events that requires further investigation. They are represented by a rectangle shape. The root of a tree is considered as an intermediate event.

Gates: shows the causal way of combining input/top events to produce output/below event. The most widely used gates in a fault tree are:

- **AND-gate** identifies the combination by which the co-occurrence of all input events is essential to produce the output event.
- **OR-gate** defines the combination whereby the output event will occur if any input event occurs.

Process: The analysis process of FTA is structured, systematic, and repetitive, where it is conducted in two steps [88]:

(1) Identify the top-level hazardous events.

(2) For each hazardous event, do the following:

   (a) Construct a fault tree, for the chosen event/the tree's root, by determining the combination of intermediate, basic and undeveloped events that may cause the root using Boolean logic gates.

   (b) Define the minimum *cutset*, which is the smallest collection of tree's leaves leading to the top hazardous event.

Using Time: FTA can be applied at any stage of the system development life-cycle.

Disadvantages: FTA does not have explicit guidance to develop its tree structure [96]. However, it has certain questions answered at each gate level to determine it's type and inputs. These questions have two concepts: *state-of-the-system* (SS), a fault does not arise from component failure, and *state-of-the-component* (SC), a fault that does arise from component failure. The SS questions are: "What is immediate (I), necessary (N), and sufficient (S) to cause the event?". While the SC questions are: What are the Primary (P), Secondary (S), and Command (C) causes of the event?". These questions enable the FTA user to focus on determining the next failure element in the cause-effect sequence style. However, it seems to be not obvious how answering these questions gives enough

guidance during the analysis process. The application of FTA needs a practically and theoretically experienced analyst to identify the main causes of hazardous events.

3.3.1.1 Failure Modes and Effect Analysis

Definition and Application Areas: The Failure Modes and Effect Analysis (FMEA) is, as its name suggests, a technique for identifying the potential failure modes of the system's components and evaluating their possible effects [81]. The FMEA technique stems from the development of the U.S. Military procedure MIL-STD-1629A [12], which is an evaluation procedure of failure modes for weapon systems. Since then, it has been applied in a wide range of industries, such as: aerospace [133], automotive [195], space rocket [203], textile industry [186], and so on.

Distinctive Features and Advantages: FMEA has a detailed version, known as Failure Mode, Effects and Criticality Analysis (FMECA), where the evaluation of the criticality for failure modes is added [81].

The FMEA/FMECA is a bottom-up inductive technique, that starts with a system description, and main constraints for its success and failure. The analysis output of FMEA/FMECA is documented in a worksheet, which contains a description for the following column headers [81]:

- **Component** is either a part of a system or subsystem being analysed, or a step of a process under analysis.
- **Failure mode** is a way in which a component might fail; the mode of a component after failure.
- **Cause** is the potential factor(s) causing the failure mode.
- **Effect** is a serious repercussion(s) or consequence(s) failure mode on a whole system and it's environment.
- **Severity classification** is a classification or ranking of the severity for the failure mode, using the following severity levels: catastrophic, critical, marginal, and negligible.
- **Criticality ranking** is a qualitative or quantitative measure of failure mode criticality. This column is only included in the FMECA worksheet.

- **Recommendations** are possible suggestions for mitigating or eliminating the failure mode effects.

Process: The FMEA/FMECA can be applied through a structured process consisting of the following steps [81]:

(1) Define the system under analysis, it's functional requirements, and it's main assumptions and constraints.

(2) Construct the FMEA/FMECA worksheet to record the results of the analysis session.

(3) Pursue the analysis by conducting the recommendations when it is possible.

Using Time: FMEA/FMECA should be applied as early in the development life-cycle as possible to make early change to the system design according to the obtained results from this technique [96].

Disadvantages: In [96], it has been shown the following limitations. The FMEA/FMECA does not provide any guidance to prompt imaginative thinking during the analysis session. As a result, conducting the FMEA/FMECA requires a multi-disciplined and experienced team to increase overall creativity of the process. FMEA/FMECA can not determine hazardous and unpredictable effects unrelated to the considered failure modes.

3.3.2 HAZards and OPerability Analysis

Definition and Application Areas: The HAZards and OPerability Analysis (HAZOP) study was originally developed by Imperial Chemical Industries (ICI) in the United Kingdom, as an analysis technique for identifying and assessing hazards and operability problems in the pipework and instruments design of a chemical plant [76, 119]. It has since then been used in many domains, including but not by way of limitation to nuclear [199], pharmaceutical [120], safety-critical domains [66], etc.

Distinctive Features and Advantages: The HAZOP conducts inductive bottom-up analysis by examining how deviations from the aim of system design or the intention of the planned operation can result in hazardous outcomes [84]. The examination of these deviations is based on using a set of guide words, listed in Table 3.1, as a facility for perceiving deviations applied for each system's part.

The findings of HAZOP are reported in a worksheet, which includes information about the following column headings [84]:

Guide Word	Meaning
No	The design intention is not attained
More	A quantitative increase
Less	A quantitative decrease
More than (As well as)	The design intentions achieved with additional output
Part of	The design intention is partially achieved
Reverse	The opposite or contrary to the intention happens
Early / Late	Something occurs in a different intended time
Before / After	Something occurs in a different intended order or sequence

Table 3.1: HAZOP guide words

- **Item** is an entity, a function, a process, or any part of a system being analysed.
- **Parameter** is an attribute related to the item under analysis, which affects the item's operation comparing with its design intention.
- **Deviation-Guide Word** is a deviation from the design aim, that is determined by combining a guide word with a parameter, and provoking a question about whether such a combination could arise a deviation.
- **Cause** is a potential causal element that contributes to producing a defined deviation.
- **Consequence** is a hazardous effect of the identified deviation.
- **Risk** is two qualitative words describe how the consequence, described above, could be risky. The two words are chosen from risk's *severity*: "catastrophic", "critical", "marginal", "negligible", and *probability*: "frequent", "probable", "occasional", "remote", "improbable".
- **Recommendations** are possible recommended actions for alleviating or removing the hazardous consequences.

Process: The process of HAZOP is accomplished in an examination session, during which a multi-disciplinary team performs the following steps [84]:

1) Determining the main items or components of the system.
2) Select an item and define its design intent and its parameters or attributes.
3) Locate a deviation that can lead to a hazard, by combining a suitable guide word with an item's parameter.
4) Identify the cause and consequence of the defined deviation.
5) Suggest remedial measures.

6) Record the outcomes in a worksheet.

7) Repeat steps 2 to 6 for each item.

Using Time: HAZOP should be used in the early design stage, but it entails that detailed system design is near completion to be analysed effectively, i.e., identify the possible recommendations that reduce the possibility of the hazards.

Disadvantages: HAZOP, though gives guidance to the analysis session by its flexible guide words, requires a trained and experienced multi-disciplinary teamwork. HAZOP also lacks guidance on determining the causes and effects of deviations. Furthermore, focusing on guide words lead to ignoring some hazards not associated with a guide word [96].

HAZOP, like FTA, ET, and FMEA techniques, focuses on a single part of a system design instead of considering the system as a whole. Moreover, this technique suffers from repetition.

3.3.3 System Theoretic Process Analysis

Definition and Application Areas: System Theoretic Process Analysis (STPA) is a safety analysis technique that is designed with the aim of providing more advantages over traditional safety analysis techniques, through changing the analysis focus from a single component failure into a whole system dynamic behaviour [149]. In fact, STPA is similar to FTA in terms of following a top-down analysis strategy, but instead of identifying only the hazard causes (scenarios) that originate from a failure, STPA adds an extensive causes set, including causes related to failed interactions among non-malfunctioning components, human behavior mistakes, software errors, system design and requirement flaws [152].

STPA technique was originally applied to analyse a complex software-intensive system in aerospace, which was unmanned space vehicle, known as H-IB Transfer Vehicle (HTV) [126], but has since then been used for many systems in different application areas, including but not limited to: Aircraft Wheel Braking system [154], Adaptive Cruise Control system [16], PROSCAN Proton Therapy system [29], Space Shuttle Operations [185], Robotic Telesurgical System [26], etc.

Distinctive Features and Advantages: STPA uses a type of accident causation model, called System-Theoretic Accident Model and Processes (STAMP) [146, 147]. STAMP

changes the accident causation models from being chain-of-failure-event causality models, which underlie most of traditional safety analysis techniques, into systematic models, whereby accidents emerge when the interactions among the system components: human, social, physical, software, violate the safety constraints related to these components. STAMP is built on *systems theory*, which deals with a complex system by structuring it as a hierarchical controlled levels. This is different from the traditional *reliability theory*, which handles the behavioral complexity of a system by dividing this system into separated components and proving the functions for each component [152].

Based on STAMP, STPA deems safety as a system control problem instead of a component failure problem. In a typical manner, STPA models a system as a functional control loop (or what is called functional control structure), and it views accidents as the consequence of inadequate control. Figure 3.2 presents a simple control loop. For complex systems, this loop may contain multiple controllers ordered hierarchically. The simple loop is formed of a controller, actuator, sensor and controlled process. The controller includes a model of the process which it is controlling its behavior. The process model helps the controller to decide which control action to be issued. The actuator executes this action on the controlled process, and the sensor updates the process model for the controller by returning to it the current status information about the controlled process. By this system theoretic view, the accidents can be investigated though checking when the controlled process may produce unsafe behavior, and the conclusion can then be made by considering that the behaviors of other components are hazardous and the main causes for the controller process outputs.

In addition to identifying the potential causes of a predicted accident, STPA helps to elicit safety requirements (safety constraints) which will be imposed on the system design to generate safe behavior. Safety requirements are elicited by examining each control action, that changes the controlled process status, under different timed provision conditions which affect the safety of action, and determining whether this action associating with these conditions is a hazardous one [149]. As a result, eliciting safety requirements by STPA is based on a *what-if* query on every control action.

Process: STPA has a guided and systematic process. The main terminology that has been used with STPA process are [149]:

- **Accident:** An undesired event that occurs unintentionally (but not necessary unex-

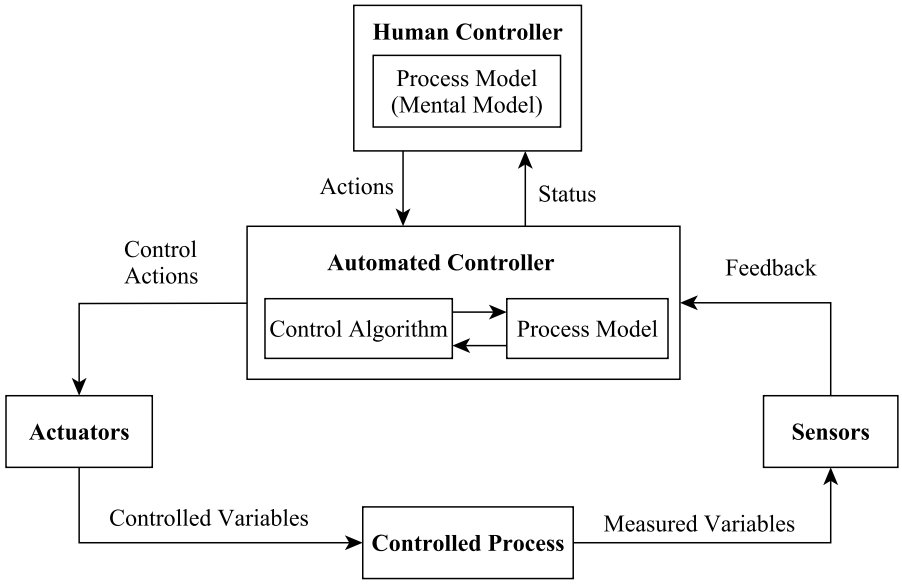

Figure 3.2: A basic control loop (adapted from [146])

pectedly) and brings an unacceptable loss.

- **Hazard:** A set of system conditions that, if combined with a specific set of worst-case environmental conditions, will result in an accident (loss).
- **Unsafe control action:** A hazardous command which might happen when it is provided or not provided as required from the human or automated controller.
- **Process model:** A set of system and environmental variables updated by different status feedback of the system's components. This set is utilized to take a decision by a controller to provide or not provide a control action.
- **Safety constraints:** Safety conditions that safeguard a system against potential accidents.
- **Causal Factors:** The main causes that would explain the accident scenarios for how unsafe control actions might violate safety constraints.
- **Controlled process:** A system's process or a component that is under the control of the activities for human or automated controller.
- **Controlled variable:** A variable whose values manipulated by the actuator when the action is provided.

- **Monitored variable:** A variable whose values that are set by the human or environmental controller.
- **Measured variable:** A variable which keeps the state of the controlled process to be used by the sensor.

STPA process is described in two steps [149]:

Step 1 Identify the possible control actions that could lead to a hazardous situation, as follows:

Step 1.1 Construct the analysis foundation by identifying the expected accidents of a system, and the possible hazards which likely give rise to these accidents.

Step 1.2 Draw an exploratory safety control loop diagram, which shows the system's components and their control and feedback tracks.

Step 1.3 Ask an expert about how could each control action related to each controller leads to the identified system hazard(s). An expert is guided by the following four conditions to identify unsafe actions:

- An essential control action for safety is "not provided" or not issued.
- An unsafe control action is "provided" when it is not required and this causes a hazard.
- A safe control action is provided "too early" or "too late" in wrong order or time.
- A safe control action (only continuous action) is "stopped too early" or "applied too long".

Step 1.4 Document each unsafe control action in a table whose columns representing the provision conditions for each unsafe action, see Table 3.3. The entries of this table can then be mapped into high-level safety requirements or what are called safety constraints.

Step 2 Identify the causal factors (scenarios) for each unsafe control action defined in Step 1 to realize how these actions could occur. This step is guided by the functional control loop. Each causal factor can then be translated into low-level safety requirements that help to mitigate against the severity of unsafe actions.

Our work for analysing safety-critical systems focuses on the first step of the STPA process. The methodological procedure for achieving STPA **Step 1** was extended into a more

systematic one which gives enough guidance for identifying unsafe control actions based on a defined formal structure for every action. An unsafe control action is formalized as a four-tuple ($\mathcal{CA}$, $\mathcal{C}$, $\mathcal{T}$, $\mathcal{CO}$), where:

- $\mathcal{CA}$ is the control action that should be analysed.
- $\mathcal{C}$ is the controller that issues the $\mathcal{CA}$ control action.
- $\mathcal{T}$ is the type of the $\mathcal{CA}$ which can be either *Provided* at any time, too early, too late, or *Not provided*.
- $\mathcal{CO}$ is the context of providing/not providing $\mathcal{CA}$, which can be constructed by combining values for the variables in the process model. More precisely, $\mathcal{CO} = (\mathcal{CO}_1, \mathcal{CO}_2, ..., \mathcal{CO}_n)$, where each $\mathcal{CO}_i$ is a pair of the $\mathcal{V}_i$ variable in the process model that has $v_{i,j}$ value, i.e., $\mathcal{CO}_i = (\mathcal{V}_i, v_{i,j})$.

In general, the extended method for STPA **Step 1** is a formal process for generating tables, known as *context tables* facilitating the safety evaluation for every action. The first column of a context table is the control action under analysis. The second column represents the process model variables for this control action, which can be filled depending on the values of these variables. While the last column denotes the decision that should be taken by an expert on whether this action is hazardous. The extended method for STPA **Step 1** can be summarized into the following steps:

1. Determine the relevant variables of the process model for each control action which is issued from a controller in the functional safety control loop.
2. Construct context table for each action with respect to the four provision condition. Table 3.2 is a generic context table for evaluating the $\mathcal{CA}_i$ control action.
3. Review and evaluate each row in the context table by an expert. An expert can use yes/no to denote the presence of a hazard or not. However, we find that the evaluation is not a closed-ended process since providing/not providing some actions with a specific context will not lead to a hazard, but there is a flaw with the system's function, see Chapter 5.

Control Action	**Process Model Variables**				**Hazardous Action?**			
	$\mathcal{V}_1$	$\mathcal{V}_2$	...	$\mathcal{V}_n$	**Provided at any time**	**Provided too early**	**Provided too late**	**Not Provided**
$\mathcal{CA}_i$	v_{11}	v_{21}	...	v_{n1}	yes/no	yes/no	yes/no	yes/no
	...	...	...	...	yes/no	yes/no	yes/no	yes/no
	v_{1n}	v_{2n}	...	v_{nn}	yes/no	yes/no	yes/no	yes/no

Table 3.2: A generic context table of providing the $\mathcal{CA}_i$ control action

STPA has been applied to train door controller example [227]. As the work in this thesis focuses only on the first step, we present the results from the first step in Table 3.3, which must be converted into a safety requirement using *must/must not*. For example, the unsafe action: open action is not provided once the train stops at a platform, can be translated into the following safety requirement: open action must be provided once the train stops at a platform. Accordingly, Table 3.3 shows the outcomes (unsafe actions) that are obtained from answering questions in the context table.

Control Action	**Not Providing Leads to Hazards**	**Providing Leads to Hazards**	**Wrong Timing or Order Leads to Hazards**	**Stopped Too Soon or Applied Too Long**
Open the door action	Open action is not provided once train stops at a platform Open action is not provided for emergency evacuation Open action is not provided while a door is closing on a person	Open action is provided while train is in motion situation Open action is provided while train is not aligned at a platform	Open action is provided before train has stopped or after it started to move Open action is provided later than the train has stopped Open action is provided later than an emergency occurs	Opening the door is stopped too soon during normal stop Opening the door is stopped too soon during emergency stop
Close the door action	Close action is not provided before moving	Close action is provided while a person in the doorway Close action is provided during an emergency evacuation	Close action is provided too early, before passengers finish entering/existing Close action is provided too late, after train starts moving	Closing the door is stopped too soon, or not completely closed

Table 3.3: Unsafe control actions for train door controller example (adapted from [227])

Using Time: STPA technique can be applied at the development stage, as it can be used to analyse a system that is already available or it is under design.

Disadvantages: We observe that STPA has no means to identify the accidents and hazards of a system and to ensure the completeness of these hazards.

3.4 Evaluation of Safety Analysis Techniques

Given the complexity of modern safety-critical systems and their increasing reliance on software, this section evaluates the safety analysis techniques with respect to their ability to analyse the safety for modern critical systems. In Table 3.4 though, we summarize the main features of these techniques based on their application, which is show in A, to the TDC example.

FTA and FMEA techniques are considered as failure-based methods, as they are generally developed to capture the causes of accidents that are component failures-originated. However, when a component of a system is a software, then these techniques, in almost all instances, can not address the software errors for this component, or in their best situation they will only provide a general cause labeled as undeveloped software failure [145].

In addition to determining the component failure causes, HAZOP technique emphasizes the deviations, which are based on guide words, from the component design aim during identifying hazard's causes. The HAZOP success depends on the choice of guide words that point out deviations, the availability of the design details, and the understandability of the system's behavior. Concerning software systems, the HAZOP guide words are insufficient to address software errors, like the wrong value of delivered data, and the modern software systems behavior is unexpected or unanticipated and slightly understandable [197].

A variety of modifications have been added to these techniques to better identify the software-related hazards, such as Software FTA (SFTA) [151], Software FMEA (SFMEA) [196].

SFTA [151] is an extended version of FTA, that is employed to detect the possible safety-related code errors and software bugs for the system under analysis. SFMEA [198] is an advanced form of FMEA that is applicable to a system contained software faults which could cause failures and it is capable of stating the effects of these failures. SHARD [196] is originated from the HAZOP technique with some extra variations regarding the

employed guide words and the guidance for identifying the potential software failure causes. SHARD was applied to analyse the data(information) flows in a software system.

Despite these extensions, the techniques with a single software failure focus, seem less useful to capture the major causes of the accidents for modern systems which are: the flawed requirements and misapprehension about the hazards that are associated with software functionality [148, 160]. In fact, unlike the hardware, the software is not harmful by itself. It fails only functionally but not haphazardly [96].

All the above techniques can not analyse multiple failures at a time and deal with the dynamic interactions between components. Thus, these limitations together with the inability to address software-related hazards make them limited in analysing the modern complex systems.

STPA is a relatively new analysis technique as compared with these techniques. It is capable of capturing more hazard causes than the transitional techniques, in particular, those are pertaining to requirement flaws, design errors, inadvertent human behavior, component interaction failures, and software errors [149, 150, 105].

The key advantage of STPA is taking into account functional interactions among system components during the analysis of expected system accidents. In other words, unlike other techniques that consider accidents stemming from a component failure chain, STPA relates accidents to the un-imposing safety constraints on functional interactions among system components.

The capability of identifying the requirement flaws and locating safety constraints related to functions enables the STPA to analyse the software safety of a system. Moreover, the safety constraints identified by STPA has been mathematically formalized and this assists to enforce safe behavior during the development process [226].

	FTA	FMEA	HAZOP	STPA
Purpose	Identifying the root causes of a top-hazardous event	Analysing known component failures and evaluating their effects	Evaluating hazards and operation problems that are arose from component design aim deviations	Identifying accident causes that arise from inadequate enforcement of constraints on system dynamic behavior
Application area	It mainly covers: missiles, spacecraft, and aviation areas	It mostly covers military and aerospace areas	It is widely applied to chemical, pharmaceutical, and nuclear areas	It is particularly applied to aerospace, medical, robotic areas focussing on complex software controlled systems in these ares
Analysis method	Deductive top-down	Inductive bottom-up	Inductive bottom-up	Deductive top-down
Analysis documentation facility	Graphical structured tree	Textual table	Textual table	Textual table
Guidance	State of the system/component questions	No explicit guide	Guide-words associated with component's parameters	What-if questions associated with timed provision conditions
Special features	Presenting graphical based boolean logic models for the causes of component failures; calculating the probability of component failure	Analysing the function of a single component at a time; showing the effects of functional failure in a structured way	Determining the deviations from the component design intention in a structured and predictive way	Considering the whole system during the analysis; focusing on the interactions among system components; dealing with complex software-intensive systems

Table 3.4: Summary of safety analysis techniques

3.5 Security Analysis Techniques

This section reviews a number of analysis techniques from the security domain. The key features of these techniques are discussed together with the main principle of their processes. Some of the reviewed techniques were originally developed to analyse security protocols in an informal way, but the formal analysis techniques for protocols will be surveyed and discussed later in Chapter 4.

3.5.1 Attack Tree

Definition and Application Areas: The Attack Tree (AT) is a way of depicting and investigating the security of a system based on the possible attacks [211, 213]. This technique models a system's attack in a way that helps to describe how an attacker would carry out potential attacks, and to identify the countermeasures to thwart them. An AT constitutes the base of understanding the security process for hardware and software systems belonging to any domain [211].

Distinctive Features and Advantages: Although AT is similar to FTA in terms of employing graphical logic-based diagram, it is rather a security breach analysis technique, not a failure analysis one. In particular, it uses a tree structure to represent attacks upon a system, where the topmost node (root) is the attacker goal, and the nodes below are sub-goals or different ways of attaining the main goal. As in FTA, the AND and OR gates are used to describe the relationship between the parent node and its children. It is possible to assign values to tree nodes, starting from the leaf nodes and then moving up to calculate the cost of an attack and probability of the success, or to estimate the need for special equipment, the possibility, and the legality of the goal, and so on [211].

Process: An AT has an ad-hoc and repetitive process consisting of the following steps [211]:

(1) Define the anticipated attack goals (possible attacks on a system).

(2) Construct a separated tree for each goal, where the root of the constructed tree is further refined into sub-goals, and the refinement process is repeated for each sub-goal node until no more refinement is possible and the top attack goal is achieved.

(3) Optionally assign a value (numerical or boolean) to each node in every tree to determine the probability of attacks success.

Using Time: AT can be applied in the late stage of the development process, to have enough knowledge about the components that might be assaulted [211].

Disadvantages: Seemingly, AT lacks adequate guide for identifying all the attacker goals and for thinking about all the ways for achieving the goals. Hence, it needs knowledgeable and experienced analyst.

3.5.2 Vulnerability Identification and Analysis

Definition and Application Areas: Based on HAZard and OPerability (HAZOP) and Software Hazard Analysis and Resolution in Design (SHARD) techniques, a Vulnerability Identification and Analysis (VIA) technique was designed for the security protocols domain [107, 108]. VIA aims to analyse and extract further requirements for security protocols, starting with the initial requirements that must be achieved by them.

Distinctive Features and Advantages: VIA allows analysing both initial and more detailed requirements to identify vulnerabilities and weaknesses at a various level of abstraction. VIA is similar to the HAZOP and SHARD techniques in terms of employing a set of guide words for pinpointing the deviations, and a tabular worksheet for documenting the analysis output. However, the guide-words for VIA were chosen in a way that enables to address issues relating to protocols, such as disclosing secret message and repeat sending a message, see Table 3.5 which enumerates the VIA guide words and their meaning. In general, the guide words aid at determining deviations to protocol requirements which may lead to a situation in which the protocol is vulnerable to an attack and its desired properties are violated. Furthermore, the worksheet includes the following column headings: *Guide word-Deviation*, *Causes*, *Effects*, *Recommendations*, and *Comments*, which are slightly different from the ones for HAZOP. Typically, it does not include the parameter column heading, as the deviation is identified by combining a guide word with functional requirement instead of the parameter, and it has an extra column heading which is *Comments* for noting down assumptions about the analysis, which do not suit other headings.

Process: VIA has a structured and iterative process adopting core ideas from both HAZOP and SHARD techniques. The process of VIA can be described through the following steps [84]:

Guide Word	Meaning
Omission	The event does not occur
Inclusion (spurious)	The unexpected event occurs once
Inclusion (repetition)	The unexpected event occurs several times
Value	The information in the event has incorrect value
Disclosure (external)	The information in the event has been revealed to an intruder
Disclosure (internal)	The information in the event has been revealed to participants taking part in the protocol
Early (absolute)	The event occurs earlier than the intended real-time deadline
Early (relative)	The event occurs earlier than another intended event
Late (absolute)	The event occurs later than the intended real-time deadline
Late (relative)	The event occurs later than another intended event

Table 3.5: VIA guide words

(1) Determine the high-level functional requirements of a protocol under analysis. The functional requirements are determined by defining what the protocol should achieve with respect to the protocol's parties and their actions.

(2) Select a high-level functional requirement and combine it a guide word to locate a deviation.

(3) Analyse the causes of the located deviation based on the Primary-Secondary-Command rule, adopted from FTA. This rule helps to identify causes originated from the failure of the participant to carry out its particular action (primary cause), the failure of the communication medium (secondary cause), or the failure of the event which prompts the next action to be carried out (command cause).

(4) Highlight the effects of the identified deviation using the question: "Does this deviation have any security implication on the protocol?". This question can be answered while testing how can the messages, message components, the protocol's run, an attacker's and participants' knowledge be affected.

(5) Suggest a recommendation to address the defined deviation. The recommendations represent low-level requirements, which elicit by putting forward a measure that could prevent a violation to the assumed security property of a protocol, or that could detect when such a deviation has occurred and how to recover from it.

(6) Select another guide word, combine it with the functional requirement, and repeat step 3 to 5 until no more guide words left.

(7) Select another high or low-level requirement and combine it with a suitable guide word and repeat step 3 to 6 until no more requirements left.

Using Time: VIA should be used at the early stage of the development process, i.e., before the actual design of a protocol is produced [108].

Disadvantages: VIA, despite providing guidance for identifying the vulnerabilities using a set of guide-words, it seems to be restrictive to specific vulnerabilities determined by this set. Furthermore, VIA is time-consuming and suffers repetition in the elicited requirements.

3.5.3 Requirements Analysis and Elicitation

Definition and Application Areas: The Requirements Analysis and Elicitation (REA) is a tree technique for analysing and eliciting requirements for security protocols [108]. It is developed depending on a number of tree based analysis techniques from safety and security domains, including fault tree analysis, event tree, threat, and attack tree techniques. REA can be applied to any security-critical system, in addition to security protocols, for identifying vulnerabilities which in turn facilitates the elicitation of more detailed requirements.

Distinctive Features and Advantages: REA is a goal decomposition technique for investigating the potential vulnerabilities of the protocol. It starts from an undesired event (negated protocol goal) and decomposes it down to the basic events (basic vulnerabilities) that lead to its occurrence. This allows traceability of the identified vulnerabilities. REA documents its findings in a graphical tree similar to the one used in fault tree analysis technique in terms of using graphical representation for basic, intermediate, and undeveloped events, and employing boolean gates for showing the relationships between these events. However, it adds an oval shape for incorporating assumptions made by the analysis into the tree diagram. The documented assumptions provide a clear description of when a specific issue emerges in the analysis.

Process: The analysis process of REA is structured, systematic, and repetitive, where it is conducted in two steps [108]:

(1) Identify the top-level undesirable events, by negating the high level security protocol requirements.

(2) For each insecure event, do the following:

   (a) Construct an REA tree, for the chosen event, through determining the combination of intermediate, basic and undeveloped events that may cause the chosen

event using Boolean logic gates. The determination of these events is performed by asking the question "What do we mean by [the event under consideration]?" and answering it based on two ways. Either by checking various security properties, such as availability, confidentiality, authenticity, and etc., or by using the Primary-Secondary-Command rule, adopted from FTA, which help to identify the events originated from malicious and non-malicious actions. Any assumption made during the analysis must be linked to the appropriate basic event.

(b) For each determined event, repeatedly define further low-level events until the events cannot be further decomposed.

(c) For each basic event, identify the recommendations for overcoming the discovered vulnerabilities in the security protocols.

Using Time: REA should be used at an early development stage, i.e., before the actual design of a protocol is produced.

Disadvantages: As REA starts by negating the protocol's goal, it may overlook some vulnerabilities during the analysis [108]. Accordingly, if REA is integrated with another technique for identifying all the possible undesirable situations of a protocol, then this will positively enhance the REA findings.

3.5.4 System Theoretic Process Analysis for Security

Definition and Application Areas: The System Theoretic Process Analysis for Security (STPA-Sec) is an extended version of the STPA technique that includes security analysis for cybersecurity problems [242]. It was designed to control the vulnerabilities that could be exploited by threats instead of identifying all threats that can potentially cause security incidents. It has been applied to safety-critical cyber-physical systems, such as ballistic missile defense system.

Distinctive Features and Advantages: STPA-Sec is a top-down technique used to identify the security vulnerabilities for a system and the main causes that lead to violating its security constraints. It considers security as a system vulnerabilities control problem rather than a threat guarding problem. Controlling vulnerabilities means preventing incidents that arise from known (inside the system) and unknown (outside) threats. STPA-Sec, like STPA, employs functional control diagram which is structured hierarchically to model a system. In this diagram, the control actions issued from higher level controllers will be

executed via actuators at the lower level processes, whereas feedbacks are provided from the lower level to the higher level via sensors. STPA-Sec investigates how could the functions of the system's components (controller, actuator, sensor, and controlled process) be affected by the external attacks.

Process: STPA-Sec has a process similar to that used by STPA. Though all the steps in both techniques are identical, the STPA-Sec includes terminology suitable for security contexts, such as *losses, vulnerabilities*, and *insecure control actions*, in addition to terms of *accidents, hazards*, and *unsafe actions* that are used by STPA. The process of STPA-Sec can be described through the following steps:

Step 1 Identify the possible control actions that could lead to a hazardous situation, as follows:

Step 1.1 Identify the losses of a system whether intentional or not, the hazards, and the vulnerabilities.

Step 1.2 Create an exploratory functional control loop diagram, which shows the system's components and their control and feedback tracks.

Step 1.3 Determine unsafe control actions that could lead to hazards or insecure control actions that could lead to the identified vulnerabilities.

Step 1.4 Ask an expert about how could each unsafe/insecure action lead to the identified hazards/vulnerabilities. Answering this question is based on the following four conditions:

- A required control action is "not provided" or missed.
- An incorrect control action is "provided".
- A correct control action is provided "too early" or "too late".
- A correct control action is "stopped too early" or "applied too long".

Step 1.5 Document each unsafe/insecure control action in a table whose columns represent the provision conditions for each action. The entries of this table can then be mapped into high-level safety/security requirements.

Step 2 Identify the causal factors (scenarios) for each unsafe/insecure control action defined in Step 1 to realize how could these actions occur. This step is guided by checking the correctness, appropriateness, and provision of the information for components in the functional control loop. Each causal factor can then be translated into low-level safety/security requirements.

To understand the above steps, we can illustrate then to our given example that we called a Single Authentication Protocol (SAP). This example is an authentication protocol for two participants: initiator and responder. The protocol aims to authenticate the responder to the initiator through exchanging two messages. In the first message, the initiator sends its identity together with a randomly chosen nonce to the responder, i.e., the initiator tells the responder, *This is my identity and I want to communicate with you. The nonce is my challenge*. Upon receipt, the responder encrypts the nonce with a secret key shared between it and the initiator. Once the second message is decrypted and the nonce checked by the initiator, this will confirm that the responder is an authenticated participant.

By applying the first step of the process for STPA-Sec technique to the SAP protocol, we obtain the following losses for this protocol:

L1 The initiator wrongly believes that the responder is an authenticated participant.

L2 The responder incorrectly believes that it is communicating with the true initiator.

The vulnerabilities related to the above losses are:

V1 The inability of the responder to prove that the initiator is alive at the executing protocol run.

V2 The initiator is unable to guarantee the freshness of the responder's message.

V3 The association of the nonce with its legitimate participant is not explicitly stated

Based on the identified vulnerabilities, we document the insecure control actions in Table 3.6.

Using Time: STPA-Sec should be used at an early development stage and in situations where a specific system's component is not available.

Disadvantages: STPA-Sec does not provide enough guidance to determine the vulnerabilities of a system and the main causes of insecure actions. It also can not address some critical problems, like non-repudiation, confidentiality, and anonymity. We believe that a significant improvement could be made on STPA-Sec by extending the functional control diagram to model the attackers.

Control Action	Not Providing Leads to Vulnerabilities	Providing Leads to Vulnerabilities	Providing Too Early or Too Late Leads to Not Providing Leads to Vulnerabilities	Stopped Too Soon or Applied for a Long Time
Sending the first message	N/A	The first message is sent several times (V1)	N/A	N/A
Sending the second message	The second message is not sent when the initiator already sent its message (V2)	The second message is sent by the responder while the initiator did not send any message before (V1)	The second message is sent earlier than the initiator's expectation (V1, V3) The second message is sent later than the initiator's expectation (V1, V3)	N/A

Table 3.6: Insecure control actions for single authentication protocol

3.6 Evaluation of Security Analysis Techniques

As one of the concerns for this thesis is analysing security protocols, we evaluate the security analysis techniques reviewed here, in terms of their applicability to analyse the security protocols and to elicit the security requirements for these protocols. The application of these techniques to the single authentication protocol is shown in Appendix A. We recap the general features of these techniques in Table 3.7.

Attack Tree technique helps to identify the effects of attacks on a system, and to determine the route that an attacker may take to attain its goals. Regarding security protocols, AT can provide a description of the possible attacks on the protocol that shows the attacker's capabilities. However, it guarantees neither describing all the possible attacks upon a protocol, nor covering the attacker's capabilities comprehensively.

VIA is a systematic technique tailored for analysing security protocols. It inductively applies a set of guide words for eliciting requirements to avoid the possible vulnerabilities. However, this technique suffers from repetition in the extracted requirements and its analysis is seemingly restricted to specific vulnerabilities determined by the utilized guide words.

REA is another technique aids to analyse security protocols. Its intersecting feature

is the clarity of showing the exact path for exploiting a potential vulnerability. However, being solely dependent on decomposing the negated protocol goal into vulnerabilities could result in overlooking some vulnerabilities during the REA analysis.

STPA-Sec is a useful technique when applied to software-intensive system where both safety and security aspects are considered. Concerning the security protocols, we can say that the four questions used by the STAP-Sec are insufficient to address all possible requirements to control vulnerabilities. Furthermore, the starting point of the STPA-Sec lacks the guidance to identify losses and vulnerabilities which consequently impacts the completeness of the analysis outputs. Moreover, the four questions can identify the vulnerabilities in only an abstract way without showing their types and nature.

All the above techniques can identify only known flaws and vulnerabilities, but no new ones. Furthermore, they are applicable in early development stages, where the actual design is not decided yet. Even though, developing formal notations for the output statements of these techniques can be used in the later development stage.

	AT	VIA	REA	STPA-Sec
Purpose	Analysing attacker's capabilities and goals and identifying attack's impact on a system	Analysing and eliciting requirements for security protocols	Identifying requirements that will help to avoid potential vulnerabilities in the protocol	Co-analysing safety and security aspects
Application area	Hardware and software systems related to any domain	Security protocols	Security protocols	Cyber-physical systems
Analysis method	Deductive top-down	Inductive bottom-up	Deductive top-down	Deductive top-down
Analysis documentation facility	Graphical structured tree	Textual table	Graphical structured tree	Textual table
Guidance	No explicit guidance for identifying attacker goals	Guide-words tailored for security	What-if questions based on a range of security properties	What-if questions associated with timed provision conditions
Special features	Showing a graphical model based boolean logic for attacker goals in a progressive way	Determining the deviations from the expected protocol's requirements in a gradual way starting from the high-level requirements moving to the low-level ones	Using a graphical tree to display the identified protocol vulnerabilities and requirements, and to annotate the assumptions which are made as the analysis progresses	Eliciting requirements to control the vulnerabilities rather than to protect against unknown threats

Table 3.7: Summary of security analysis techniques

3.7 Conclusion

In this chapter, we review some of the common and well-known techniques that are used for safety and security analysis. We compare their strengths and shortcomings.

This review gives rise to several observations about analysis techniques, such as:

- There is a kind of resemblance among the safety analysis techniques in terms of using the same concepts to conduct the analysis.
- Traditional safety analysis techniques with their main focus on a single component do not analyse the interactions among components or trace the failure to requirements flaws.
- Safety analysis techniques are seemingly more systematic than the security analysis techniques, and security techniques either take inspiration from the ideas of the safety techniques, or they have foundations in safety analysis techniques.
- Security analysis techniques need more extensions to be useful for security protocols.
- A major benefit from the STPA technique is taking into account the whole system behavior and the requirement flaws.
- The what-if query of STPA associated with providing conditions can be developed into what-if query associated with violation security properties to identify the security protocol requirements.

Chapter 4

Security Protocols Background

4.1 Introduction

Security protocols, also called *cryptographic protocols*, are predefined and distributed procedures making use of cryptographic operations to accomplish one or more security services, such as data secrecy, in an insecure environment. More precisely, a security protocol is a description of a communication transaction over a computer network, that involves multiple participants acting cooperatively according to the predetermined series of send and receive actions.

The security protocols can be used to provide one or more of the following services:

- **Confidentiality or Secrecy:** It means that the message content should not be read or sniffed by a deceptive participant, i.e., the sent message should remain secret and only the intended addressee is able to read it.
- **Integrity:** It denotes the ability to ensure that the message content has not been tampered or modified during the transition.
- **Authentication:** It refers to the ability to confirm the origin of the sent message and obtain proof about liveness or activeness of the communicating participants at the time the protocol is executed.
- **Non-Repudiation:** It points out an assurance that a participant is not able to repudiate having sent or received a message, and this merges authentication and integrity.

- **Anonymity:** It concerns the inaccessibility to the identities for the protocol's participants, which must be unknown.

Designing a correct and secure protocol is notorious for being difficult and error-prone task. This is due to many silent assumptions on the environment and participants' capabilities together with the presence of an intruder that can exploit subtle flaws in the design to launch its attacks. As a consequence, the researchers have been developed different analysis methods for assisting in designing a correct protocol as possible.

This chapter provides some background information on the security protocols field. It starts by presenting the key concepts of the cryptographic primitives. Then, it introduces some terms and notations which are used for describing security protocols. It also illustrates a number of external attacks upon security protocols. Next, it presents some protocol examples that have been used as case studies throughout Chapter 6. Finally, it provides an overview of the formal analysis methods and tools which have been applied to security protocols. The reviewed methods have been classified into four categories: the reasoning using logic methods that are used to reason about participants' belief and knowledge, the model checking-based methods that have been developed to use the general purpose model checking techniques for analysing security protocols, the term-rewriting rules-based methods which uses the term-rewriting rules to represent the security protocols and to search about insecure state, and the simulation-based methods which apply the simulation to discover attacks against protocols.

4.2 Cryptographic Primitives

As building and developing security protocols involve using cryptographic primitives to achieve its security-related goals, this section briefly introduces the fundamental definition of these primitives.

Cryptographic primitives are algorithms used for securing messages of online communications and providing a means of achieving protocol's services [137]. A cryptographic algorithm employs mathematical function to perform encryption, decryption, or hashing operation on sensitive messages. Generally, an encryption operation is a process of converting an ordinary text, called *plaintext*, for a message into an obscured form, called *ciphertext*. While the *decryption* is an operation of reverting back the *ciphertext* into its original *plaintext*. Both operations depend on using an additional parameter known as *key*.

A key is a piece of data that enables only the participants that hold it to carry out the above operations on a message. A *hashing* operation is an irreversible process of producing a shorter fixed-length output from the input text of undefined length.

Based on the number of keys that are used for encryption and decryption operations, the following cryptographic primitives are distinguished:

Symmetric Key Cryptography: is conventional cryptography that uses a single key for both encryption and decryption operations [237]. Usually, the key is kept secret and shared between two communicating participants. These methods can operate in either *stream mode* or *block mode*. With a stream mode, one symbol of a data is encrypted/decrypted at a time. While with a block mode, a fixed-size block of a data is encrypted/decrypted at a time. The symmetric key cryptographic methods entail that the symmetric encryption key is exchanged between the involved participants before it can be used for decryption. The well-known symmetric key algorithms are Advanced Encryption Standard (AES), Data Encryption Standard (DES), Rivest Cipher 4 (RC4), and Rivest Cipher 5 (RC5).

Public Key Cryptography: is also known as asymmetric key cryptography, where no secret key is shared between participants [90]. Typically, the public key cryptography uses a pair of keys: a public key which is publicly known to each communicating participant, and a private key which is known only to its owner. For every public key, there is a corresponding private key. In public key cryptography, the encryption process is available for anyone to encrypt messages using a known public key, while the decryption process is kept private to the owner of the corresponding private key. Some public key paradigms allow also to use the private key for encryption, and the corresponding public key for decryption. Note that, when a message is encrypted with a private key for a participant, it is said that this message is *signed* by this participant. Besides encryption and decryption services, public key cryptography can be used for providing a digital signature, message sender authentication, key agreement, and so on. The common algorithms that use public key cryptography are RivestShamirAdleman (RSA) and Digital Signature Algorithm (DSA).

Hashing Function: is also called as *message digest* or *one-way hash* function, which does not use key [170]. A hash function maps an input message of arbitrary length into the message of a fixed length, such that it is impossible to recover the content or

the length of that input. It is beneficial for detecting whether a message has been modified during the transition. This can be through sending the original message with its hash value to the receiver, which can then calculate the hash value for the original message and compare it with the received one to deduce if the message has been altered. The most common hashing algorithms are the Secure Hash Algorithms (SHA-1, SHA-256, SHA-384, etc.) and the Message Digest 5 (MD5) algorithm.

4.3 Algebraic Properties of Cryptographic Primitives

Cryptographic primitives may exhibit particular algebraic properties to reflect their behavior, such as commutativity, homomorphic, associativity, etc [83]. In this section, we will only present the commutativity property, as our work which is presented in Chapter 6 deals with this property (for further details about these properties and the protocol examples that use them see [83]).

A cryptographic primitive is called commutative if its encryption function satisfies the following property [166]:

$$\{\{m\}_{k_1}\}_{k_2} = \{\{m\}_{k_2}\}_{k_1}$$

where m is a plaintext message, k_1 and k_2 are encryption keys. This property means that the encryption operations which are accomplished using different keys produce the same results regardless of the order for the used keys.

In the commutative cryptographic, the encryption and decryption functions are sometimes the same, i.e., this cryptographic satisfies the following property:

$$[\![\{\{m\}_{k_1}\}_{k_2}]\!]_{dk_1} = [\![\{\{m\}_{k_2}\}_{k_1}]\!]_{dk_1} = \{m\}_{k2}$$

where dk_i is a decryption key that corresponds to the encryption key k_i, and $[\![\{m\}_{k_i}]\!]_{dk_i}$ denotes an decryption operation performed on the encrypted message $\{m\}_{k_i}$. Furthermore, the double encryption with this cryptographic returns the same message, i.e., it satisfies:

$$\{\{m\}_{k_1}\}_{k_1} = m$$

The common ciphers that use commutative function are one-time-pad [166], Massey-Omura [165], and Pohlig-Hellman ciphers [194].

4.4 Protocol Notation

For the sake of readability, we use the basic notation, called Common Syntax(CS) [132], that is commonly adopted in the literature to describe the security protocols [128].

A protocol is a finite sequence of message exchange steps. Each step is of the following form:

$$i.\ X_i \rightarrow Y_i : M_i$$

where $1 \leq i \leq n$, is the i^{th} step of protocol, n is the protocol steps number, X_i is the supposed sender of step i, Y_i is the supposed receiver of that step, and M_i is the message of step i. This form means M_i is sent from participant X to participant Y. X_i and Y_i could be legitimate participants with identities A, B, C, D, etc., or a trusted third-party server with identity S, S_0, S_1, etc.

A protocol message may contain one or more components, sometimes called fields. A component can be categorized into two types:

- **Unencrypted** component that may contain one of the following components:

  Identity: is an identifier representing a participant of a protocol. It is usually denoted by a capital letter, such as A, B, or S as illustrated above.

  Fresh Component: is a newly generated and non-repeating component which is employed as a proof of message freshness. A fresh component is distinguished into:

  Nonce (Number-Used-Once): : is a random number that is used in only one communication session of a protocol. It is denoted by N_X, where X is the participant who created the nonce N.

  Timestamp: is a numeric value taken from system clock to ensure accurate timeliness of a message. The notation T_X denotes a timestamp sent by X.

  Key: is a key for encryption, decryption, or performing both processes on a message based on the used cryptographic method. Accordingly, it is classified into:

  Symmetric Key : is a key, as its name suggests, used in symmetric key encryption method [237]. The symmetric key is shared between two protocol participants. In this thesis, $ssk(X, Y)$ is a key shared between X and Y.

  Asymmetric Key: is a key utilized in public key encryption method [90]. As this method uses public and private keys, we denote them by $pk(X)$ and

$prk(X)$, respectively. Where $pk(X)$ is a key belongs to X, and publicly known, and $prk(X)$ is owned and known only by X.

Hash Value: is a value produced from applying a hash function h on a message M. In this thesis, a hash function is denoted by $h(M)$.

Text: is any text component.

- **Encrypted** component, often represented as $\{M\}_k$, is a protocol message or sub message encrypted using the key k, which is either $ssk(X, Y)$, $pk(X)$, or $prk(X)$.

In the protocol specification, the participant identities and the message components are represented as variables. A *single session* of a protocol is an instantiation of these variables to generate a sequence of concrete identities and message content. By *multiple* sessions we understand several instantiated sequences (possibly interleaved) with their own identifier numbers. These numbers are used to differentiate between sessions.

A *protocol run* is a concrete instantiation for a sequence of protocol steps resulting from a single session or from a fixed number of multiple sessions. Within a protocol run, each participant plays a specific role. In our work, we consider only three roles which are *initiator*, *responder*, and *server*. An initiator is the protocol participant who initiates a protocol run. A responder is the protocol participant that the initiator intends to communicate with. A server, also called certification authority, is the participant who responsible for generating fresh key, or storing the secret keys of the participants and distributing these keys on request.

4.5 Protocol Attacks and Attack Scenarios

An *attack* on a protocol is a protocol run that satisfies an undesirable property, such as breaching the authentication, or revealing a secret. Launching an attack on a protocol is performed by a deceptive participant called intruder (or attacker) with identity I. The $I(X)$ notation means that the intruder impersonates the legitimate participant with X acting.

Usually, an attack exploits certain flaws of a protocol to be succeeded. The common protocol flaws are as follows [188, 92, 239]:

- The *liveness* of the protocol's participants is not proved. A protocol's participant is said to be alive if it has indeed participated in the run of this protocol.

- The *freshness* message component, which is either timestamp, nonce, or fresh key, is not guaranteed to be freshly generated for the current protocol run.
- The *association* of the freshness component with its generator and receiver is not clearly stated.
- The *perfect distinctness of messages* is unachievable by the protocol participants. Typically, the participants are unable to distinguish the content of messages and to associate them with their proper intended steps of the protocol. The non-distinction arises from the similarity in the type, structure, length, or content between the expected message and the one belonging to the same or other protocol.
- The *incorrect order of encryption and signature* or *insecure sending for signature*. Typically, when a signature is affixed to an encrypted message, an intruder can impersonate the signer and resigning the encrypted message with its private key. Furthermore, if a signature is sent without secrecy guarantee, it inspires an intruder to use its signature instead.
- The encryption method exhibits a particular *algebraic property* that could be easily exploited to decrypt the encrypted message.
- The *authentication and secret distribution are not interleaved*. This flaw can be noticed when a protocol is designed in such a way that a key is distributed after successful authentication between participants. In this situation, an intruder can take over the protocol session after the authentication and before the secret reaches its intended participant.
- The *use of repeated authentication*. In this case, an intruder can successfully attack the subsequent authentication without breaching the initial one.

There are many skillful attacks launched on protocols by the intruder that has complete control over the communication between the honest participants. These attacks are broadly subsumed under two categories: *passive attack* and *active attack* [222]. In the passive attack, the intruder only eavesdrops and analyses the protocol run without altering it. Conversely, the intruder that launches active attack alters the protocol run by impersonating any other participant, or by deleting, inserting, manipulating, and replaying messages. The active attack can further be subdivided into the following types:

- **Man in The Middle (MITM) Attack**: It is an attack where the intruder aims to secretly intercept, replay, alter messages between two participants and leaving them to believe that they are only communicating with each other. With this attack, an intruder resides between two participants. Often this attack occurs by exploiting one or more of the following protocol flaws: liveness, freshness, and association. A known example of this attack is upon the Needham-Schroder public key protocol which is shown in Figure 4.2b.

- **Replay (REPL) Attack.** It can be classified with respect to its goal into:

  - **Denial of Service Replay (DoS_REPL) Attack**. It aims to overwhelm the responder of the session with excessive messages by running several parallel sessions with it. Typically, these sessions start by replaying the first message intended to the responder. In this case, the honest initiator is deprived of access to its required service. Often this attack occurs by exploiting the liveness and freshness flaws. As an example of this attack, the one against the Denning-Sacco protocol [89], see Figure 4.4b.
  - **Simple Replay (Simple_REPL) Attack.** In this attack, the intruder fraudulently replays any eavesdropped message or sub-message from the old sessions. In fact, the first session is performed without impersonation, while in the next session the impersonation occurs when the discovered non-fresh message is sent. This attack is feasible when the authentication and the secret distribution are not interleaved. As an example, the Andrew Secure Remote Procedure Call protocol [205] is vulnerable to this attack [77], see Figure 4.6c.
  - **Reflection (REFL) Attack**. This attack is based on reflecting the exact overheard message (or just some parts of it) back to its original sender. In other words, when the initiator sends a message to the responder, the intruder intercepts it and re-sends it back to the original sender in another session. This attack also exploits the liveness and freshness flaws. As an example, the Three Pass protocol [214, 166] is vulnerable to this attack, see Figure 4.5b.

- **Interleaving (INTRL) Attack.** In this attack, the intruder interleaves the simultaneous executions of a protocol when it has a repeated authentication part. Typically, the intruder uses the responder as an oracle to obtain some freshness information that the intruder cannot generate by its own. The intruder makes use of

this information to counterfeit new messages that are inserted to another parallel session. The Kehne Langendorfer Schoenwalder protocol is vulnerable to an attack of this type, see Figure 4.7b.

- **Type Flaw Attack:** In type flaw attack, the intruder manipulates the type of a message in a way that causes the receiver to misinterpret the intended type. The receiver accepts the manipulated message as valid and fails to discover the type mismatch. This attack exploits the lack of proper message type checking by the receiver. As an example, the Andrew Secure Remote Procedure Call protocol [205] is vulnerable to this attack [132], see Figure 4.6b.

An *attack scenario* is an abstracted protocol run where the message content is abstracted away. In fact, a scenario does not specify the message content, except the repetition case of previously sent messages which can be identified [129]. Attack scenarios are specified according to known attacks by considering their related pattern of arranging the protocol steps for the involved sessions. A scenario has predefined information about assigning the participant identities and the intruder impersonations to each participant's role, see Figure 4.1.

$$
\begin{aligned}
&1.1\ A \rightarrow I &&: M_1\\
&2.1\ I(A) \rightarrow B &&: M_1\\
&2.2\ B \rightarrow I(A) &&: M_2\\
&1.2\ I \rightarrow A &&: M_2\\
&1.3\ A \rightarrow I &&: M_3\\
&2.3\ I(A) \rightarrow B &&: M_3
\end{aligned}
$$

Figure 4.1: An example of MITM attack scenario

Figure 4.1 presents an example of Man In The Middle (MITM) attack scenario. The left column of this figure shows a pattern of ordering the protocol steps for the involved sessions in which an assignment of the identities and intruder impersonation to each participant's role has been defined. While the right column manifests a metavariable M_i, which has a value specified by the intruder or the honest participants.

4.6 Protocol Examples

This section introduces different protocol examples which have different goals.

4.6.1 Needham-Schroder Public Key Protocol

The Needham-Schroder Public key (NSPK) protocol [174] is a classic example of the authentication protocols, shown in Figure 4.2a. It uses public key encryption approach [90]. NSPK protocol is intended to achieve mutual authentication between an initiator A and a responder B.

$$
\begin{aligned}
&1.\ A \rightarrow B : \{A, N_A\}_{pk(B)} \\
&2.\ B \rightarrow A : \{N_A, N_B\}_{pk(A)} \\
&3.\ A \rightarrow B : \{N_B\}_{pk(B)}
\end{aligned}
$$

(a) The NS protocol's messages

$$
\begin{aligned}
&1.1\ A \rightarrow I &&: \{A, N_A\}_{pk(I)} \\
&2.1\ I(A) \rightarrow B &&: \{A, N_A\}_{pk(B)} \\
&2.2\ B \rightarrow I(A) &&: \{N_A, N_B\}_{pk(A)} \\
&1.2\ I \rightarrow A &&: \{N_A, N_B\}_{pk(A)} \\
&1.3\ A \rightarrow I &&: \{N_B\}_{pk(I)} \\
&2.3\ I(A) \rightarrow B &&: \{N_B\}_{pk(B)}
\end{aligned}
$$

(b) The MITM attack on NSPK protocol

Figure 4.2: The NSPK protocol example

In NSPK protocol, A sends the first message to B which involves a nonce N_A and its identity A, together encrypted with B's public key. Upon receiving this message, B decrypts it, gets the nonce N_A, and generates it's nonce N_B. Next, B confirms that it is the one who decrypts A's message, by sending to A the concatenation of the two nonces encrypted using A's public key. A's decrypts B's message when the message is received and verifies it contains the right nonce. Then, in order to prove that it is the true participant who decrypts B's message, it sends the B's nonce back encrypted using B's public key. At the end of this protocol, both A and B share the secret nonces N_A and N_B, and both participants believe that they only know about the secret nonces. Unfortunately, this is not a correct assumption, as the protocol is subject to the MITM attack reported in [156], which occurs when the initiator communicates with a dishonest participant that exploits the absence of the B's identity in the second message. The attack is shown in Figure 4.2b

4.6.2 Needham-Schroder Symmetric Key Protocol

The Needham-Schroder Symmetric Key (NSSK) protocol [174], is an authentication protocol that is based on the symmetric key encryption approach [237]. It is designed to distribute a fresh session key shared between two participants A and B, with the aid of

a trusted server acting as a *certification authority*, and to authenticate the participants' liveness.

In the NSSK protocol, as shown in Figure 4.3, the participant A randomly generates a nonce N_A for the current protocol run and informs S that it wants to communicate with B. Then, S generates a fresh key $ssk(A, B)$ to be shared between A and B for this run, and sends this key to A within an encrypted message under $ssk(A, S)$, and to B within an encrypted sub message under $ssk(B, S)$. Upon receives this message, A verifies N_A, trusts S's liveness for this run, and accepts $ssk(A, B)$ as a new shared session key. Next, A forwards the B's sub message as a third message. When this message is received, B decrypts and picks the new session key $ssk(A, B)$. Later, B generates a nonce N_B, encrypts it with the gained key to show its possession of this key to A. Following that, A decrypts the encrypted none, increases the decrypted nonce by one, encrypts the increased nonce with the new session key $ssk(A, B)$, and sends the encrypted output to B. Upon receiving, B believes A is alive due to the freshness of the nonce N_B.

1. $A \rightarrow S : A,\ B,\ N_A$
2. $S \rightarrow A : \{N_A,\ B,\ ssk(A,\ B)\ ,\ \{ssk(A,\ B),\ A\}_{ssk(B,\ S)}\}_{ssk(A,\ S)}$
3. $A \rightarrow B : \{ssk(A,\ B),\ A\}_{ssk(B,\ S)}$
4. $B \rightarrow A : \{N_B\}_{ssk(A,\ B)}$
5. $A \rightarrow B : \{N_B{+}1\}_{ssk(A,\ B)}$

Figure 4.3: The NSSK protocol example

The main problem in this protocol is that B has no way of confirming the freshness of the third message. As a consequence, the intruder can replay the third message in another session and deceive B into accept the key in the replayed message as a new session key. This is possible in the case that the intruder is able to guess (by cryptanalysis) or steal the old session key $ssk(A, B)$, which is required to decrypt the fourth message and construct the fifth message for the new session.

4.6.3 Denning-Sacco Protocol

The Denning-Sacco (DS) protocol is a key distribution protocol invented for rectifying the freshness problem in the NSSK protocol [89]. It adds timestamps to handle this problem. DS protocol also reduces the number of messages for the NSSK protocol as shown in Figure

4.4a.

As you can see in Figure 4.4a, the nonces have been not used in DS protocol. Therefore, the fourth and fifth messages in NSSK protocol are deleted here. Furthermore, the timestamp, which is generated by S, is included in the second and third messages to confirm their timeliness. Despite that this protocol fixes the freshness problem of the NSSK protocol, it faces replaying problem [158]. In other words, the intruder can replay the third message and fool B into believing that A is attempting to launch another session with it. The replay attack on this protocol is shown in Figure 4.4b.

1. $A \rightarrow S : A, B$
2. $S \rightarrow A : \{B, ssk(A, B), T_S, \{ssk(A, B), A, T_S\}_{ssk(B, S)}\}_{ssk(A, S)}$
3. $A \rightarrow B : \{ssk(A, B), A, T_S\}_{ssk(B, S)}$

(a) The DS protocol's messages

1. 1 $A \rightarrow S$: A, B
1. 2 $S \rightarrow A$: $\{B, ssk(A, B), T_S, \{ssk(A, B), A, T_S\}_{ssk(B, S)}\}_{ssk(A, S)}$
1. 3 $A \rightarrow B$: $\{ssk(A, B), A, T_S\}_{ssk(B, S)}$
2. 3 $I(A) \rightarrow B$: $\{ssk(A, B), A, T_S\}_{ssk(B, S)}$
3. 3 $I(A) \rightarrow B$: $\{ssk(A, B), A, T_S\}_{ssk(B, S)}$

(b) The DoS_REPL attack on DS protocol

Figure 4.4: The DS protocol example

4.6.4 Three-Pass (TP) Protocol

The Three-Pass (TP) protocol [214, 166], shown in Figure 4.5a, is designed to transmit a secret message over an insecure network, without the need to share, or distribute the encryption keys. TP works using the commutative encryption method which satisfies the following property: $\{\{m\}_{prk(A)}\}_{prk(B)} = \{\{m\}_{prk(B)}\}_{prk(A)}$, where the encryption is order independent.

This protocol is called Three-Pass, since it consists of three exchanged messages. Firstly, A sends an encrypted message with its private key to B. Secondly, B encrypts the received message with its own key, and sends it to A. Lastly, when the second message is received, A decrypts it and sends the decryption output to B. Despite the advantage of not sharing keys, this protocol is subject to several attacks that exploit the commutativity property used by this protocol. For example, the reflection attack in Figure 4.5b exploits this

property to compromise the secret message. This attack can be prevented if A checks that it does not receive a message encrypted with its own key. However, a MITM attack with one session can be launched to circumvent the A's check, see Figure 4.5c.

$$
\begin{aligned}
&1.\ A \rightarrow B : \{m\}_{prk(A)} \\
&2.\ B \rightarrow A : \{\{m\}_{prk(A)}\}_{prk(B)} \\
&3.\ A \rightarrow B : \{m\}_{prk(B)}
\end{aligned}
$$

(a) The TP protocol's messages

$$
\begin{aligned}
&1.1\ A \rightarrow I(B) : \{m\}_{prk(A)} \\
&2.1\ I(B) \rightarrow A : \{m\}_{prk(A)} \\
&2.2\ A \rightarrow I(B) : m \\
&1.2\ I(B) \rightarrow A : text \\
&1.3\ A \rightarrow I(B) : \{text\}_{prk(A)} \\
&2.3\ I(B) \rightarrow A : \{text\}_{prk(A)}
\end{aligned}
$$

(b) The first attack on TP Protocol

$$
\begin{aligned}
&1.1\ A \rightarrow I(B) : \{m\}_{prk(A)} \\
&1.2\ I(B) \rightarrow A : \{\{m\}_{prk(A)}\}_{prk(I)} \\
&1.3\ A \rightarrow I(B) : \{m\}_{prk(I)}
\end{aligned}
$$

(c) The second attack on TP protocol

Figure 4.5: The TP protocol example

4.6.5 Andrew Secure Remote Procedure Call Protocol

The Andrew Secure Remote Procedure Call (AS_RPC) protocol [205], shown in Figure 4.6a, is a security protocol utilizing the symmetric key encryption approach [237]. It is designed to distribute a new session key $ssk(A,B)'$ between two participants A and B who already share a $ssk(A,B)$ key.

In this protocol, the first three exchanged messages are dedicated for achieving mutual authentication between two participant A and B. Where B authenticates itself to A, by increasing the A's nonce (N_A) by one and encrypting the output together with the B's nonce (N_B) by the shared key $ssk(A,B)$. In a similar way, A authenticates itself to B, but this time the B's nonce is increased. The last message is devoted to distribute a new generated key, i.e., B generates a fresh secret key $ssk(A,B)'$ and sends it to A. Notice that, the N'_B nonce is a nonce generated by B to be used in the subsequent communication.

$1.\ A \rightarrow B \quad : A, \{N_A\}_{ssk(A,\,B)}$
$2.\ B \rightarrow A \quad : \{N_A{+}1, N_B\}_{ssk(A,\,B)}$
$3.\ A \rightarrow B \quad : \{N_B{+}1\}_{ssk(A,\,B)}$
$4.\ B \rightarrow A \quad : \{ssk(A,\,B)', N_B'\}_{ssk(A,\,B)}$

(a) The AS_RPC protocol messages

$1.1\ A \rightarrow B \quad : A, \{N_A\}_{ssk(A,\,B)}$
$1.2\ B \rightarrow A \quad : \{N_A{+}1, N_B\}_{ssk(A,\,B)}$
$1.3\ A \rightarrow I(B) \quad : \{N_B{+}1\}_{ssk(A,\,B)}$
$1.4\ I(B) \rightarrow A \quad : \{N_A{+}1, N_B\}_{ssk(A,\,B)}$

(b) The type flaw attack on AS_RPC protocol

$1.1\ A \rightarrow B \quad : A, \{N_A\}_{ssk(A,\,B)}$
$1.2\ B \rightarrow A \quad : \{N_A{+}1, N_B\}_{ssk(A,\,B)}$
$1.3\ A \rightarrow B \quad : \{N_B{+}1\}_{ssk(A,\,B)}$
$1.4\ B \rightarrow A \quad : \{ssk(A,\,B)', N_B'\}_{ssk(A,\,B)}$
$2.1\ A \rightarrow B \quad : A, \{M_A\}_{ssk(A,\,B)}$
$2.2\ B \rightarrow A \quad : \{M_A{+}1, M_B\}_{ssk(A,\,B)}$
$2.3\ B \rightarrow A \quad : \{M_B{+}1\}_{ssk(A,\,B)}$
$2.4\ B \rightarrow I(A) \quad : \{ssk(A,\,B)'', M_B''\}_{ssk(A,\,B)}$
$2.4\ I(B) \rightarrow A \quad : \{ssk(A,\,B)', N_B'\}_{ssk(A,\,B)}$

(c) The Simple_REPL on AS_RPC protocol

Figure 4.6: The AS_RPC protocol example

Although this protocol is seemed to be carefully designed, it has two issues. The first one is an implementation-related issue [132, 77], while the second one is a freshness-related one [67]. More precisely, in [132], it was said that if the $N_A{+}1$ nonce and the $ssk(A,B)'$ key are both implemented using the same length bit sequences, then the protocol is vulnerable to the type flaw attack, whereby the $N_A + 1$ can be simply injected as the new session key, see Figure 4.6b. In [67], it was stated that there is no guarantee that the new shared key $ssk(A,B)'$ is fresh and it has been generated in the same session. Therefore, the intruder can simply replay the fourth message in another session, see Figure 4.6c.

4.6.6 Kehne Langendorfer Schoenwalder Protocol

The Kehne Langendorfer Schoenwalder (KSL) protocol [138], is a security protocol that was originally introduced in [138]. KSL is based on the symmetric key encryption to achieve its dual aim: establishing a new session key between two authenticated participants and allowing a repeated authentication process through a generated ticket, which contains a generalized timestamp. KSL consists of two parts and eight messages. The first part

attaining by the first five messages is the key and ticket distribution part. While the second part, which is attained by the last three messages, is the repeated authentication part. The protocol is detailed in Figure 4.7a.

1. $A \rightarrow B : N_A , A$
2. $B \rightarrow S : N_A , A , N_B , B$
3. $S \rightarrow B : \{N_B , A , ssk(A, B)\}_{ssk(B, S)} , \{N_A , B, ssk(A, B)\}_{ssk(A, S)}$
4. $B \rightarrow A : \{N_A , B , ssk(A, B)\}_{ssk(A, S)} , \{T_B , A , ssk(A, B)\}_{ssk(B, B)} , N_C, \{N_A\}_{ssk(A, B)}$
5. $A \rightarrow B : \{N_A\}_{ssk(A, B)}$

Repeated Authentication Part:

6. $A \rightarrow B : N_A^{'} , \{T_B , A , ssk(A, B)\}_{ssk(B, B)}$
7. $B \rightarrow A : N_B^{'} , \{N_A^{'}\}_{ssk(A, B)}$
8. $A \rightarrow B : \{N_B^{'}\}_{ssk(A, B)}$

(a) The KSL protocol messages

1. 6 $I(A) \rightarrow B : N_I , \{T_B , A , ssk(A, B)\}_{ssk(B, B)}$
1. 7 $B \rightarrow I(A) : N_B^{''} , \{N_I\}_{ssk(A, B)}$
2. 6 $I(A) \rightarrow B : N_B^{''} , \{T_B , A , ssk(A, B)\}_{ssk(A, B)}$
2. 7 $B \rightarrow I(A) : N_B^{'''} , \{N_B^{''}\}_{ssk(A, B)}$
1. 8 $I(A) \rightarrow B : \{N_B^{''}\}_{ssk(A, B)}$

(b) The INTRL attack on KSL protocol

Figure 4.7: The KSL protocol example

The first part of the protocol can be outlined as follows. Firstly, A asks B to communicate with it by plainly sending its identity and its nonce (N_A) in the first message. Next, B inquires about proving the identity of A, through linking A's message with its identity and N_B nonce and sending them together to S. Now, the server generates a new session key $ssk(A, B)$ and retrieves from its database the two secret key $ssk(A, S)$ and $ssk(B, S)$. Following that, the server sends message 3 to B which proves the A identity. Upon receiption of the third message, B decrypts the first part of it, checks the presence of its nonce N_B, and agrees that the key $ssk(A, B)$ is a new session key. Next, B generates a ticket which contains three components encrypted under a key $ssk(B, B)$ known only to B. These components are a generated timestamp T_B (an advocated expiration time

for the authentication), A's identity, and the new session key $ssk(A, B)$. After that, B forms message 4 and sends it to A. When the fourth message is received, A performs its verification by decrypting the first part and checking whether the session key $ssk(A, B)$ is the same as the key that has been used to encrypt the fourth part of this message. If they are the same, A trusts the session key $ssk(A, B)$, accepts the ticket and sends the fifth message as an evidence of its acceptance. At this point, the first part of the protocol is successfully finished, and the second part of it can be started if A wants to use the session key again. In fact, the second part can be repeated many times as long as the ticket is not expired.

The second part can be summarized as follows. When a ticket in the sixth message is received, B compares the time in T_B with the current time of its local clock and if they match, the ticket is accepted and a new nonce, which is N'_B, is generated. This nonce is exchanged through message 7 and 8 to mutually authenticate A and B. The second part is susceptible to the interleaving attack, see Figure 4.7b.

Not all the illustrated attacks at the mentioned protocol examples can be discovered by the reviewed methods below, while our method, described in Chapter 6, can detect them all except the type flaw attack.

4.7 Protocol Analysis Methods and Tools

In Chapter 3, we discussed two informal methods for analysing security protocols, while in this chapter, we discuss the methods for analysing security protocols in a formal way. Typically, different formal methods and tools have been adapted or developed to specify and analyse security protocols. These methods attempt to prove that a protocol specification meets its established goals. In fact, the formal methods help to raise our confidence in the security of a protocol, but they cannot utterly provide assurance of its security.

This section reviews some of these methods. All the methods reviewed here are based on the Dolev-Yao intruder model [91]. This model assumes that the intruder cannot decrypt the encrypted message except if it has the decryption key, i.e., the encryption method is perfect. In addition, it is assumed that the intruder has complete control over the communication between the participants. Due to its complete control, the intruder can carry out many actions to compromise the security of a protocol, limited only by its knowledge. For example, the intruder can generate any message based on its knowledge and it can overhear and intercept any sent message.

The reviewed methods here can generally be categorized into the following categories:

- **Reasoning Using Logic Methods:** are those employing belief and knowledge logic to reason over security protocols. With these methods, the protocol is directly specified in the logic, and the logical inference rules are used to derive assertions about the beliefs or knowledge of the participants, which are then analysed to ensure the satisfaction of protocol goals.

- **Model Checking-Based Methods:** are those using model checker for brute force exploring all the states of a system running the protocol in order to detect attacks upon it. A general *model checker* is an automatic verification technique allowing to decide whether a system model satisfies a certain property. To this end, it entails a finite-state model of the system, a temporal logic property, and a brute force exploration algorithm. This algorithm checks whether the given model satisfies the stated property or not. When the model fails to satisfy a required property, the model checker generates a faulty trace called counter-example representing a system's run for the unsatisfied property. To apply a model checker to security protocols, it takes an additional input which is a model of the intruder actions, together with a protocol model and a formal security property for that protocol. An attack upon the protocol is detected if a property is not satisfied. A satisfied property means there is no attack on the finite given model, but it may exist on the infinite one.

- **Term-Rewriting Rules-Based Methods:** are those using term-rewriting rules for expressing the security protocol, the intruder, and for searching about an insecure state. Generally, the term-rewriting rules are reduction rules applied to terms consisting of variables and constants, where the left-hand side of the rule is not a variable, and the right-hand side is either a constant or a subterm of the left-hand side. These rules do not require limiting the number of variables for terms, instead, they gradually rewrite (reduce) the initial term into a simple term that cannot be further reduced. This is very fruitful for analysing security protocols, due to allowing to start with an unbounded search space, and sometimes to reduce it to a finite one, without a need to limit the number of protocol sessions.

- **Simulation-Based methods:** are those applying simulation to uncover hidden flaws in the design of protocols and discover attacks against them. Simulation refers to the process of executing a specified design to explore the state space in a scalable

and non-exhaustive way in order to assess the correctness of the invariant conditions on that design. In other words, simulation performs partial exploration that is simulated by a planned and specified control flow part for the design under analysis. The partial exploration which is guided by the specified control flow patterns does not mean missing attacks that have been looked for by simulation. This is an advantageous way to analyse large protocols where the exhaustive verification methods are impractical to apply.

In Table 4.1, we summarize the main features of the methods and tools for analysing security protocols.

4.7.1 BAN logic

A well-known formalism for analysing authentication protocols is BAN logic, which was named according to its inventors: Burrows, Abadi, and Needham [67]. It is the initial endeavor to employ logic for reasoning about authentication protocols. In particular, the logic is used for describing the beliefs and actions of the honest participants involved in a protocol, and investigating how the evolution of these beliefs can help to uncover flaws and redundancies in that protocol. BAN logic is popular for its simplicity, ease of implementation, and operability at high level of abstraction.

The main purpose of BAN logic is to answer questions about what the exact achievement of the protocol under analysis, what the necessary assumptions needed to be assumed, and which redundant data in messages or redundant encryption operations must be eliminated.

BAN logic is a many-sorted model logic which consists of a set of constructs that is used to build the initial statements of the beliefs for the participants, and a set of five inference rules which derives further belief statements from the initial ones. The BAN logic is based on a number of assumptions, in particular, the encryption mechanism is unbreakable without the related decryption key, and the exchanged messages can be tampered and received by the intruder.

Some constructs, that are used in BAN logic, with their associated notations are: (below, M is a formula (representing a message), P, Q are participants, K is a key)

- $P \mid\equiv M$: P believes M, such that participant P may act as if M is true.
- $P \triangleleft M$: P sees M, or P has received and read a message M.

- $P \mid\sim M$: P *once said* M, or P believes that it has sent M in one of the previous sessions or in the current one.
- $P \mid\Rightarrow M$: P has control over M. The participant P should be trusted according to M. For example, it may be stated by the assumption that the participant Q believes that P has the authority to generate M.
- $\#(M)$: The message M is fresh; that is, M has not been sent at any time prior to the current protocol run.
- $P \overset{K}{\leftrightarrow} Q$: P and Q may share the key K to communicate, or the key K is generated by a trusted party for P and Q communication.

While some samples of the inference rules, also called *logical postulates*, are:

- *Message meaning rule for shared keys:* which says that if a participant, P, sees an encrypted message, M, under the key, K, and P believes that this key is a good key for communicating with Q, then it can be inferred that P believes that M was once said by Q. This is formally stated as:

$$\frac{P \mid\equiv P \overset{K}{\leftrightarrow} Q, P \triangleleft \{M\}_K}{P \mid\equiv Q \mid\sim M}$$

- *Message components rule:* which says that If a participant, P, sees a message, M, then it can be deduced that P also sees components of M. This is formulated as:

$$\frac{P \triangleleft (M, N)}{P \triangleleft M}$$

The actual analysis process of BAN logic consists of the following stages:

- **BAN Specification of Beliefs and Goals:** The informal protocol's goals and the beliefs of the participants are formalized into statements written in BAN logic notation.
- **Idealization Process:** The common protocol specification is transformed into an idealized form using BAN logic notation. The idealized form is produced by replacing protocol's messages with logical formulae that omit the message parts which are not involved in determining the beliefs of their receivers. The plaintext components of a

message are usually deleted from the idealized form, as they can contribute to giving hints of what the encrypted messages might be contained.

- **Step-wise Derivation:** The inference rules are applied after each received message to check whether all the goals are derived from the initial beliefs.

To develop a good understanding of the above stages, we will use a BAN analysis of the AS_RPC protocol mentioned in the original paper [67]. The first stage includes expressing statements about initial beliefs and goals. The initial beliefs for the AS_PRC participants in the BAN logic are:

- A and B initially share a key, $ssk(A, B)$:

$$A \mid\equiv A \xleftrightarrow{ssk(A,B)} B$$
$$B \mid\equiv A \xleftrightarrow{ssk(A,B)} B$$

- A trusts B to generate a good communication key, $ssk^{'}(A, B)$:

$$A \mid\equiv B \mid\Rightarrow A \xleftrightarrow{ssk(A,B)} B$$

- A and B gnerate fresh nonces:

$$A \mid\equiv \#(N_A)$$
$$B \mid\equiv \#(N_B)$$
$$B \mid\equiv \#(N_B^{'})$$

- B generates a fresh key, $ssk^{'}(A, B)$:

$$B \mid\equiv A \xleftrightarrow{ssk^{'}(A,B)} B$$

The goals for AS_PRC in BAN, which are generating a secret key by B which is known only to A and B, and both of these participants are willing to use this key, can be stated

as:

$$B \mid\equiv A \xleftrightarrow{ssk^{'}(A,B)} B$$

$$B \mid\equiv A \mid\equiv A \xleftrightarrow{ssk^{'}(A,B)} B$$

In the second stage, the AS_RPC protocol is converted into the idealized version, see Figure 4.8. Notice that, in the idealized form in Figure 4.8, the numerical increments in N_A and N_B nonces are deleted, due to their little significance for the subsequent use of the fresh session key.

$$\begin{array}{l} 1.A \rightarrow B : A, \{N_A\}_{ssk(A,B)} \\ 2.B \rightarrow A : \{N_A + 1, N_B\}_{ssk(A,B)} \\ 3.A \rightarrow B : \{N_B + 1\}_{ssk(A,B)} \\ 4.B \rightarrow A : \{ssk^{'}(A,B), N_B^{'}\}_{ssk(A,B)} \end{array} \Longrightarrow \begin{array}{l} 1.A \rightarrow B : \{N_A\}_{ssk(A,B)} \\ 2.B \rightarrow A : \{N_A, N_B\}_{ssk(A,B)} \\ 3.A \rightarrow B : \{N_B\}_{ssk(A,B)} \\ 4.B \rightarrow A : \{A \xleftrightarrow{ssk^{'}(A,B)} B, N_B^{'}\}_{ssk(A,B)} \end{array}$$

The original AS_RPC protocl ***The idealized AS_RPC protocl***

Figure 4.8: The Application of Idealization Process to AS_RPC Protocol

The last stage implies the application of the inference rules to each message. For brevity, we show the application of these rules to only the fourth message of the idealized AS_RPC protocol (Figure 4.8). This message can be stated as:

$$A \triangleleft \{A \xleftrightarrow{ssk^{'}(A,B)} B, N_B^{'}\}_{ssk(A,B)}$$

As we have the assumption:

$$A \mid\equiv A \xleftrightarrow{ssk(A,B)} B$$

the message meaning rule for shared keys can be applied to yield the following:

$$A \mid\equiv B \mid\sim (A \xleftrightarrow{ssk^{'}(A,B)} B, N_B^{'})$$

Now by applying the message components rule to the above statement, we can add the

following statements to the initial beliefs set for AS_RPC protocol:

$$A \mid\equiv B \mid\sim A \xleftrightarrow{ssk'(A,B)} B$$
$$A \mid\equiv B \mid\sim N'_B$$

At this point, no further rule can be applied, and according to the output, the goal: $B \mid\equiv A \mid\equiv A \xleftrightarrow{ssk'(A,B)} B$, can not be obtained. The reason for that is no information available in the fourth message that leads A to believe that the $ssk'(A, B)$ key is fresh.

In [67], it has been concluded that there is a freshness flaw in the AS_RPC protocol. This means that there is a possibility of replaying the fourth message in an old session as a new session message in which the session key is compromised. As a result, the AS_RPC protocol has been revised in [67], such that the nonce is used in the last message and the encryption for the first and fourth messages is deleted. However, the revised BAN version of the protocol was analysed by the work in [157], and it was shown that the revised protocol is subject to the REFL attack, see Fig. 4.9b.

1. $A \rightarrow B : A,\ N_A$
2. $B \rightarrow A : \{N_A,\ ssk'(A,\ B)\}_{ssk(A,\ B)}$
3. $A \rightarrow B : \{N_A\}_{ssk'(A,\ B)}$
4. $B \rightarrow A : N'_B$

(a) The BAN's revised AS_RPC protocol

1. 1 $A \rightarrow I(B) : A, N_A$
2. 1 $I(B) \rightarrow A : B, N_A$
2. 2 $A \rightarrow I(B) : \{N_A,\ ssk'(A,\ B)\}_{ssk(A,\ B)}$
1. 2 $I(B) \rightarrow A : \{N_A,\ ssk'(A,\ B)\}_{ssk(A,\ B)}$
1. 3 $A \rightarrow I(B) : \{N_A\}_{ssk'(A,\ B)}$
2. 3 $I(B) \rightarrow A : \{N_A\}_{ssk'(A,\ B)}$
1. 4 $I(B) \rightarrow A : N_I$
2. 4 $A \rightarrow I(B) : N_B$

(b) The REFL attack on BAN's revised AS_RPC protocol

Figure 4.9: The BAN's revised AS_RPC protocol and its attack

Despite of its ability to locate flaws in protocols, BAN logic has been widely criticized. Essentially, the idealization process was criticized for its informal implementation. In other words, the idealization process has been verbally demonstrated in [67] depending on a number of examples; without giving formal rules to follow. As a result, the idealized statements obtained from implementing the idealization process to the original protocol

specifications may be stated in many different ways, with the possibility of not conforming to the original specifications [164].

Another informal aspect in the BAN logic is the selection of the initial set of beliefs and assumptions. It has been noted in [164] that the final analysis conclusion is based on this set, and there is no way in BAN logic to show that this set is necessary or adequate to achieving the analysis.

In addition, BAN logic is criticized for its inability to handle the following: 1) modeling a message or part of a message that should not be known by a participant [175]; 2) detecting attacks where the participant is fooled by an intruder into decrypting or encrypting messages which are sent to another participant [175]; 3) differentiating between possession and belief in a message, as it is assumed that the participants believe that every message sent by them is authentic [113]; 4) formally distinguishing between a good session key required for secure communication between participants, and a good session key required to be used timely by the participants, e.g., in the current protocol run [163].

The limitations of BAN logic open the door to accommodate other logics, which either attempt to overcome these limitations or to extend BAN logic. However, these logics are notorious for their complexity and difficulty to apply.

4.7.2 GNY logic

The Gong, Needham, and Yahalom (GNY) logic is an extended and improved version of the BAN logic [113]. Thus, GNY logic introduces many features over BAN logic, that enable to analyze a wider collection of protocols, including the ones employ one-way hash function. This logic adds new rules and constructs which facilitate its generalization.

GNY logic eliminates some assumptions presumed by BAN logic, such as GNY logic does not assume that a participant believes every received message based on the encryption key, instead it assumes that a participant believes a message that conforms with its expectation. GNY logic makes a formal and clear distinction between what a participant possesses and what a participant believes in. This allows separating the reasoning about the actual message content from the one about the participant's beliefs in it.

Despite the fact that GNY logic is more expressive and elaborated than BAN logic, it is more difficult to apply, and its rules can sometimes lead to incorrect conclusion.

4.7.3 Casper/FDR

The Casper/FDR, which is an abbreviation of Compiler for Analysis of Security Protocols/Failures Divergences Refinement, is an approach for discovering attacks automatically on security protocols [93]. It is based on using two tools: the Casper compiler [159] for security protocols, and the FDR model checker for Communicating Sequential Processes (CSP) description [201].

The process algebra CSP [121] has been used in [200] to describe and model the security protocols in a generic manner. The general description consists of three aspects: 1) a set of processes representing the participants and the intruder of a protocol; 2) a number of channels between these processes dedicated for passing messages; and 3) a specification of the desired security properties that must be met by the protocol. Verifying the satisfaction of these properties' specification is performed by feeding the CSP description into the FDR tool[1].

The FDR tool, which is dedicated to CSP, differs from other model checking tools in the way it works. Generally, instead of checking whether a system meets a property specified in a language different from the system specific language, the FDR checks *CSP refinement*. More precisely, the FDR checks whether the CSP specification for a system is a refinement to that of a property. If a check fails, then a trace is generated by this tool showing a list of events that have caused this failure. In the case of the security protocol, the trace shows a list of actions performed by the intruder.

As the manual construction of the CSP description is a laborious and fallible task, the Casper/FDR method employs the Casper tool, which automatically translates the common protocol specification into CSP description suitable for direct checking by FDR tool. This translation is performed by using the Casper input script file. The Casper input file consists of two parts: the generic *protocol template* part and the *template instantiation* part. The *protocol template* part includes a declaration of the variables that the protocol utilizes, protocol description written in Common Syntax, and specification of the protocol requirements (desired properties). While the *template instantiation* part instantiates the former part and specifies the intruder knowledge.

The applicability of the Casper/FDR method to analyse security protocols has been proven in [93]. Typically, authors of this work analysed 49 out of 50 protocols in a library of [132], and reported that they re-detected 20 attacks of the previously detected 25 attacks

[1]FDR is a product for Formal Systems (Europe) Ltd.

of insecure protocols, detected attacks on 10 protocols known to be secure, and discovered 6 new attacks on dubious protocols, and failed to recognize attacks on the rest protocols. They stated that the analysis failed with the rest protocols due to the inability of this method to detect type flaw and repeated authentication part attacks. In addition, it is infeasible with this method to analyse large protocols that require growing storage space for the generated messages, nonces, and keys.

In general, Casper/FDR method has the ease of usage feature, but it can only analyse finite and small models and it is unable to detect type flaws and repeated authentication attacks. Furthermore, this method requires that a user has a good understanding of specifying the security requirements in Casper, to obtain correct analysis outputs.

4.7.4 SPIN

The SPIN, which stands for Simple Promela INterpreter (SPIN), is a general purpose model checking tool that automatically verifies the correctness of the design for asynchronous and distributed software systems [122].

In SPIN, the system design must be written in a modelling language known as PROMELA (PROcess MEta LAnguage), and the required properties must be specified in the LTL (Linear Temporal Logic) formula [193].

Promela is a process modeling language that allows for synchronizing and coordinating concurrent processes. A Promela model consists of processes, message channels, and variables. Processes specify a finite behavior of the system's components. Message channels are employed to transfer data between processes. Variables define the environment in which the processes are executing, and these variables must have finite values.

The LTL formula models time in a manner similar to Natural Numbers, $\mathbb{N}$. The LTL model can be described as a sequence of infinite states, where each state is a set of propositional symbols. Those symbols are satisfied at the i^{th} moment time.

SPIN verifies the fact that a given Promela model satisfies a given LTL property, by checking whether that property holds on all the execution traces of the model. In the violation case, SPIN reports a counter-example showing the erroneous execution path of the model, i.e., the path that does not match the desired property. SPIN covers safety, liveness, deadlock, and invariant correctness properties.

The idea of how to extend SPIN to consider security protocol verification is presented in [131]. The main benefit of this extension is to investigate certain protocol errors that are

undiscoverable by BAN logic method; in particular, investigating protocol manipulations which emerge from replay attacks. An extension is presented by specifying a process for every protocol participant and the intruder together with imposing limitations on the complexity of the resulted specification to avoid the state space explosion problem.

The theoretical idea was converted into a concrete one in [162], where the authors described a general methodology, illustrated on the NSPK protocol, for building a Promela model ofr the protocol and the intruder's abilities. With this method, several formal steps for building the protocol model are defined, and informal procedure for constructing the intruder model of the NSPK protocol is described. Later, several endeavours emerged to develop the intruder model into a generic one which can be used for different protocols [47, 73].

In general, SPIN is praised for being an automated verification way that produces either a 'true' answer when a property is met or a 'false' answer and diagrammatic/textual output showing a sequence of messages related to the discovered attack. Although it can verify authentication and secrecy properties for different protocols, SPIN can only discover attacks on protocols with a limited number of sessions and messages and restricted intruder model. The intruder model is restricted by only generating fixed length messages and storing limited eavesdropped messages due to its bounded storage space.

4.7.5 NuSMV

The NuSMV, abbreviated for New Symbolic Model Verifier, is a symbolic model checking for finite state systems. It supports checking properties expressed either as LTL (Linear Temporal Logic) formulas or as CTL (Computation Tree Logic) formulas. NuSMV provides the SMV modelling language which is tailored to describe finite models of systems in a modular way. The input for the NuSMV tool is a textual program that describes the SMV model and the temporal logic properties. NuSMV returns either a 'true' answer when the property is satisfied or a 'false' answer and textual output representing the execution sequence for the unsatisfied property. It has been used to analyse the security protocols in [189, 155, 21], where its ability to check secrecy and authentication properties has been shown. Unfortunately, this model checker shares with SPIN checker the same limitations of detecting attacks only with bounded settings.

4.7.6 ProVerif

ProVerif is an automatic tool for verifying cryptographic protocols [51, 54, 8]. It aims at verifying protocol runs for an unbound number of sessions and message space. This aim was accomplished by using a simple *protocol representation* and *resolution algorithm*.

The protocol representation is a set of Prolog rules. Each rule has the following form:

$$P_1 \wedge P_2 \wedge ... \wedge P_n \rightarrow P$$

where $P, P_1, .., P_n$ are predicates used to represent facts about terms (messages), and the predicates on the left-hand side of the arrow is a hypothesis of the rule while that on the right-hand side is the conclusion of the rule. If a rule has no hypothesis then it is simply written as P [51]. The set of rules represents the following:

1. The computation intruder's capabilities. An example of that would be decrypting a message when the intruder has a ciphertext for this message and the decryption key.

2. The protocol steps themselves which are written in Common Syntax. The user must add one rule for each step.

3. The facts about the initial intruder's knowledge. For example, the intruder knows the public keys for all participants.

The protocol representation for ProVerif originally stems from the *Horn theory* [53]. A horn theory is a theory in which each axiom is a Horn clause [124]. A Horn clause is a set of predicates that has disjunction or implication form.

The resolution algorithm is that used to determine whether a given fact(s) can be inferred from the representation rules or not. More precisely, the resolution algorithm applies the representation rules to construct a sequence of steps that shows how the intruder achieves the stated fact(s). During this sequence constructing, a non-termination problem arises due to the unbounded depth of the nesting terms for facts. The resolution algorithm alleviates this problem by using the unification of rule pairs that could be combined together. This is possible when the unified facts of the two rules, that are applied one after the other, are not of the form for the given fact. In that case, a new rule is generated by resolving the second rule with the first one upon the given fact. The unification can limit the search space, but can not guarantee the termination. Therefore, the algorithm enforces termination by limiting the depth of terms appearing in the rule for participants but not

for an intruder, so that each term is checked and will be replaced with a new variable if its generation exceeds this limit [51].

ProVerif analyses protocol specifications which are written in several languages. Currently, the formal input language for this tool is the applied pi calculus [54]. The applied pi calculus is an extension of pi calculus where new semantics are added to define the cryptographic primitives either as a set of rewrite rules or equations [15].

There are two inputs for ProVerif tool: a model of the protocol which is written in the applied pi calculus, and the security properties that must be met. The rules for the intruder's capabilities are built-in to ProVerif. ProVerif can verify different properties, including secrecy, correspondence assertions, and observational equivalence (only with bounded sessions number). Correspondence assertions are the properties of the form; if one participant ever executes an event in a protocol, then in the past some other participant has executed some other events in this protocol. The observational equivalence is used to prove that there two processes having the same structure, but the message content is different. These properties are specified as *queries*.

Internally, ProVerif translates the applied pi calculus model into a set of Horn clauses. It also translates the queries into derivability queries on the obtained clauses [53]. Later, ProVerif applies the resolution algorithm to examine whether a query is derivable from the clauses. ProVerif produces *true* (if the query is not derivable by the intruder) to indicate that the query is proved and there is no attack against the protocol. Otherwise, ProVerif produces *false* and an attack trace showing the intruder's actions that lead to disproving the query. However, ProVerif is not complete and does not always produce *true/false* output, as it sometimes presents *false attack*. False attach is a textual description of a situation in which the query is true, but ProVerif could not prove it. Reporting false attack is due to using approximation measure for the translation into Horn clauses [52]. The approximation means modeling the Horn clauses as a linear logic model that ignores the number of repetitions required for each clause during the application.

The attack trace which is produced by ProVerif is very detailed and verbose. Therefore, understanding this trace for protocols that have more than three steps could be a tedious task.

ProVerif can not terminate during analysing protocols in which participants initially exchange encrypted secret data and then they reveal it in later protocol steps.

ProVerif also can not terminate its analysis when the analysed protocol is based on cryptographic primitives with a certain algebraic property that requires an equivalence

relation between terms. The non-termination originates from the way of modelling this property as an *equation* formalism directed from left to right, where the right hand side is a constant or a simple term, and the left hand side is the most complex term. However, ProVerif has been extended to support the algebraic properties for Exclusive-Or and modular exponentiation operators using two different translators, namely XOR-ProVerif [140] and DH-ProVerif [139], respectively. Both of these translators translate the protocol specification, which is written in Prolog language into Horn clauses to be compatible with ProVerif. The two translators require the SWI-Prolog (version 5.6.14) tool in order to work. These tools takes long execution time when the number of the variables involved in these operators is more than eight variables.

4.7.7 Tamarin

Tamarin is a relatively new tool for the symbolic modeling and analysis of security protocols [168, 85, 9]. It is able to handle an unbounded number of sessions by considering the logic of protocol interactions. It also supports the equational specification of two cryptographic operators: the Diffie-Hellman exponentiation and the Exclusive-or.

Tamarin takes three inputs: a model of the protocol, specification of the intruder capabilities, and specification of the security properties. Based on multiset rewriting rules, Tamarin provides an expressive language for specifying its inputs.

In Tamarin, protocols are modelled as a collection of multiset rewriting rules that defines a state machine with labelled transitions. A rewrite rule is expressed as a multiset of facts. Facts are predicates used to store information about the machine's state, such as the sent and received messages, or the public key possession. A rewrite rule is defined by a specific name and three types of facts: *premises*, *actions* (or *labels*), and *conclusions*. Premises are facts for characterizing a machine's state before executing the rule. They are situated on the left-hand side of the rule. Conclusions are facts resulted after executing the rule. They are situated on the right-hand side of the rule. Action facts are used to label execution traces (transitions). Once a rule is executed at a time point, the actions are appended to that time point which then will be used as a reference to define the property trace. A transition to a new state is decided by a rule, and that is performed through replacing the instantiated conclusions of this rule with its premises if the later exist in the current state.

Concerning the intruder, Tamarin offers built-in rewriting rules for specifying all the

possible intruder's actions, such as intercepting messages, generating arbitrary messages, and eavesdropping all sent messages. While analysing a protocol using Tamarin, the instances of the protocol's rules are interleaved in parallel with those for the intruder.

Tamarin employs trace properties to specify the protocol properties. A trace of the protocol is a record of action facts defined by the applied rewrite rules in a particular execution. A trace property is a description of a set of traces. This description is defined as a guarded fragment of first-order logic which allows quantification over action facts and time points. Tamarin proves that a trace property holds for a protocol when it holds in all traces of the protocol. The main properties that can be analysed using Tanamrin are: secrecy, authentication, and observational equivalence.

Tamarin's language is more expressive than that of ProVerif in specifying security properties, as it supports direct modelling for temporal properties. However, Tamarin is seemingly slower than ProVerif on analysing secrecy and authentication properties, as its way in modelling adds complexity to the analysis procedure [141].

Tamarin analyses the given security properties in two modes: *automatic* and *interactive*. In the automatic mode, Tamarin terminates and returns either a proof showing the correctness of the stated property or a counter-example representing an attack which falsifies the given property. While in the interactive mode, Tamarin can not terminate and the user must intervene to manually inspect the proof states and the attack graphs. For example, Tamarin cannot terminate on proving the secrecy of nonce for the NSPK protocol [85].

The cryptographic primitives and their properties are modelled in Tamarin based on a fixed set of built-in function symbols and equations. This set supports modelling the following: hashing, public-key encryption, symmetric key encryption, diffie-hellman, and xor. Due to its use the built-in functions, Tamarin is not flexible to model another primitive, such as commutative encryption. Apart from the built-ins, Tamarin only supports the specification of subterm-convergent educational theories, in which the right-hand side of the equation is a simple term or a sub-term of the left-hand side. This per se leads to undecidable verification problem when a more complex property, which requires equivalence relation between its terms, is analysed.

4.7.8 Simulation Based Attack Scenarios

The idea of attack scenarios, described in Section 4.5, is introduced in [129] with the aim of making the protocol verification process decidable. This is achieved by simulating the security protocols using scenarios of known attacks. The attack scenarios allow reducing the number of explored runs of a protocol while looking for an attack by simulation.

In [129], five schemes of the origin and destination attacks from [224], including MITM, REFL, DoS_REPL, Simple_REPL, and INTRL attacks, had been defined. For each of the above schemes, a generic algorithm is designed, which when given parameters of the protocol, such as its steps number, sessions number, and other parameters particular for a type of attack, it will produce a scenario of an attack for this scheme. The algorithms also need to define an assignment of the participant's identity together with the intruder's impersonation to each role. In the beginning, the produced scenario is written in Common Syntax [128] and then it is translated into an Estelle specification [65], to return the concrete simulated scenario. The honest participants and the intruder construct the messages in a scenario.

The practical application of the simulation based attack scenarios method has been shown for several protocols. However, the intruder in this work composes messages using only type matching restriction, which can still increase the message space with unacceptable messages. In addition, this work does not show how to analyse protocols that make use of certain algebraic properties and it only checks the correspondence assertion property. Furthermore, in this work, the designed scenarios are all simulated even those whose simulation is definitely not leading to produce attacks. Accordingly, we aim to adopt this method while addressing these limitations.

4.8 Conclusion

Security protocols are key elements of secure communications. However, there are often subtle flaws and vulnerabilities in their design attract malicious intruder to breach the security objectives of the protocols. In this chapter, we introduced the foundations of security protocols and details some known flaws and attacks on such protocols. Furthermore, we reviewed some methods used for security analysing the protocols. In particular, we recapitulated the methods that use the beliefs and knowledge logic, the model checker, the term-rewriting rules, and the simulation.

We have deduced the following strengths and drawbacks of these methods. The reasoning using logic methods are useful to reason about the beliefs of the participants involved in protocols and to locate some flaws in their designs. Unfortunately, BAN logic was criticized for the informality associated with its operation semantics, and the ambiguity associated with its idealisation process step. Model checking methods find attacks on protocols by exhaustively search all the reachable states and checking that the undesired property is unreachable. However, these methods entail a finite model for a protocol under analysis and bounded instances of its execution in order to work successfully. In the term-rewriting rules-based methods, an unbounded analysis for protocols are achieved, but at the cost of the non-termination and undecidability problems, which are emerged at analysing some protocols and algebraic properties. Simulation-based methods help to reduce the number of the explored runs while searching for attacks by means of simulation. Our work pays particular attention to these methods as they guarantee the termination at the analysis time and they do not miss attacks that their specified scenarios are given to these methods.

Methods/ Tools	Methodology	Intruder Model	Security Properties	Large Protocols Applicability	Guaranteed Termination	Bounded Sessions Number	Bounded Message Space
BAN Logic	Employing logic developed to analyse knowledge and beliefs	None	None	Difficult	Yes	No	No
GNY Logic	Employing logic developed to analyse knowledge and beliefs	None	None	Difficult	Yes	No	No
Casper/FDR	Exhaustively checking whether a protocol model meets a desired property	Only the intruder's knowledge needs to be added	Casper specifications	Impossible	Yes	Yes	Yes
SPIN	Exhaustively checking whether a protocol model meets a desired propert	Needs to be added	LTL processes	Impossible	Yes	Yes	Yes
NuSMV	Exhaustively checking whether a protocol model meets a desired propert	Needs to be added	LTL/CTL specifications	Impossible	Yes	Yes	Yes
ProVerif	Analysing unbounded number of protocol's runs based on employing term-rewriting rules	Built-in	Querries	Possible	No	No	Approximated
Tamarin	Analysing unbounded number of protocol's runs based on employing term-rewriting rules	Built-in	First order trace properties	Possible	No	No	No
Simulation Based Attack Scenarios	Simulating the designed scenarios for known attacks	Needs to be added	Invariant conditions	Possible	Yes	Yes, but without missing attacks	No

Table 4.1: Summary of protocol analysis methods and tools

Chapter 5

Safe Design Development Methodology for Safety-Critical Systems

5.1 Introduction

This chapter presents a systematic methodology for developing safe designs of safety-critical systems by combining the Abstract State Machine (ASM) method with the System-Theoretic Process Analysis (STPA) technique. The main aims of this methodology are developing safe ASM specifications together with adequate and concise temporal formalizations of the STPA requirements.

As mentioned in Chapter 2, the ASM method has been used to model and analyse safety-critical systems in [33, 30, 35]. In these works, it has been shown how the ASM method, within the open source ASMETA framework, supports different development activities, including designing, simulating, validating, and verifying the ASM model. The verification activity in these works focuses only on proving the functional requirements to produce a correct model. Nevertheless, the correctness of the model is not enough to ensure its safety. As a result, the verification activity must be guided by the safety requirements alongside the functional requirements during the development process. The safety requirements can be elicited by applying an appropriate safety analysis technique.

As illustrated in Chapter 3, the STPA technique is able to identify a wide range of safety

requirements emerging from unsafe component interactions and inadequate control in the systems design. Recently, the STPA technique has been used, in [17, 19], as an integrated tool with the verification activity, by supplying it with the formulated safety requirements. However, the formalization process of safety requirements is both cumbersome and not accurately capturing some of the temporal aspects of the requirements.

For these reasons, we present a methodology for developing correct and safe critical systems, that is based on the ASM method, the STPA technique, and temporal logic. It starts with modelling the system in the AsmetaL and simulating it by AsmetaS to obtain an accurate mathematical representation. Then, using the AsmetaV validation tool, the model validation process is applied to ensure that it meets the functional requirements. Next, the STPA technique is utilized to elicit safety requirements. These requirements are formalized into Linear Temporal Logic (LTL) that can be verified against an AsmetaL model using the AsmetaSMV verification tool.

We apply this methodology to two case studies which are the Train Door Controller (TDC) [227] and the Insulin Pump Control System (IPCS) [220].

The main contributions of this chapter are:

1) a systematic methodology for developing safety critical systems through combining ASM with STPA, with the target of developing safe specifications; and

2) adequate and concise temporal formalizations of the STPA requirements.

The rest of the chapter is organized as follows: 5.2 details our methodology. Section 5.3 presents the application of our methodology to the TDC case study, while Section 5.4 illustrates this application to the IPCS case study. Then in Section 5.6, we evaluate our methodology. Section 5.7 reviews the related work. Section 5.8 concludes the chapter.

5.2 The Proposed Methodology for Developing Safe Design

This section presents the ASM-STPA-SA methodology that combines the ASM method with the STPA technique to develop safe designs for systems. This methodology is based on some of ASMETA tools, STPA, and temporal logic. The ASMETA tools that have been utilized in the ASM-STPA-SA methodology are as follows: (1) The AsmetaS tool [112], which executes ASM models that are written in AsmetaL. (2) The AsmetaV tool which validates AsmetaL specifications by scenarios written in Avalla [69]. (3) The AsmetaSMV

model checker tool which formally verifies the AsmetaL specifications [31]. Note that, the AsmetaSMV can be used to verify both functional and safety requirements, but in our work which focuses on verifying safety requirements, we use only it to verify these requirements.

Using this methodology, we aim to guide the modeller to improve the ASM model, depending on the detection of any violations to the functional and safety requirements via the validation and verification tools, and to provide the verification tool with the STPA requirements in a formal way.

Figure 5.1 shows an overview of our methodology which includes the following stages:

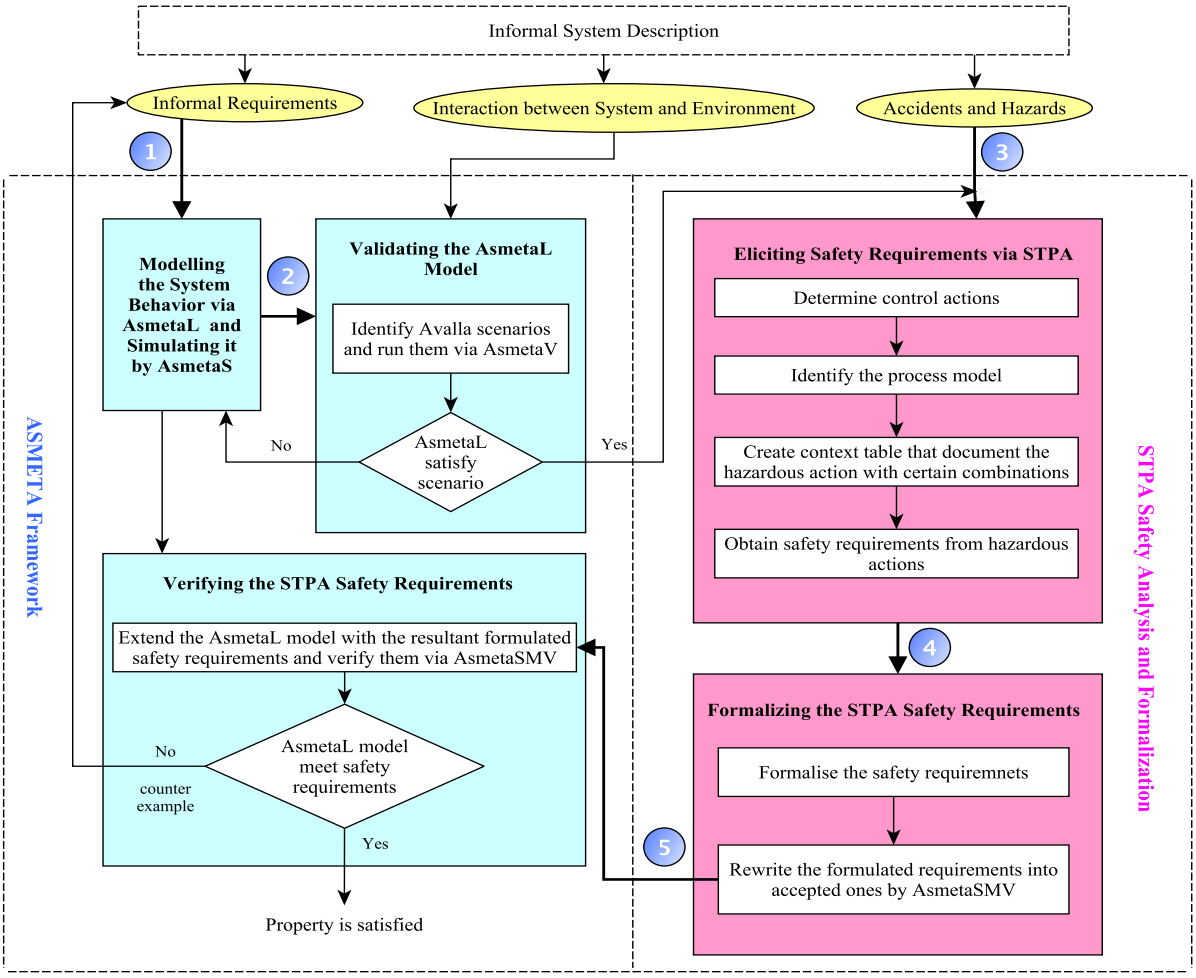

Figure 5.1: The safe design development methodology

(1) **Modelling the System Behavior via AsmetaL and Simulating the Resultant Model:** Through this stage, we aimed at modelling the system under analysis in AsmetaL specification in order to provide the formal foundation for our methodology. In addition, we intended to simulate the AsmetaL specification by the AsmetaS tool. The reasons for that are to check whether there is a state in which incorrect rule may be executed and to gain enough confidence that the AsmetaL captures the desired informal requirement.

(2) **Validating the AsmetaL Model:** This stage attempts to validate that the AsmetaL model satisfies the functional requirements, which relate to the user needs about the system by using the AsmetaV tool. This tool allows constructing particular scenarios that describe the interactions between the system and its environment to represent informal functional requirements. Avalla has several constructs to write the scenario, such as `set`, `step`, `check`, etc. The AsmetaV tool reads a scenario written by the user in Avalla and invokes the AsmetaS tool to simulate this scenario and checks if the AsmetaL model satisfies this scenario or not. In the event that any of these scenarios are not satisfied, the AsmetaL model must be modified. The AsmetaV tool and its associated language have already been described in Chapter 2; Section 2.4.9.

(3) **Eliciting safety requirements for the system via STPA:** This stage involves eliciting the informal safety requirements by applying the STPA procedure, which consists of the following steps:

- Indicating the main expected accidents for the system.
- Identifying the possible hazards that can lead to accidents in the previous step.
- Drawing the schematic safety control loop which describes the interactions between the software controller and other system's components, such as hardware, software, or human factors.
- Using the control loop as a guide to determine the main control actions issued by the controllers that can lead to the identified hazards.
- Identifying the process model for the controller. We define it as a set of monitored and controlled functions of the AsmetaL model, such that this set affects providing the control actions. Each member in this set consists of the function

name and its values. As mentioned earlier in Chapter 3, the functions of the process model for a controller represent the environmental and system states that affect the safety of the control actions. These functions can be derived from the identified system hazards. For instance, from the hazard which is *a door opens when the train moving or stopped unaligned to a platform*, we can deduce that the train motion and position are ones of the monitored functions that must be considered in the process model because they affect the safety of open the door action.

- Evaluating the combination of the function values for each control action under four contexts: 'Provided', 'Provided too early', 'Provided too late' and 'Not Provided'. The evaluation process is performed through asking a question to an expert in the following form: if the controller receives a certain combination of function values, will (provide, provide too early/too late, not provide) the action in the next state by the controller lead to a hazard or not. The results of the evaluation are documented in the context table.

- Translate each combination that has yes answer in the table into informal safety requirements using the phrases *must* (for 'Not Provided') and *must not* (for 'Provided', 'Provided too early', 'Provided too late'.

(4) **Formalizing the Elicited STPA Safety Requirements:** In this stage, the informal STPA requirements are formulated into LTL specifications using the following steps:

- Identify the main function combinations that have yes answers in only the 'Not Provided' condition, in order to reduce the number of the requirements but without ignoring the safety issue. Typically, the function combinations that have yes answers in the 'Not Provided' condition are translated into the requirements for providing an action, hence, by ensuring that the action always satisfies these requirements not too early, on time, and not too late, then there is no need to check that the action is not provided with the combinations in the other conditions. Thanks to the if and only if operator that helps us to ensure that, see the next point.

- Use the identified combinations to write the following formula:

$$\square((com_{i1} \vee com_{i2} \vee ...com_{in}) \leftrightarrow \bigcirc(CA_i)) \tag{5.1}$$

where: CA_i is the i^{th} control action, com_{in} is the nth combination that relates to the ith action, and it has yes answer in the 'Not Provided' condition. Each combination is connected by $\vee$ operator. The formula informally means, that the control action is always provided in the next state, if and only if, one of the determined combinations occurs. The $\leftrightarrow$ operator puts a strong condition on providing the action, i.e., the action will not be provided with another combination or later/earlier than satisfying the determined combination. Furthermore, employing the $\leftrightarrow$ and $\vee$ operators help to reduce the number of properties to be verified.

- Rewrite the formulated requirements, which are based on 5.1, into other ones using the syntax of the AsmetaSMV tool, i.e., the syntax of its propositional and future-time connectives.

(5) **Verifying the Formulated Safety Requirements:** This stage is intended to verify that the AsmetaL model satisfies the formulated STPA safety requirements. If any of these requirements is not satisfied, then a counter-example will guide the modeller to improve the AsmetaL model.

5.3 Case Study: The Train Door Controller System

In this section, we show how to apply the ASM-STPA-SA method to the Train Door Controller (TDC) case study. This case study, presented in Section 2.3.1, has been used in [227] as a simple illustrative example to introduce the analysis procedure of the STPA technique. While in our work, we focus on feeding the AsmetaSMV model checker with the STPA analysis results to verify them and to build a safe design for this example. In the following, we will describe the detailed steps and the results of applying the ASM-STPA-SA method to the TDC system.

5.3.1 Modelling the system Behavior via AsmetaL and Simulating the Resultant Model

The complete model for the TDC system is shown in Code 2.1 which is presented in Chapter 2. In which, we specify changing the door state using several rules and also we specify the main rule that controls the whole system behavior. As there are two states for the TDC: SENSING and EXECUTING, the main rule starts by checking these states. When the state is SENSING, then in parallel, the controller receives information from the sensors, and the state is changed into EXECUTING. While when the state is EXECUTING, then firstly the emergency is checked whether it exists, to open the door immediately, but if it does not exist, then the rulers for changing the door state are fired in parallel, see Code 5.1a.

```
main rule r_Main =
 if state=SENSING then
   par
     r_EmergencySensing[]
     r_TrainMotionSensing[]
     r_ObstacleSensing[]
     state:=EXECUTING
   endpar
 else
   if emergency=EXIST then
     r_HandleEmergency[]
   else
     par
       r_closed_to_opening[]
       r_opening_to_opened[]
       r_opened_to_closing[]
       r_closing_to_closed_or_opened[]
       state:=SENSING //Only here the
         //sate is changed to sensing
     endpar
   endif
 endif
```

(a) The wrong specification

```
main rule r_Main =
 if state=SENSING then
   par
     r_EmergencySensing[]
     r_TrainMotionSensing[]
     r_ObstacleSensing[]
     state:=EXECUTING
   endpar
 else
   par
     if emergency=EXIST then
       r_HandleEmergency[]
     else
       par
         r_closed_to_opening[]
         r_opening_to_opened[]
         r_opened_to_closing[]
         r_closing_to_closed_or_opened[]
       endpar
     endif
     state:=SENSING
   endpar
 endif
```

(b) The correct specification

Code 5.1: The main rule for TDC example

The simulation of the TDC which is produced by running the AsmetaS tool shows that when there is an emergency, the system work is terminated without checking whether the emergency has been handled to resume the system to its usual work, see Figure 5.2.

The notations used in Figure 5.2 are: the *question mark* indicates that the value of the monitored function must be entered by a user, and the *hyphen* sign denotes that the user has not been asked for the value of the monitored function. In Figure 5.2, at state 0, the user is asked to enter the value of the emergencySensor function which is EXIST. Following that, the simulator calculates the update set to produce state 1. Since the value

of the emergency is EXIST in this state, the simulator updates the value of the doorStatus to opened in order to produce state 2. After that, the simulator finds the state 3 has a similar update sets to those in state 2, therefore it terminates the run. This happened because, at the specification for the main rule in 5.1a, the system state is not changed into SENSING state and it remains at EXECUTING state when the emergency exists. This gave us an indication to correct the main rule in Code 5.1a, such that the state function is updated to SENSING even when there is an emergency, see Code 5.1b, in order to ensure that the system can resume its normal work when there is no emergency.

State		0	1	2	FINAL STATE 3
Controlled Functions	doorStatus	CLOSED	CLOSED	OPENED	OPENED
	trainMotion	MOVING	STOPPED	STOPPED	STOPPED
	trainPosition	NOTALIGNED	ALIGNED	ALIGNED	ALIGNED
	obstacleStatus	NOTEXIST	NOTEXIST	NOTEXIST	NOTEXIST
	emergency	NOTEXIST	EXIST	EXIST	EXIST
	state	SENSING	EXECUTING	EXECUTING	EXECUTING
Monitored Functions	emergencySensor?	EXIST	-	-	-
	trainMotionSensor?	STOPPED	-	-	-
	trainPositionSensor?	ALIGNED	-	-	-
	obstacleSensor?	NOTEXIST	-	-	-

Figure 5.2: Part of the simulation for the TDC model with incorrect main rule

5.3.2 Validating the AsmetaL Model

We have then built a scenario corresponding to the informal requirement of checking that the door state is changed from CLOSING to OPENED when there is an obstacle. The scenario is written in Avalla to be an input to the AsmetaL, see Code 5.2. It permits to drive the execution of a model in a way which facilitates producing a run that is expected to satisfy the desired requirement. The scenario in Code 5.2 can be described as follows: the emergency, train motion, and train position sensors must be set into the following values (before moving into a new step): NOTEXIST, STOPPED, and ALIGNED, respectively; after that several simulation steps must be made until reaching the state in which the train is stopped and it is aligned with the platform. At this state, the door state is checked whether it is OPENING. Following that, the system will be at the SENSING state such that the EMERGENCYSENSOR is set to NOTEXIST. Next, the door state is changed to OPENED. When there is no emergency and obstacle, the door state is updated to CLOSING. At that state, when the sensor of an obstacle returns EXIST value, then the door state is checked whether it is OPENED.

```
scenario CheckOpenStateAfterObstacle

load ./TrainDoorController.asm

set emergencySensor:= NOTEXIST;
set trainMotionSensor:=STOPPED;
set trainPositionSensor:=ALIGNED;
set obstacleSensor :=NOTEXIST;

step until state=SENSING and trainMotion=STOPPED and
                             trainPositionSensor=ALIGNED;

check doorStatus=OPENING;

set emergencySensor:= NOTEXIST;
step
check state=EXECUTING;
step
check doorStatus=OPENED;

set emergencySensor:= NOTEXIST;
set obstacleSensor :=NOTEXIST;
step
check state=EXECUTING;
step
check doorStatus=CLOSING;


set emergencySensor:= NOTEXIST;
set obstacleSensor :=EXIST;
step
check state=EXECUTING;
step
check doorStatus=OPENED;
```

Code 5.2: The scenario which corresponds to the requirement that entails the door is opened when there is an obstacle

State		0	1	2	3
Controlled Functions	doorStatus	CLOSED	CLOSED	OPENING	OPENING
	trainMotion	MOVING	STOPPED	STOPPED	STOPPED
	trainPosition	NOTALIGNED	ALIGNED	ALIGNED	ALIGNED
	obstacleStatus	NOTEXIST	NOTEXIST	NOTEXIST	NOTEXIST
	emergency	NOTEXIST	NOTEXIST	NOTEXIST	NOTEXIST
	state	SENSING	EXECUTING	SENSING	EXECUTING
Monitored Functions	emergencySensor?	NOTEXIST	-	NOTEXIST	-
	trainMotionSensor?	STOPPED	-	-	-
	trainPositionSensor?	ALIGNED	-	-	-
	obstacleSensor?	NOTEXIST	-	-	-
Verdict				CHECK SUCCEEDED: doorStatus=OPENING	
State		**4**	**5**	**6**	**7**
Controlled Functions	doorStatus	OPENED	OPENED	CLOSING	OPENED
	trainMotion	STOPPED	STOPPED	STOPPED	STOPPED
	trainPosition	ALIGNED	ALIGNED	ALIGNED	ALIGNED
	obstacleStatus	NOTEXIST	NOTEXIST	NOTEXIST	NOTEXIST
	emergency	NOTEXIST	NOTEXIST	NOTEXIST	EXIST
	state	SENSING	EXECUTING	SENSING	EXECUTING
Monitored Functions	emergencySensor?	NOTEXIST	-	NOTEXIST	-
	trainMotionSensor?	-	-	-	-
	trainPositionSensor?	-	-	-	-
	obstacleSensor?	NOTEXIST	-	EXIST	-
Verdict		CHECK SUCCEEDED: doorStatus=OPENED		CHECK SUCCEEDED: doorStatus=CLOSING	CHECK SUCCEEDED: doorStatus=OPENED

Figure 5.3: Simulation of the scenario in Code 5.2

```
main rule r_Main =
 if state=SENSING then
   par
     r_EmergencySensing[]
     r_TrainMotionSensing[]
     r_ObstacleSensing[]
     r_cancel_action[]//This rule must be
                      //deleted
     state:=EXECUTING
   endpar
 else
   if emergency=EXIST then
      r_HandleEmergency[]
   else
     par
        r_closed_to_opening[]
        r_opening_to_opened[]
        r_opened_to_closing[]
        r_closing_to_closed_or_opened[]
        state:=SENSING
     endpar
   endif
 endif
```

(a) The wrong specification

```
main rule r_Main =
 if state=SENSING then
   par
     r_EmergencySensing[]
     r_TrainMotionSensing[]
     r_ObstacleSensing[]
     state:=EXECUTING
   endpar
 else
   if emergency=EXIST then
      r_HandleEmergency[]
   else
     par
        r_closed_to_opening[]
        r_opening_to_opened[]
        r_opened_to_closing[]
        r_closing_to_closed_or_opened[]
        state:=SENSING
     endpar
   endif
 endif
```

(b) The correct specification

Code 5.3: The r_Main rule for TDC example

The simulation of this scenario returns a successful verdict, see Figure 5.3, since at state 6 in Figure 5.3, the door state is CLOSING and there is an obstacle at the doorway, therefore the door state is changed to OPENED and the validation is succeeded.

5.3.3 Eliciting safety requirements for the system via STPA

Following the validation, we elicit the safety requirements by applying the STPA procedure which implies the following steps:

- **Identifying the main accidents**. The system has the following accidents:
  - A closing door traps a person.
  - A person is enclosed in the train during an emergency.
  - A person is injured by falling out of a train.
- **Indicating the main hazards related to the identified accidents**. The system has the following hazards:
  - A door closes on a person while (s)he is still in the doorway.
  - A person is unable to leave the train during emergency.

  - A door opens when the train is moving or stopped unaligned to a platform.
  - A door does not open when it should.

- **Drawing the schematic safety control loop**. Figure 5.4 shows the safety control loop diagram for the TDC case study.

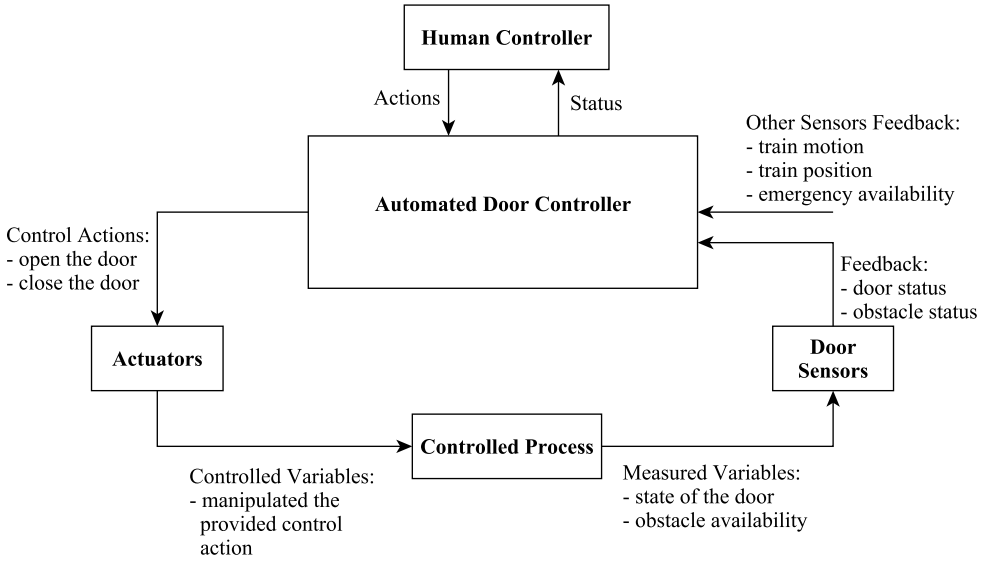

Figure 5.4: Safety control loop for automated train door controller case study from [227]

- **Determining the main control actions**. The control actions here are open and close the door.

- **Identifying the process model**. We define the process model as a set of the following functions: trainMotion, trainPosition, emergency, obstacleStatus, doorStatus.

- **Evaluating the combination of the function values and documenting the results in a context table**. The evaluation results are documented in Table 5.1. The no/fun answer in Table 5.1 means the action will not lead to a hazard, but providing or not providing it includes a flaw in the system function. For example, when the door is already opened, the train is stopped such that it is aligning with the platform, and the person is not in the doorway, then providing the open action, in this case, implies that there is a flaw in the controller's function.

- **Translate each combination that has yes answer in the table into informal safety requirements using *must/must not* phrases**. For example, the the open action must be provided when the door is closing on a person in the doorway.

Process Model					Hazardous Control Action?			
trainMotion	emeregency	trainPosition	obstacleStatus	doorStatus	Provided	Provided too early	Provided too late	Not Provided
Stopped	Not Exist	Aligned	Not Exist	Closed	no	no/fun	no/fun	nos
Stopped	Not Exist	Aligned	Not Exist	Opened	no/fun	no/fun	no/fun	no
Stopped	Not Exist	Aligned	Not Exist	Closing	no/fun	no/fun	no/fun	no
Stopped	Not Exist	Aligned	Not Exist	Opening	no/fun	no/fun	no/fun	no
Stopped	Not Exist	Not Aligned	Not Exist	any	yes	yes	yes	no
Stopped	Not Exist	Aligned	Exist	Opened	no/fun	no/fun	no/fun	no
Stopped	Not Exist	Aligned	Exist	Opening	no/fun	no/fun	no/fun	no
Stopped	Not Exist	Aligned	Exist	Closing	no	no/fun	yes	yes
Stopped	Not Exist	Not Aligned	Exist	any	no/fun	no/fun	no/fun	no/fun
any	Exist	any	any	Closed	no	no/fun	yes	yes
any	Exist	any	any	Closing	no	no/fun	yes	yes
any	Exist	any	any	Opening	no/fun	no/fun	no/fun	no
any	Exist	any	any	Opened	no/fun	no/fun	no/fun	no
Moving	Not Exist	any	any	any	yes	yes	yes	no

Table 5.1: The context table for the open door action

5.3.4 Formalizing the Elicited STPA Safety Requirements

After applying the STPA procedure, we formulate the STPA requirements using the following the steps:

- Determine the combination of the main functions according to the yes answers in the 'Not Provided' condition only. The purpose for that is to ensure that the action is provided (not too early, on time, not too late) with these combinations not with the other. For example, if we ensure that the open door action is only given when there is an emergency or the train stopped in a way to the platform and there is no an obstacle, and these circumstances always imply providing this action, then there is no need to check whether the open action is not provided when the train is moving.

  With regard to Table 5.1, the combinations that have been identified are:

  1. doorStatus=CLOSING, obstacleStatus=EXIST, emergency=NOTEXIST, trainPosition=ALIGNED, and trainMotion=STOPPED.
  2. doorStatus=CLOSED, and emergency=EXIST.

3. doorStatus=CLOSING, and emergency=EXIST.

- Formulate the above combinations using formula 5.1, to produce the following LTL formula:

$$\Box\ (((\text{doorStatus}=\text{CLOSING} \wedge \text{obstacleStatus}=\text{EXIST} \wedge \text{emergency}=\text{NOTEXIST} \wedge \text{trainPosition}=\text{ALIGNED} \wedge \text{trainMotion}=\text{STOPPED}) \vee (\text{doorStatus}=\text{CLOSED} \wedge \text{emergency}=\text{EXIST}) \vee (\text{doorStatus}=\text{CLOSING} \wedge \text{emergency}=\text{EXIST})) \leftrightarrow \bigcirc(\text{doorStatus}=\text{OPENED}))$$

5.3.5 Verifying the Formulated Safety Requirements

In this stage, we use the AsmetaSMV tool to perform the verification activity. As a result, we rewrite the resulting formulated requirements in the previous stage into another one accepted by this tool. Accordingly, the LTL property in the previous section is rewritten into the following form:
LTLSPEC g(((doorStatus=CLOSING and obstacleStatus=EXIST and emergency=NOTEXIST and trainPosition=ALIGNED and trainMotion=STOPPED) or
(doorStatus=CLOSED and emergency=EXIST) or
(doorStatus=CLOSING and emergency=EXIST)) iff x(doorStatus=OPENED)).

The LTLSPEC means that the property is from linear time temporal logic. The property informally means that the door is OPENED in the next state if and only if one of the identified combinations occurred. This property is not met by the AsmetaL model for the TDC case study, as the door is opened with one more combination: doorStatus=OPENED and emergency=EXIST. The failing trace for this property is shown in Figure 5.5.

State	1.1	1.2	1.3	1.4	1.5
doorStatus	CLOSED	CLOSED	OPENED	OPENED	OPENED
emergency	NOTEXIST	EXIST	EXIST	EXIST	EXIST
state	SENSING	EXECUTING	EXECUTING	EXECUTING	EXECUTING
emergencySensor	EXIST	EXIST	EXIST	EXIST	EXIST

Figure 5.5: Failing trace of running open door action

At state 1.1 in this Figure, the door state is CLOSED, the emergency does not exist, the system state is SENSING, and the emergency sensor detects that there is an emergency. The system state is changed into EXECUTING at state 1.2 and the emergency function is updated to EXIST. At state 1.3, the door state is changed to OPENED. The loop starts at

state 1.4 showing that the door state is updated to OPENED when there is an emergency and the door state is already opened. This happens because the AsmetaL specification of the case study includes a rule, called r_HandleEmergency, in which the the door state is changed to OPENED without checking if it has already been opened, see Code 5.4a. In order to avoid the unnecessary opening of the door when the door is already opened, we updated the r_HandleEmergency rule such that the door is only opened when it has not been opened before, see Code 5.4b.

```
rule r_HandleEmergency=
    doorStatus:=OPENED
```

(a) The wrong specification

```
rule r_HandleEmergency=
 if doorStatus!=OPENED then
     doorStatus:=OPENED
 endif
```

(b) The correct specification

Code 5.4: The r_HandleEmergency rule for TDC example

5.3.6 Evaluation of the TDC Case Study

The application of the ASM-STPA-SA methodology to the TDC case study helps to improve the model for this case study by utilizing the outcomes of this methodology stages. With the simulation stage, we were able to notice that the system cannot resume its normal work after handling the emergency. This happens because the TDC model was wrongly specified such that when there is an emergency, the system stops its work and never goes to the SENSING state.

The validation stage helped to increase our confidence that the TDC model reflects the user requirement which is related to ensure that the door must be opened while there is an obstacle in the doorway.

The verification stage aided at observing that the door is also opened with a non-mentioned combination which is the emergency exists and the door is already opened. This observation enables us recognizing that the model does not consider checking the state of the door before opening it.

5.4 Case Study: The Insulin Pump Control System

The second case study of this work is the Insulin Pump Control System (IPCS). The IPCS is chosen because its design is more complex than the one for TDC case study, and the

investigation of its safety aspects is still ongoing. In this section, we will introduce this case study and we will show in details how to apply the ASM-STPA-SA method to it.

5.4.1 The IPCS Case Study Description

The Insulin Pump Control System (IPCS) is a therapeutic system used to improve diabetes treatment. The problem with traditional treatments is the possibility of taking an insulin overdose or insufficient dose due to focusing only on the current glucose value and ignoring the last insulin injection time. It has been utilized as a case study for software analysis of safety-critical systems in [220].

The IPCS consists of sensor, software controller, pump, and insulin reservoir. The sensor measures the current glucose value and delivers it to the controller. The controller analyses the received value, calculates and delivers the required dose, and tests the hardware units. The dose is delivered by sending a signal to the pump for each insulin unit. The maximum capacity of the insulin reservoir is 100 units. The IPCS works in three different modes: automatic, manual, and switching off.

In the automatic mode, the software controller can implement one of the following two activities at a time: running which is performed every 10 minutes, or testing which is performed every 30 seconds. In addition, the software controller resets the cumulative dose to 0 every 24 hours.

The running activity starts with sensing the current glucose value, then this value is analysed by comparing it with two saved values (10 and 20 minutes prior) in order to calculate the required dose. Typically, the controller computes and delivers a dose through carrying out the following checking steps: (in the steps below, v_2 is the current glucose value, v_1 is the glucose value 10 minutes before, and $v02$ is the glucose value 20 minutes before)

(1) If the current value is less than the minimum safe limit (6), then the computed dose is equal to zero.

(2) if the current value is within the safe range (6..14, i.e., minimum safe limit=6 units, and maximum safe limit=14 units), then the insulin dose will be computed depending on the following:

   (a) If the glucose level is decreasing or stable ($v_2 \leq v_1$), or the glucose level is increasing ($v_2 > v_1$) but the rate of glucose level increase is decreasing (($v_2 - v_1$)<

$(v_1 - v_0)$), then the computed dose is equal to zero.

(b) If the glucose level is increasing ($v_2 > v_1$) and the rate of glucose level increase is rising (($(v_2 - v_1) \geq (v_1 - v_0)$), then the computed dose will either be equal to the minimum dose (1 unit) if the rounded division of the ($(v_2 - v_1)/4$) is equal to zero; otherwise, it is equal to the rounded division itself.

(3) If the current value exceeds the maximum safe limit, the glucose level is falling, and the rate of decrease is increasing, then no dose must be delivered; otherwise, a dose must be computed such that it will bring down the glucose level. More precisely, the computed dose is calculated as follows:

(a) If the glucose level is stable or it is falling but the rate of decrease is decreasing, then the computed dose is equal to the minimum dose.

(b) When the glucose level is increasing, the computed dose will either be equal to the minimum dose (1 unit) if the rounded division of the ($(v_2 - v_1)/4$) is equal to zero; otherwise, it is equal to the rounded division itself.

(4) When there is a dose must be delivered to the patient (the computed dose is not equal to zero), three safety checks must be examined before delivering it.

(a) If the summation of the computed dose and the cumulative dose is greater than the maximum daily dose (25 units), then the computed dose must be equal to subtracting the cumulative dose from the maximum daily dose.

(b) If the summation of the computed dose and the cumulative dose is less than the maximum daily dose and the computed dose is greater than the maximum single dose (4 units), then the computed dose must be equal to only 4 units.

(c) If the summation of the computed dose and the cumulative dose is less than the maximum daily dose (25 units) and the computed dose is less than or equal to the maximum single dose (4 units), then the computed dose is delivered as it is without modification.

Within the running activity, the warning alarm must be run when the received glucose value is less than the minimum safe limit, the available insulin is less than or equal to the four maximum single doses, or delivering the dose will exceed the maximum daily dose. The testing activity involves detecting any hardware unit failure, including sensor, battery,

needle, and insulin reservoir, to suspend the IPCS work and to run a failure alarm. In the manual mode, the system will deliver the dose manually, hence the software controller will not perform safety checking, but it will update the quantity of the available insulin and the cumulative dose. The maximum manual dose is 5 units. The complete requirements are documented and specified in the Z language in [219], and part of the specification is provided in [220].

In the following, we will describe the detailed steps and the results of applying the ASM-STPA-SA method to the IPCS system.

5.4.2 Methodology Applied to IPCS Case Study

In this section, we apply our methodology to the IPCS case study.

5.4.2.1 Modelling the System Behavior via AsmetaL and Simulating the Resultant Model

In this stage, we show how to model the IPCS via AsmetaL and how to exploit the AsmetaS tool to simulate the obtained model.

Specification. As a simplified model for the IPCS was already presented in [220, 219], we focus here on modelling issues unaddressed by these works, such as switching between the system operation modes (automatic, manual, or switching off) at any state by the user, and timing details of the software controller activities, which include: testing, running, and setting the cumulative dose to 0, occurring every: 30 seconds, 10 minutes, and 24 hours, respectively. As a result, in this section, we discussed the specifications for these issues[1]. Specifications of the switching off mode and the running activity stages, that include sensing the glucose value, analysing it, calculating the insulin dose, checking the calculated dose and delivering it, are not presented here, since they are detailed in [220, 219].

Code 5.5 shows the r_main rule for the IPCS, in which changing the system operation modes at any state by the user has been modelled. In this code, we defined the boolean monitored function switchMode to denote whether a user wants to change the system's operation mode. This function drives the transition from any state into another one in the whole model. We also declared the btn controlled function that represents the operation mode of the system, which can be ON (automatic mode), MANUAL (manual mode),

[1]The complete model for the IPCS together with the Avalla scenarios are available in [25]

or OFF (switching off mode). The value of the btn function is updated by the button monitored function when the switchMode function has true value. While when this function has false value, three rules are executed in parallel, which are: r_Automatic[], r_Manual[], and r_Switchoff[], that represent the system's operation mode: automatic, manual, and switching off, respectively.

```
enum domain Button={ON, OFF, MANUAL}
monitored button:Button
monitored btn:Button
monitored switchMode: Boolean

main rule r_Main =
  if switchMode=true then
    par
      btn:=button
      r_CancelActions[]
    endpar
  else
    par
      r_Automatic[]
      r_Manual[]
      r_Switchoff[]
    endpar
  endif
```

Code 5.5: The r_main rule for the IPCS case study

The r_Automatic[] rule, shown in Code 5.6 is fired when btn is ON.

```
controlled spn:Boolean
rule r_Automatic=
 if btn=ON then
  par
    if (spn=false) then
      par
        r_Running[]
        r_Testing[]
      endpar
    else
      par
        r_Check_Reservoir[]
        r_Check_Needle[]
      endpar
    endif
    r_Update_Timecycle[]
    r_CancelAction[]
  endpar
 endif
```

Code 5.6: The r_Automatic rule for the IPCS case study

In this rule, when the system is not suspended from its work (the spn function false value), it, depending on the time constraints and software controller state, either implement the running activity by the r_Running[] rule or the testing activity by the r_Testing[] rule.

While, when the system is suspended due to a hardware unit failure, it must not implement any of these activities but it keeps checking whether the failure is managed to resume its work again. In the r_Automatic[] rule also, whether the system is suspended or not, the time must be updated through the r_Update_Timecycle rule which we will explain it later.

As the running and testing activities are based on the state of the software controller, we defined the cS function to represents this state, which can be SNS (sensing the current glucose value), ANLSCAL (analysing the current glucose value and calculating the dose), SFTCK (safety checking for the computed dose before delivering it), DLVR (delivering the dose), or TST (testing). Using this function we modelled the running activity by the r_Running rule, see Code 5.7. This rule starts when the controller state is SNS. If it is so, then the current glucose value is obtained from the sensor via the valS function, it is saved in the cR function, and the state of the controller is set to ANLSCAL. After calculating the dose, the value of cS function becomes SFTCK. When the safety checking for the computed dose is finished by the r_SafetCheck rule, the cS value changes into DLVR. At the time that the full dose has been delivered, the running activity ends and the cS function turns into TST to perform the testing activity.

```
enum domain Controller={SNS, ANLSCAL,
    SFTCK, DLVR, TST}
domain Dose subsetof Integer
controlled cS:Controller
controlled cR:Dose
controlled mSD:Dose
controlled mDD:Dose
controlled cD:Dose
controlled comD:Dose
controlled dD:Dose
monitored valS:Dose
function mSD=4
function mDD=25
domain Dose={0..35}

rule r_Sense=
 if ((iA>=mSD) and (cD<mDD)) then
   if (exist $x in Dose with $x =valS)
      then
     par
       cR:=valS
       cS:= ANLSCAL
     endpar
   endif
 endif
```

```
rule r_SafetyCheck=
 if comD=0 then
   dD:= 0
 else
  if (comD+cD)>mDD then
     dD:=mDD-cD
  else
    if (comD+cD)<mDD then//No equality
      if comD<=mSDthen
        dD:=comD
      else
        dD:=mSD
      endif
    endif
  endif
 endif

rule r_Running=
 switch cS
  case SNS: r_Sense[]
  case ANLSCAL: r_AnalyseAndCalculate[]
  case SFTCK: r_SafetyCheck[]
  case DLVR: r_Deliver[]
 endswitch
```

Code 5.7: Specification of running activity for the IPCS case study

Performing the testing activity will be through inspecting the values for the iA, ndlExist,

rsvExist, and failure functions in the r_Testing rule, see Code 5.8. The iA function is the available insulin value, which must not be less than the maximum single dose (mSD=4). The rsvExist function records whether the reservoir exists (prs) or not (not), and the same role for the ndlExist function of the needle. The failre function records whether there is a failure in any hardware unit, such as the sensor, pump, needle, or battery. When there is no failure in the system, the needle exists, and the reservoir exists, the returned values by the fl, ndl, and rsv monitored functions must be kept checked, respectively. If the returned values show that the system has a hardware failure or no existence state, then the system must be suspended by updating the value of the controlled function spn to true, and at the same time the alarm command must be run through the alarmCommand function and a convenient message must be sent via msgCommand function as well.

```
enum domain Message={NOINSULIN |
    NORESERVOIR | FAILURE |NONEEDLE}
enum domain Present={PRS | NOT}
monitored rsv: Present
monitored ndl:Present
monitored fl:Boolean
controlled rsvExist: Boolean
controlled failure: Boolean
controlled ndlExist: Boolean
controlled alarmCommand: Boolean
controlled msgCommand: Message->Boolean
controlled display1:Message
rule r_Testing=
  if cS=TST then
   par
     if failure=true then
      par
        msgCommand(FAILURE):=true
        display1:=FAILURE
      endpar
     else
       if (fl=true) then
         failure:=true
       endif
     endif
     if rsvExist=false then
       par
         msgCommand(NORESERVOIR):=true
         display1:=NORESERVOIR
       endpar
     else
       if rsv=NOT then
         rsvExist:=false
       endif
     endif
   if ndlExist=false then
     par
       display1:=NONEEDLE
       alarmCommand:= true
     endpar
    else
      if ndl= NOT then
        ndlExist:=false
      endif
    endif
    if iA<mSD then
      par
        msgCommand(NOINSULIN):=true
        display1:=NOINSULIN
      endpar
    endif
   endpar
   if (failure=true) or (rsvExist=false) or
       (ndlExist=false) or (iA<mSD) then
     par
        spn:=true
        alarmCommand:= true
     endpar
   endif
  endif
```

Code 5.8: Specification of testing activity for the IPCS case study

Executing the testing and running activities are also restricted by time (30 seconds and 10 minutes). This is carried out by the r_Update_Timecycle rule, see Code 5.9.

```
controlled sC30:ThirtyC
controlled mC10:TenC
domain ThirtyC subsetof Integer
domain TenC subsetof Integer
domain Seconds={30}
domain ThirtyC={1..20}
domain TenC={1..144}
rule r_Update_Timecycle=
  if (sC30=1) and (cS!=TST) then
    if (cS=DLVR) and (nP=0) then
      par
        sC30:=sC30+1
        cS:=TST
      endpar
    endif
  else
    if (pas(30)=true) then
      if (sC30=20) then
        par
          sC30:=1
          cS:=SNS
          if (mC10=144) then
            par
              mC10:=1
              cD:=0
            endpar
          else
            mC10:=mC10+1
          endif
        endpar
      else
        par
          sC30:=sC30+1
          cS:=TST
        endpar
      endif
    endif
  endif
```

Code 5.9: Timing aspects specification of the IPCS case study

As there is no tool to deal with time within the ASMETA framework, we treat time in an abstract manner. To achieve this, we use the controlled function sC30 to represent the number of 30 second cycles in 10 minutes. The maximum value for this function is 20. As the controller performs the running activity every 10 minutes and the testing activity every 30 seconds (but running and testing can not take place at the same time), one of these 20 cycles is for running and the other cycles are for testing. During the running activity, the controller sets the cumulative dose cD to 0 every 24 hours. We use the controlled function mC10 to represent the number of 10 minute cycles in 24 hours (its maximum value is 144). When this function reaches 144 and the sC30 function reaches 20, then the controller will set the cumulative dose to 0. Furthermore, we deal with increasing these functions in an abstract manner via the boolean monitored function pas(30). This means, when the pas(30)=true, some function should be increased, and at the same time, some activity should be performed. For example, if sC30=20 and 30 seconds has passed since the last update of sC30 to 20, then the running activity must be started by changing cS into SNS, sC30 becomes 1, and at the same time mC10 is checked. If it has reached 144, it is set to 1, otherwise it is increased to the next value. If 30 seconds has passed since the last update of sC30 to a value within 1-19, then the testing activity must be performed and sC30 is increased to the next value.

Regarding the manual mode for the IPCS, we specified this mode by the r_Manual rule, see Code 5.10. In the r_Manual rule, when the monitored function mD returns a value within the range of the manual dose (1..5), the available insulin is updated by providing

the action updateiACommand, and the cumulative dose is updated by issuing the action updatecumDCommand.

```
monitored mD: Dose
controlled updateiACommand: Boolean
controlled updatecumDCommand: Boolean
rule r_Manual=
 if btn=MANUAL then
   if (exist $md in Dose with
   (($md=mD) and ($md>=1 and $md<=5)))
   then//Here there is no checking whether
   the available insulin is equal or
   greater than the maximum manual dose
      par
        updateiACommand:=true
        updatecumDCommand:=true
        display1:=MANUALLY
        iA:=iA-mD
        cD:=cD+mD
      endpar
    endif
  else
    par
      updateiACommand:=false
      updatecumDCommand:=false
    endpar
  endif
```

Code 5.10: Specification of the manual mode for the IPCS case study

Simulation. Concerning the simulation of the IPCS model by the AsmetaS tool, it enabled us to identify the incorrect ASM state that emerges from firing inaccurate specified rule. This rule is the r_CheckReservoir. It is specified for checking that the reservoir has been returned back after removing it, where the reservoir is removed when it becomes empty. The specification for this rule is shown in Code 5.11a.

Note that the monitored and controlled functions used for the IPCS are enumerated in Table 5.2 with their descriptions.

```
rule r_Check_Reservoir=
 if rsvExist=false then
   if rsv=PRS then
     par
        iA:=100
        rsvExist:true
        spn:=false
     endpar
   endif
 endif
```

(a) The wrong specificatio

```
rule r_Check_Reservoir=
 if rsvExist=false then
   if rsv=PRS then
     par
        iA:=100
        rsvExist:true
        if (ndlExist=true) and (cD<mDD)
              and (failure=false) then
           spn:=false
        endif
     endpar
   endif
 endif
```

(b) The correct specification

Code 5.11: The r_Check_Reservoir rule

Function Name	Function Definition	Function Type	Function Values	
			Value	Value Definition
alarmCommand	Alarm action	Controlled	True False	Alarm action is provided Alarm action is not provided
btn	Button for changing the operation mode of the system	Controlled	ON MANUAL OFF	Automatic operation mode Manual operation mode Switching off operation mode
Button	Button for changing the operation mode of the system	Monitored	ON MANUAL OFF	Automatic operation mode Manual operation mode Switching off operation mode
cD	Cumulative dose	Controlled	0 to 35	-
cR	Current read of the glucose value	Controlled	0 to 35	-
cS	State of the controller	Controlled	SNS ANLSCAL SFTCK DLVR TST	Sensing the current glucose value Analysing the current glucose value and calculating the dose Safety checking for the computed dose before delivering it Delivering the dose Testing
dD	Delivered dose	Controlled	1 to 100	-
failure	Hardware unit failure	Controlled	True False	The failure exists The failure does not exist
fl	Hardware unit failure	Monitored	True False	The failure exists The failure does not exist
iA	The amount of the available insulin	Controlled	1 to 100	-
mC10	The number of 10 minute cycles in 24 hours	Controlled	1 to 144	-
mD	Manual dose	Monitored	0 to 35	-
mDD	Maximum daily dose	Controlled	0 to 35	-
minD	Minimum dose	Controlled	0 to 35	-
mSD	Maximum single dose	Controlled	0 to 35	-
ndl	Needle	Monitored	PRS NOT	The needle exists The needle does not exist
needleExist	The existence of the needle	Controlled	True False	The needle exists The needle does not exist
nP	Number of pulses	Controlled	0 to 35	-
pas(30)	Passing 30 seconds	Monitored	True False	The 30 seconds are passed The 30 seconds are not passed
pR	Reading the glucose value10 minutes prior to the current one	Controlled	0 to 35	-
sC30	The number of 30 second cycles in 10 minutes	Controlled	1 to 20	-
rsvExist	The existence of the reservoir	Controlled	True False	The reservoir exists The reservoir does not exist
rsv	Insulin reservoir	Monitored	PRS NOT	The reservoir exists The reservoir does not exist
spn	suspention	Controlled	True False	The work of the system is suspended The work of the system is not suspended
updateiACommand	Updating the available insulin action	Controlled	True False	Updating available insulin action is provided Updating available insulin is not provided
valS	Sensing the current glucose value	Monitored	0 to 35	-

Table 5.2: The function for the IPCS and their denotations

In Code 5.11a, the monitored function rsv is used as a sensor to return feedback about the reservoir's presence. When it has the PRS (present) value, the controlled function rsvExist is updated to true and the function spn is changed to false in order to make the system resumes its work again. After simulating the AsmetaL code that includes this rule, there is a chance of an incorrect ASM state emerging from hardware failure that coincides with the reservoir returning process, then changing the value of the spn function without prior check is inaccurate, because that means the system might return to its operating conditions even with a failure. The specification of the r_CheckReservoir rule in Code 5.11a was inspired from [220], in which the system resumes its work when the reservoir is returned back to the system with 100 units, without considering that there is a possibility of a hardware unit failure at the same time as the reservoir is returned back.

Figure 5.6 shows part of the simulation trace for the AsmetaL that includes the r_CheckReservoir rule with inaccurate specification. In this Figure, state 7 manifests that the value of spn is updated to false even when the value of the failure function is true. According to this simulation, the specification for the r_Check_Reservoir rule is updated to include a no failure check before canceling the suspension situation, see Code 5.11b.

State		5	6	7
Controlled Functions	btn	ON	ON	ON
	cS	TST	TST	TST
	spn	false	true	false
	ndlExist	true	true	true
	rsvExist	true	false	true
	failure	false	true	true
	sC30	2	3	4
Monitored Functions	switchMode?	false	false	false
	pas(30)?	true	true	true
	fl?	true	true	true
	ndl?	PRS	PRS	PRS
	rsv?	NO	PRS	PRS

Figure 5.6: Part of the simulation for the IPCS model with incorrect rule

5.4.2.2 Validating the AsmetaL Model

Following the simulation, we specify different scenarios to validate that the AsmetaL model satisfies them. A scenario describes the identifiable interactions between the system and its environment to represent informal functional requirements. These interactions for the IPCS are represented by the current glucose value and the delivered dose. We identify 14 scenarios that correspond to the delivered dose quantity requirements, which are introduced

in Section 5.4.1. From these scenarios, we only discuss the scenario that has a fail verdict[2]. This scenario corresponds to the following requirements: if the cumulative dose does not exceed the maximum daily dose, and the computed dose itself is less than or equal to the maximum single dose, then the delivered dose is equal to the computed dose.

```
//setting the initial 250 states
set switchMode:=false;
set valS:=22;
step
check cR=22 and cS=ANLSCAL;
step until cS=TST;
check cD=22;
set switchMode:=false;
set pas(30):=true;
set fl:=false;
set rsv:=PRS;
set ndl:=PRS;
step until sC30=20;
```

```
check spn=false;
set switchMode:=false;
step
check cS=SNS;
set switchMode:=false;
set valS:=34;
step
check cR=34 and cS=ANLSCAL;
step
check comD=3 and comD+cD<=25 and
                          comD<=mSD;
step
check dD=comD;
```

Code 5.12: The scenario which corresponds to the requirement of delivering the computed dose if it was less than or equal to the maximum single dose and the total amount not exceeding the maximum daily dose

The scenario in Code 5.12, which is written in Avalla, can be described as follows: the system is operating in automatic mode and remains in this mode, the current glucose value is 34, the previous glucose value from 10 minutes earlier is 22, the cumulative dose is equal to 22, the system is not suspended from its work, the computed dose is 3 units, and the requirement that must be checked is: the delivered dose should be equal to the computed dose.

The simulation of the scenario in Code 5.12 is illustrated in Figure 5.7. In this figure, the comD and dD functions represent the computed dose and the delivered dose, respectively. The simulation shows that we obtain the success verdict for the first received glucose value (22), the cumulative dose, no suspension, the second received glucose value (34), the sum of the computed dose and the cumulative dose that is equal to the maximum daily dose (25), and the computed dose is less than the maximum single dose (4), while a fail verdict is obtained at state 283, due to missing the equality operator in the safety condition on the computed dose before delivering it (see r_SafetyCheck rule in Code 5.7). This condition checks whether the summation of the computed dose plus the cumulative dose is greater or less than the maximum daily dose, but it does not checks equality situation

[2]The specifications of the 13 scenarios that have success verdicts are available online in [25], see Code 5.12, and the corresponding requirement for each scenario is in Appendix B

((comD+cD)=mDD, i.e. 3+22=25). Therefore, the delivered dose is not calculated and we obtained a failed verdict. Thus, we have shown that ignoring the equality testing in the [219] specification may lead to a serious issue in the IPCS. As a result, we updated the specification for r_SafetCheck rule in Code 5.7, such that the if-then condition, which is if (comD+cD)<mDD then, is changed to include the equality check, i.e. if (comD+cD) $\leq$ mDD then.

State		251	252	...	258	259	...
Controlled Functions	btn	ON	ON		ON	ON	
	cS	SNS	ANLSCAL		TST	TST	
	spn	false	false		false	false	
	cR	0	22		0	0	
	cD	0	0		22	22	
	comD	0	0		0	0	
	dD	0	0		0	0	
	sC30	1	1		2	3	
Monitored Functions	switchMode?	false	false		false	false	
	pas(30)?	-	-		true	true	
	valS?	22	-		-	-	
	fl?	-	-		-	false	
	ndl?	-	-		-	PRS	
	rsv?	-	-		-	PRS	
Verdict			CHECK SUCCEEDED: cR=22		CHECK SUCCEEDED: cD=22		
State		**279**	**280**	**281**	**282**	**283**	
Controlled Functions	btn	ON	ON	ON	ON	ON	
	cS	TST	SNS	ANLSCAL	SFTCK	DLVR	
	spn	false	false	false	false	false	
	cR	0	0	34	0	0	
	cD	22	22	22	22	22	
	comD	0	0	0	3	3	
	dD	0	0	0	0	0	
	sC30	20	1	1	1	1	
Monitored Functions	switchMode?	false	false	false	false	false	
	pas(30)?	true	true	-	-	-	
	valS?	-	34	-	-	-	
	fl?	-	-	-	-	-	
	ndl?	-	-	-	-	-	
	rsv?	-	-	-	-	-	
Verdict		CHECK SUCCEEDED: spn=false		CHECK SUCCEEDED: cR=34	CHECK SUCCEEDED: comD=3 and comD+cD<=25 and comD<=mSD	CHECK FAILED: dD=comD	

Figure 5.7: Simulation of the scenario shown in Code 5.12

5.4.2.3 Eliciting safety requirements for the system via STPA

After performing the validation activity, we elicit the safety requirements by applying the following steps of the STPA process:

- **Identifying the main accidents**. The IPCS has the following accidents:
  - A patient is prone to transient or permanent dysfunction or even death when an unnecessary insulin dose is given.

  - Damage to the patients eyes, kidneys, nerves, or heart if the required insulin dose is not given.
  - Exposure to allergic reactions or infections.

- **Indicating the main hazards related to the identified accidents**. The IPCS has the following hazards:
  - The users unawareness of warning signs or system failure conditions.
  - The given insulin dose exceeds the maximum single or daily dose.
  - The delivered insulin dose is less than the requisite computed dose.
  - The decision to deliver or not an insulin dose has been taken without checking the current glucose value.
  - The insulin dose is delivered at the wrong time.
  - The shown dose value on the display unit is different from the actual delivered one.
  - Unintended switching to the manual mode without showing that on the display unit.
  - The insulin pump control device is still working even while a hardware failure is located, the available insulin is less than the maximum single dose, the insulin reservoir is removed, or the needle does not exist.
  - The first display unit shows messages unrelated to the actual hardware unit's situation.

- **Drawing the schematic safety control loop**. Figure 5.8 shows the safety control loop diagram for the IPCS case study.

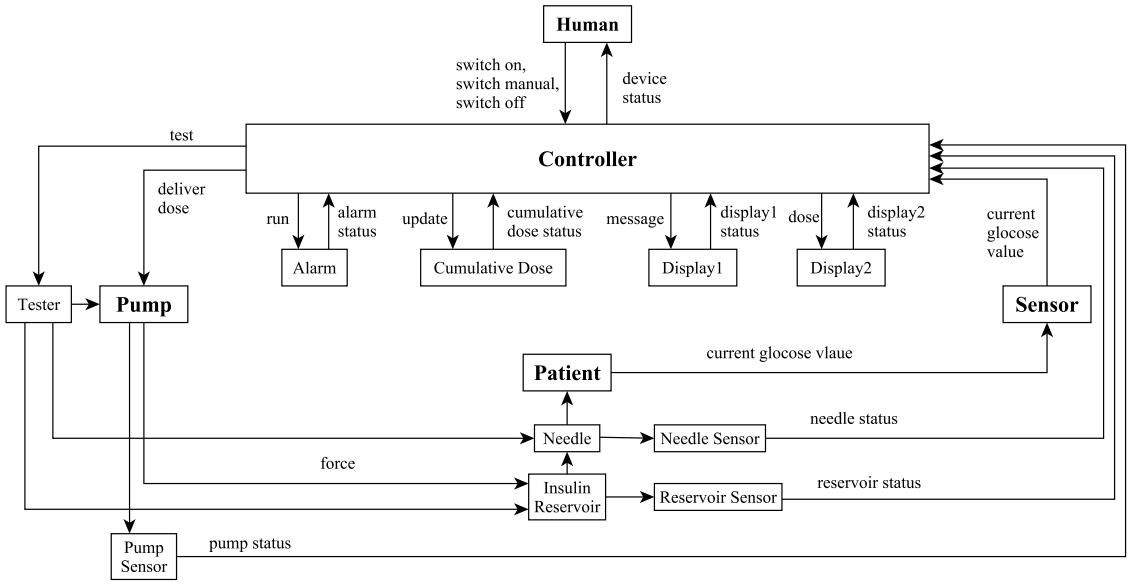

Figure 5.8: Safety control loop for the insulin pump control system

- **Determining the main control actions**. The control actions here are run the alarm, update the available insulin, deliver the dose, display messages, and test the hardware units.

- **Identifying the process model**. We define it as a set of monitored and controlled functions affecting providing the analysed actions. Each member of this set consists of the function name and its values, e.g., the process model that affects the run warning alarm action is: {switchMode=(true, false), btn=(ON, OFF, MANUAL), spn= (true, false), cR=(≥sMin, <sMin), cS=(SNS, ANLSCAL, SFTCK, DLVR), nP=(0, 1, 2, 3, 4), iA=(>4×mSD, ≤ 4×mSD), sCC=(>mDD, ≤mDD)}. Where the meaning of cR is: current read of glucose, sMin is: minimum safe limit (6), mDD is: maximum daily dose (4), sCC is: summation of computed and cumulative dose, and nP: pulses number issued by controller to deliver the insulin.

- **Evaluating the combination of the function values and document the results in a context table**. The evaluation process is through asking a question to an expert in the following form: if the controller receives a certain combination of function values, will (providing, providing too early/too late, not providing) the action

in the next state by the controller lead to a hazard? The results of the evaluation are documented in Table 5.3. This Table is only for the run warning alarm. Note that, the alarm action is issued for patient's warning and system's suspension purposes. We consider only the warning purposes in Table 5.3 in order to keep the table as simple as possible. The no/fun answer in this table represents no actual hazard will happen, but there is a flaw with the system function, e.g., it is not hazardous if the alarm action is provided earlier than realizing that the current glucose is less than the minimum safe limit.

- **Translate each combination that has a yes answer in the table into informal safety requirements using *must/must not* phrases**. According to Table 5.3, we have 6 safety requirements corresponding to the 6 yes answers. For example, the warning alarm action must be provided when the system is in the automatic mode, this mode is not changed, and the current glucose value is less than the minimum safe limit.

5.4.2.4 Formalizing the Elicited STPA Safety Requirements

The next stage is formalizing some of the obtained STPA requirements through performing the following steps:

- Determine the main functions combination according to the yes answers in the 'Not Provided' condition only. The purposes for that, are: to ensure that the action is provided with these combinations, not with the other. With regard to Table 5.1, the combinations that have been identified are:

  1. switchMode=false, btn=ON, spn=false, cS=DLVR, nP=0, and iA$\leq$ 4mSD.
  2. switchMode=false, btn=ON, spn=false, cS=SFTCK, and sCC>mDD.
  3. switchMode=false, btn=ON, spn=false, cS=ANLSCAL, and cR<sMin.

- Formulate the above combinations using formula 5.1, to produce the following LTL requirement:

$$\Box\ (((\text{switchMode}=\text{false} \wedge \text{btn}=\text{ON} \wedge \text{spn}=\text{false} \wedge \text{cS}=\text{DLVR} \wedge \text{nP}=0 \wedge \text{iA}\leq 4\text{mSD}) \vee (\text{switchMode}=\text{false} \wedge \text{btn}=\text{ON} \wedge \text{spn}=\text{false} \wedge \text{cS}=\text{SFTCK} \wedge \text{sCC}>\text{mDD}) \vee (\text{switchMode}=\text{false} \wedge \text{btn}=\text{ON} \wedge \text{spn}=\text{false} \wedge \text{cS}=\text{ANLSCAL} \wedge \text{cR}<\text{sMin})) \leftrightarrow \bigcirc(\text{alarmCommand}=\text{true}))$$

Process Model								Hazardous Action?			
switchMode	btn	spn	cR	cS	nP	iA	sCC	Provided	Provided too early	Provided too late	Not Provided
false	ON	false	any	DLVR	=0	≤4mSD	any	no	no/fun	yes	yes
false	ON	false	any	DLVR	>0 and ≤4	≤4mSD	any	no/fun	no/fun	no/fun	no
false	ON	false	any	DLVR	=0	>4mSD	any	no/fun	no/fun	no/fun	no
false	ON	false	any	SNS	any	any	any	no/fun	no/fun	no/fun	no
false	ON	false	any	SFTCK	any	any	≤mDD	no/fun	no/fun	no/fun	no
false	ON	false	any	SFTCK	any	any	>mDD	no	no/fun	yes	yes
false	ON	false	<sMin	ANLSCAL	any	any	any	no	no/fun	yes	yes
false	ON	false	≥sMin	ANLSCAL	any	any	any	no/fun	no/fun	no/fun	no
false	ON	true	any	any	any	any	any	no/fun	no/fun	no/fun	no
false	MANUAL	any	any	any	any	any	any	no/fun	no/fun	no/fun	no
false	OFF	any	any	any	any	any	any	no/fun	no/fun	no/fun	no
true	any	any	any	any	any	any	any	no/fun	no/fun	no/fun	no

Table 5.3: The context table for the run alarm action with warning conditions

5.4.2.5 Verifying the Formulated Safety Requirements

This stage is intended to verify the resulted formulated requirements against the AsmetaL model. The output of this stage is either a true answer confirming that a given formulated requirement is satisfied, or a counter-example showing how this requirement is unsatisfied.

We model checked the original model for the IPCS case study, and we faced an issue of a large state space, hence we have somewhat modified our model to reduce the state space but without affecting the results of the IPCS. To do that, we made the testing and running activities less frequent such that testing activity will be done every 2 minutes instead of every 30 seconds and running activity will be performed every 40 minutes (after 19 tests), and the cumulative dose will be set to zero every 36 runs instead of 144.

We verified the requirements by running the AsmetaSMV tool on 64-bit Windows 10 PC with 1.8 GHz Intel Core i7-4500U processor, 8 GB main memory, and 1000 GB virtual memory.

In the following, the main safety requirement that we verified, such that the first two are the unsatisfied requirements while the rest are the satisfied ones. Furthermore, the first requirement is obtained from Table 5.3, but the rest are not produced from this table. We do not show the STPA tables and how to repeat the analysis steps for these requirements in order to make reading them easier.

(1) LTLSPEC g(((**switchMode=false and btn=on and spn=false and cS=dlvr and nP=0 and iA<=4*mSD)** or (switchMode=false and btn=ON and spn=false and cS=SFTCK and sCC>mDD) or (switchMode=false and btn=ON and spn=false and cS=ANLSCAL and cR<sMin)) iff x(alarmCommand=true)).

This property informally means that the warning run alarm action is provided in the next state if and only if one of the warning combinations occurs. Part of the formula is in bold font to indicate that this part (combination which is in bold) is the reason for unsatisfying this property.

In Figure 5.9, we show the failing trace for providing the run alarm action when the available insulin quantity is equal or less than 4 times the maximum single dose. The new abbreviation that we use in this figure is: pR (previous glucose reading). From state 1.1, onwards the system is operating under the automatic mode shown by the value ON. Furthermore, this mode is not changed within these states, as the value of the switchMode function is false. At state 1.1, there is no alarm action (alarmcommand=false) and the insulin quantity is 18 (iA=18). At state 1.2, the

controller receives the current glucose value (cR=22) from the sensor (valS=22), and it computes the dose at state 1.3 (comD=((cR=22)-(pR=14))/(mSD=4)). Delivering the dose starts at state 1.4, and at state 1.6 it finishes and the available insulin becomes 16 which is equal to (4×(maximum single dose=4)). The loop starts at state 1.6 showing that the warning alarm action is not provided (alarmCommand=false), when iA=16. This happens because the initial version of the AsmetaL model relies on the specification in [219], which does not consider running the alarm at the cautionary situations for the available insulin quantity, i.e. iA ≤(4×(maximum single dose=4)).

State	1.1	1.2	1.3	1.4	1.5	1.6	...	1.9
switchMode	false	false	false	false	false	false	false	false
btn	ON	ON	ON	ON	ON	ON	ON	ON
cS	SNS	ANLSCAL	SFTCK	DLVR	DLVR	TST	TST	TST
valS	22	22	22	22	22	22	22	18
iA	18	18	18	17	16	16	16	16
cD	3	3	3	4	5	5	5	5
cR	6	22	22	22	22	6	6	6
pR	14	14	14	14	14	22	22	22
comD	0	0	2	2	2	0	0	0
dD	0	0	0	2	2	0	0	0
alarmCommand	false	false	false	false	false	false	false	false

Figure 5.9: Failing trace for running alarm action when the available insulin is equal to the 4 maximum single doses

(2) LTLSPEC g((switchMode=false and btn=MANUAL and iA<=100 and mD!=0) iff x(updateiACommand=true)).

Where the mD is the manual dose, and the property informally means that the action of updating the available insulin according to the manual dose is always provided in the next state, if and only if the system works under the manual mode, the operation mode is fixed, the available insulin is less than or equal the capacity (100 units), and there is a manual dose. In Figure 5.10, we provide a failing trace for providing the update available insulin action when the system is in manual mode. From state 1.1 onwards, the system is in the manual mode via the value manual. At state 1.1, the insulin quantity is 9 (iA=9), the manual dose is 5, and updating the available insulin action is not provided (updateiACommand=false). At state 1.2, the action is provided and the value of iA is changed to 4. The loop starts at state 1.2 showing that the insulin quantity is not updated, when the iA=4 and the mD=5. The loop arises from a lack of a constraint, in the [219] specification, on the available insulin before delivering the manual dose (see r_Manual rule in Code 5.10). This constraint

must check if the available insulin is equal or greater than the maximum manual dose (5) before delivering it.

State	1.1	1.2	1.3	1.4	1.5
switchMode	false	false	false	false	false
btn	MANUAL	MANUAL	MANUAL	MANUAL	MANUAL
iA	9	4	4	4	4
mD	5	5	5	5	5
updateiACommand	false	true	false	false	false

Figure 5.10: Failing trace for updating available insulin action when the manual dose is greater than the available insulin

Hence the r_Manual rule must be modified such that the quantity of the available insulin is checked with respect to the value of the manual dose, i.e., the condition in this rule, which is if (exist $md in Dose with (($md=mD) and ($md>=1 and $md <=5))) then, must be modified into if (exist $md in Dose with (($md=mD) and ($md>=1 and $md <=5) and iA>= $md)) then.

(3) LTLSPEC g((switchMode=false and btn=ON and spn=false and (iA≤mSD∗4) and cS=DLVR and nP=0) iff x(msgCommand(INSULINLOW)=true)).
This property informally means that the action of displaying low insulin message is always provided in the next state if and only if the system is operating under the automatic mode, the operation mode is stable, there is no suspension, the controller state is delivering the dose, and the available insulin is equal or less than 4 times the maximum single dose.

(4) LTLSPEC g((switchMode=false and btn=ON and spn=false and cS=TST and (iA<mSD)) iff x(msgCommand(NOINSULIN)=true)).
This property informally means that the action of displaying no insulin message is always provided in the next state if and only if the system is operating under the automatic mode, the operation mode is stable, there is no suspension, the controller state is testing, and the available insulin is less than the maximum single dose.

(5) LTLSPEC g((switchMode=false and btn=ON and spn=false and cS=ANLSCAL and (cR<sMin)) iff x(msgCommand(SUGARBLOODLOW)=true)).
This property informally means that the action of displaying sugar blood low message is always provided in the next state if and only if the system is operating under the automatic mode, the operation mode is stable, there is no suspension, the controller

state is analysing and calculating the dose, and the current glucose value is less than the minimum safe limit (6).

(6) LTLSPEC g((switchMode=false and btn=ON and spn=false and cS=TST and (needleExist=false)) iff x(msgCommand(NONEEDLE)=true)).
This property informally means that the action of displaying no needle exist message is always provided in the next state if and only if the system is operating under the automatic mode, the operation mode is stable, there is no suspension, the controller state is testing, and the needle does not exist.

(7) LTLSPEC g((switchMode=false and btn=ON and spn=false and cS=TST and (failure=true)) iff x(msgCommand(FAILURE)=true)).
This property informally means that the action of displaying hardware failure existence message is always provided in the next state if and only if the system is operating under the automatic mode, the operation mode is stable, there is no suspension, the controller state is testing, and the failure exists.

(8) LTLSPEC g((switchMode=false and btn=ON and spn=false and cS=TST and (rsvExist=false)) iff x(msgCommand(NORESERVOIR)=true)).
This property informally means that the action of displaying no reservoir exist message is always provided in the next state if and only if the system is operating under the automatic mode, the operation mode is stable, there is no suspension, the controller state is testing, and the reservoir does not exist.

(9) LTLSPEC g((switchMode=false and btn=ON and spn=false and cS=DLVR and (nP!=0) and (dose!=0)) iff x(pulse=true)).
This property informally means that the action of sending pulses by the controller to deliver a dose (one pulse to each unit of dose) is always provided in the next state if and only if the system is operating under the automatic mode, the operation mode is stable, there is no suspension, the controller state is delivering a dose, the amount of dose is not equal to zero, and the number of pulses dedicated to delivering a dose is not equal to zero.

(10) LTLSPEC g((switchMode=true and button=ON) iff x(btn=ON)).
This property informally means that the system's operation mode is always changed

to automatic in the next state if and only if the user wants to change the mode and selects the ON button for that.

(11) LTLSPEC g((switchMode=true and button=OFF) iff x(btn=OFF)).
This property informally means that the system's operation mode is always changed to switching off in the next state if and only if the user wants to change the mode and selects the OFF button for that.

(12) LTLSPEC g((switchMode=true and button=MANUAL) iff x(btn=MANUAL)).
This property informally means that the system's operation mode is always changed to manual delivering in the next state if and only if the user wants to change the mode and selects the MANUAL button for that.

The properties from number 3 to 12 are satisfied by the AsmetaL specification for the IPCS, but the first two properties are not. As a result, he AsmetaL specification for the IPCS needs to be modified with respect to just the first two properties, such that the r_Manual rule must be modified to include a check on whether the available insulin is graeter than or equal to the current manual dose. In addition, the r_Running rule must also be modified to add the statement of providing the alarm action alarmCommand=true when the iA $\leq$(4$\times$(maximum single dose=4)).

5.4.2.6 Evaluation of the IPCS Case Study

To evaluate the IPCS case study, we can compare our methodology results with the results in [220]. Our methodology starts with specifying the system via AsmetaL, while the work in [220] employs the Z language for specification. Our specification tries to represent the timing aspects for the system via using an abstract time representation, while the [220] specification uses the input variable clock? to obtain the current time, but it does not specify how the implementation of RUN and TEST schemas responds to this variable. In our methodology, we use the simulation, validation and verification tools to develop a safe system, whereas [220] utilizes the safety arguments method for performing manual verification. This method starts with an unsafe state, then all paths in the system code must be proven to be contradictory to this state. This method does not address the unsafe conditions determined by our methodology, which includes: (1) The patient does not take the automatic dose when the sum of the computed dose and the cumulative dose equals the maximum daily dose. (2) The system can deliver a manual dose even if it exceeds

the available insulin. (3) The system does not give an alarm if the available insulin in the reservoir is less than the sum of 4 maximum single doses. (4) The system resumes its work again when the insulin reservoir is returned back, after filling it with 100 units, even when there is a failure in the hardware unit. We believe that these unsafe conditions are not highlighted by other methods.

5.5 Evaluation

In this section, we evaluate the ASM-STPA-SA methodology. We evaluate our methodology by comparing the formalization process for the STPA requirements of [19] and ours. In [19] four types of safety requirements have been elicited and formalized, which are:

- The control action must always be provided at the next state (without being too early or too late), when a specific combination occurs. It has been formalized as follows:

$$\Box\ (Com_{ij} \rightarrow \bigcirc\ (CA_i)) \tag{5.2}$$

  where: CA_i is the ith hazardous action, and Com_{ij} is the jth combination that relates to the ith action. Such formulae are formulated for each combination of conditions presented in a line of context table with yes answer in 'Not Provided' column.

- The control action must always be provided not later than a certain combination occurrence. This requirement is elicited according to the combination line with yes answer in 'Provided too late' column of the context table. The corresponding safety property is formulated as:

$$\Box\ ((Com_{ij} \rightarrow CA_i) \wedge \neg(Com_{ij}\ U\ CA_i)) \tag{5.3}$$

  The authors of [19] claim that this formalization of the requirement "the software controller should always (...) not provide a control action CA_i too late while the occurrences of the critical set of combinations has become previously true in the execution path.". However, a simple semantics analysis does not support their claim. Indeed, the right hand side of conjunction ensures that either (1) there is no action occurred, or (2) there should be an action occurred such that at some point before

that a combination should not hold, which is different from the statement of the claim.

- The control action must always be provided not earlier than the occurrence of a combination. This requirement is specified according to the combination line with the yes answer in 'Provided too early' column of the context table. The corresponding safety requirement is formulated as:

$$\Box\ ((CA_i \rightarrow Com_{ij}) \wedge \neg(CA_i\ U\ Com_{ij})) \tag{5.4}$$

The authors of [19] claim that this formalization of the requirement "a software controller should always (...) not provide control action CA_i before the occurrence of critical combinations set (...) still not become true in the execution path and that it well provides the CA_i when the combination of (...) holds. ". However, a simple semantics analysis does not support their claim. Indeed, the right hand side of conjunction ensures that either (1) there is no combination satisfied, or (2) there should be a combination such that at some point before that a control action should not occur, which is different from the statement of the claim. Furthermore, the left hand side of conjunction can not be ensured when the control action emerges from more than one combination.

- The control action must always not be provided, when a combination occurs. It has been formalized in the following form:

$$\Box\ (Com_{ij} \rightarrow \neg CA_i) \tag{5.5}$$

This formalization is formulated for each combination line with yes answer in 'Provided' column of the context table.

In our approach all these requirements are captured by a single formula (5.1):

$$\Box((com_{i1} \vee com_{i2} \vee ...com_{in}) \leftrightarrow \bigcirc(CA_i))$$

With this property, the action will not be provided with another combination or later/earlier than satisfying the determined combination.

5.6 Related Work

Here, we discuss related work that uses hazard analysis techniques, formal methods, or both for analysing safety-critical systems. In [229], an integrated approach for combining the results of Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA) techniques into the requirements specification. The FTA results are the identification of combinations of component failures, while the FMEA identifies the failure modes and the minor errors that lead to component failure. That paper uses statecharts to bridge the semantics gap between the results of safety analysis and software requirements. In [204], a method for formalizing and verifying the safety requirements elicited by the FTA technique, is presented.

The safety analysis techniques that have been used for eliciting safety requirements in these papers rely mainly on component failure, and only partially on unintended interactions between system's components. Leveson [126] presents the STPA technique to identify the safety requirements for inadequate control actions that affect whole system functions and its components' behavior. According to STPA, the accidents do not simply arise from sequences of component failures, rather, they arise when the safety constraints related to the functional interactions among system components are not enforced. In [17, 19] the authors propose a software safety verification methodology based on the STPA technique to elicit the safety requirements and verify them to identify software risks. First, they elicit and formalize the STPA requirements (with respect to providing and not providing actions) into LTL properties and they verify them based on an SMV manual constructed model. Next they formalize the STPA requirements (with respect to providing actions too early and too late), and they build a safe behavior model of a software controller constrained by the STPA results with UML statechart, as well as they provide an algorithm to transform the safe model into an input model of the NuSMV model checker. However, the formalization process does not reflect the requirements for too early/late actions. In our work, we reformulate the four STPA requirements ('provide', 'provide too early', 'provide too late', and 'not provide') into one formula capturing these requirements, and we exploit ASMs to model the functional behavior of the system and we do not constrained the ASM model by STPA results. We choose ASM method as it supports several characteristics, including: flexibility in modelling any algorithm at an appropriate level of abstraction, and feasibility of being used in an automatic and tool supported manner during the system development process. Furthermore, ASMs have simple and well-defined formal semantics [64].

In [234] a Structured Object-Oriented Formal Language (SOFL) is adopted to build a formal specification for the IPCS. That paper shows that the SOFL provides an effective means to allow the developer to take a gradual process to build a formal specification for the system, but it does not show how to verify or validate the resulted specifications.

In [30, 33], it is shown how the ASM method serves in supporting the design, validation, and verification activities within the ASMETA framework. However, in this work the verification of safety requirements is guided only by the modeller experience, not by a safety analysis technique. Our approach utilizes the same framework (ASMETA) for developing systems, but we employ the STPA procedure for deriving the safety requirements.

Recently, the paper in [125] presents a methodology for combining the STPA technique with the Event-B method to analyse critical systems. The work in [125] couples the STPA with an intruder modelling concept to generate safety and security requirements which are then validated by applying the Event-B formal method. An intruder modelling concept is stated by identifying the main intent and actions for the intruder and parts of the system that are accessible by the intruder. This identification helps to determine the requirements for mitigating the intruder's behavior.

5.7 Conclusion

In this chapter, we combine the ASM method and STPA technique in a development methodology. Our methodology shows how the simulation-based validation, the scenario-based validation of the functional requirements, and the verification of the STPA requirements help us to modify the ASM specification.

We have demonstrated how the disjunction and if and only if operators adopted in our formalization help to capture the four STPA requirements.

The application of our methodology has been illustrated through the TDC and IPCS case studies.

We have shown how the timing aspects for the IPCS have been modelled in an abstract manner. We modelled the start point of the controller activities via using two controlled functions mC10 and sC30, and we modelled the time passing since last activity by a boolean monitored function pas. This abstract handling specifies when the activity starts but it ignores dealing with durative action, while a certain activity is performed, e.g. run alarm for 10 seconds during running activity. However, we can specify the durative actions using a monitored function similar to the pas function, but this is at the cost of making the

model more intricate.

In the specification analysis presented here, we did not consider the static analysis for the completeness and consistency properties. This by applying the AsmetaMA tool [32] to the specification.

With respect to the applicability of our methodology, it is important to show which system our methodology is applicable to. Based on the generality feature of ASM and as the first stage of our methodology implies using this method , there is no restrictions for selecting the system under analysis, except for the real-time system. While during the stage at which our methodology requires applying the STPA technique, the system under analysis must contain a controller which issues actions that any failure in their functions could lead to a hazardous state.

The stage of eliciting STPA requirements in our methodology is performed manually. Although an automatic tool has been proposed to achieve this [18], it seems only to work for up to 6 variables in a process model for the software controller. Hence, further work needs to address the automation of this stage.

Chapter 6

Security Protocols Analysis Methodology

6.1 Introduction

In chapter 5, we introduced a methodology to develop safe designs of safety-critical systems relying on the idea of combining the formal methods via ASMs with a safety analysis technique. This combination aims at verifying that the ASM model developed for a system satisfies the STPA outcomes which are the answers of the informal STPA queries. This chapter considers a similar combination but in a way that serves the security protocols domain.

As mentioned in Chapter 3 the STPA queries, whose main concern is timing provision, will help to address possible unsafe aspects of the system behaviour. Despite that, they are insufficient to fully address the insecure aspects related to the security protocols. Clearly, the timing provision of messages is not the only factor that affects the security of protocols. Furthermore, analysing the time factor gives an abstract insight into the presence of an insecure state, without precisely uncovering it, e.g., receiving a message too late implies a protocol flaw, but this does not give an insight into what type of flaw is. Recall from chapter 4 the reviewed formal analysis approaches for security protocols depend on the intruder Dolev-Yao model [91], which assumes the following:

1) the intruder can generate any message it wants basing on its knowledge;

2) the encryption mechanism is unbreakable without knowing the related decryption

key; and

3) the intruder can launch an unbounded number of sessions.

The first assumption may lead to some superfluous messages with wrong format or type, which are likely to be rejected by honest participants. The second assumption may be too strong in some contexts. In particular, some encryption mechanisms when they have certain algebraic properties can be broken [141], for instance, commutative encryption [166]. The third assumption and also the first one generally lead to the non-terminated verification problem due to unbounded state space to be analysed. Some protocol verification tools which are based on the general purpose model checkers treat the infinite state space problem by imposing limitations on the number of executed runs and the number of generated messages. Though, when the protocol under analysis is relatively complex, e.g., it consists of more than six actions, or its model includes a range of variables, this can result in a state explosion problem, which in turn will inhibit the analysis process termination. Other verification tools, e.g., ProVerif and Tamarin [51, 168], have addressed the state space explosion problem but at the cost of possible non-termination of the analysis, particularly in the presence of non-trivial algebraic properties, such as the one of commutative encryption. The work in [129] deals with the infinite state space problem by simulating security protocols using scenarios designed for known attacks. This approach helps to reduce the number of protocol runs needed to be explored with guaranteed termination. However, the intruder in work [129] constructs messages using only a type matching restriction. This still may increase the message space with unacceptable messages by the receiver, which eventually increases the number of protocol runs. Furthermore, in work [129], the user needs to simulate all specifications for the designed attack scenarios while looking for attacks including those that will definitely not compromise the protocol under analysis. Moreover, this work does not show how to analyse a protocol in the presence of an algebraic property.

Taking into account the above limitations, we propose a methodology for analysing security protocols. It consists of two phases: *manual analysis*, and *automatic analysis* phases. For the manual analysis phase, we develop the Flaws and Attack Types Identification (FATI) method to help at selecting the attack scenario specification whose simulation is likely to produce an attack. The FATI is STPA-like method which is based on a set of developed queries whose answers are capable of capturing the main flaws for security protocols. At the automatic analysis phase, we implement the attack pattern scenarios idea in

the ASM methodology. We also further reduce the number of protocol runs by modelling a "clever" intruder who generates only acceptable messages restricted by the supposed message type format and the expected message content. Furthermore, we consider the commutative property.

The main contributions of this chapter are:

1) An STPA-like method that enable us to simulate only the attack scenario specification whose simulation is likely to produce an attack.

2) Formulating a principle for generating messages by the intruder according to the receiver's expectations taking into consideration the message type format and the expected content.

3) Using the attack pattern scenarios in the actual verification/validation process based on simulation in the ASM methodology.

4) Modelling commutative encryption and analysing protocols in the presence of this algebraic property.

The rest of this chapter is organized as follows: Section 6.2 details the FATI method developed for attack types identifications. Section 6.3 introduces our methodology with respect to its two phases and discusses the aspects of its second phase in details, including modelling of the protocol, the intruder, the attack scenarios, as well as specifying the invariant conditions. Results and discussion are discussed in Section 6.4. While, Section 6.5 reviews related work. Finally, Section 6.6 concludes the chapter.

6.2 Flaws and Attack Types Identification Method

In this section, we present the Flaws and Attack Types Identification (FATI) method to identify potential flaws of protocols and the expected attack types upon them.

The FATI technique adopts the principles of two key features for the STPA technique. The first one is using a set of *what-if* queries on provision conditions to each action in order to identify system flaws. This feature can be utilized for security protocols, as any protocol is a sequence of actions that the *what-if* queries are applicable to them. The second feature relates to considering the system failing components together with the dysfunctional interactions among non-failing components as causes of system accidents. Likewise, the

components of any security protocol are the participants involved in it whereas the exchanged messages between participants represent the interactions. Based on that, possible attacks underlie any functional deficiency in a protocol that is related to the participants and/or to their interactions (messages).

According to the above two features, we developed the FATI method which proceeds by asking queries about each protocol action in a way that facilitates flaws to be recognized, and in turn be assessed collectively to show which type of attacks could exploit these flaws. In the light of attacks reviewed in Chapter 4 (except the type-flaw attack), we realized that queries based on timing provisions would not directly address flaws that lead to these attacks. For example, in a protocol, when a participant receives a message too early (before the required time for constructing and sending this message) this may give an indication that there is a flaw in this protocol, however true, but this does not give an insight into what the flaw is, or from where it arose. Furthermore, early, late, or non-received messages do not necessarily imply protocol flaws, rather, they may refer to flaws in the communication medium. This spotlights the need for specific queries that can identify flaws related to the protocol itself.

We were capable of identifying flaws by considering queries about the *liveness* of protocol's participants and about the *freshness* of messages sent by the participants. These queries are deemed to be flaw identifiers due to the following reasons. First, security protocols mostly aim at authenticating their participants and/or distributing secrets between them. Second, the attacks reviewed in chapter 4 intend to compromise the authentication and/or secrets distribution aims. The answers to these queries are according to the receiver's perspective. We developed guidelines to facilitate answering these queries and to determine the attack types.

The process of the FATI technique consists of two steps:

Step 1 Identify the possible flaws that affect the protocol functionality through performing the next sub steps:

Step 1.1 Apply the following queries to each protocol's action:

Participant Liveness Is the sender of a message related to this action alive (truly participated) from the receiver's perspective? Our guideline to answer this query is to check whether the sender sends an encrypted message containing a freshness component generated by the receiver, and this message is encrypted either with a shared key between the sender and the

receiver, or public key of the receiver.

In case that the sender sends an encrypted subcomponent asking the receiver to pass it to another participant, the liveness of the first sender from the later participant's perspective must also be enquired about.

Message Freshness Is the message related to this protocol action fresh?. Our guideline regards a message fresh when every freshness component in this message is fresh and its association is clearly shown. The freshness of a component can be deduced from its existence in a message encrypted under a receiver's public key or shared key between the sender and the receiver, and this component is generated originally by the receiver. While the association of a freshness component can be deduced either from attaching the freshness component to the identity of its generator or from associating the freshness component to the key used to encrypt the message that contains this component. In the case that there is more than one freshness component in an encrypted message, then the association of one of these components must be deduced by the key while the rest by their generators' identity.

Step 1.2 Document the answers of the above queries in a table, called identified flaws table, consists of columns for the index of action, the sender, the receiver, the message components, the queries, and comments. The comments column is for illustrating the reasons of the negative answers.

Step 2 Determine the expected types of attacks that can exploit the identified flaws. The determination is directed by the following guidelines:

Simple_REPL Attack Guideline. We expect a protocol is vulnerable to the Simple_REPL attack when the following conditions are satisfied. First, there is a message which the liveness of its sender is not proved. Second, the message in the first condition comes after two successive messages with proved their senders' liveness. Last, all the messages in the first and second conditions are between the same two participants.

DoS_REPL Attack Guideline. The DoS_REPL attack arises when both two following conditions are met:

- The liveness of the initiator for a protocol is not proved for all its sent

messages.

- The first message sent to the responder whether from the initiator or the server is not fresh, and this message is intended to authenticate the initiator to the responder.

REFL Attack Guideline. The REFL attack usually occurs when:

- The liveness of the initiator is absent for all its sent messages.
- All the messages been sent to the initiator are not fresh.

MITM Attack Guideline. It is likely to occur when the initiator sends a non-fresh message to the responder, and the responder's replay to this message is not fresh as well.

INTRL Attack Guideline. It is usually against protocols with repeated authentication part, more precisely, this attack occurred when there is, in the first authentication part, an encrypted message or sub-message containing freshness component and this message is being re-sent in the second authentication part, i.e., this message or sub-message is not fresh.

6.2.1 Application of the FATI Method to the AS_RPC Protocol

In this section, we illustrate the FATI method by applying its steps to the AS_RPC protocol. We identify the possible flaws of the AS_RPC protocol by applying the liveness and freshness queries, and we document the answers to these queries in Table 6.1. By assessing the results in Table 6.1, we conclude that the AS_RPC protocol may be subject to the Simple_REPL attack. This is because B does not prove its liveness to A in the fourth action, and this action comes after two successive ones with established sender's liveness.

Action Number	Message Context			Queries		Comments
	Sender	Receiver	Message Components	Liveness	Freshness	
1	A	B	$A, \{N_A\}_{ssk(A,B)}$	No	No	The liveness of A from B's perspective is not proved and this message may not be fresh as N_A is not generated by$B$
2	B	A	$\{N_A + 1, N_B\}_{ssk(A,B)}$	Yes	No	The freshness of this message is not proved as the association of N_B is not shown i.e., it is not associated with its generator's identity
3	A	B	$\{N_B + 1\}_{ssk(A,B)}$	Yes	Yes	
4	B	A	$\{ssk^{'}(A,B), N_B^{'}\}_{ssk(A,B)}$	No	No	The liveness of B from A's perspective is not proved as there is no freshness component generated by A, and this message is not fresh since $\{ssk^{'}(A,B)$ and $N_B^{'}$ are not generated by A and the association of $N_B^{'}$ is not shown i.e., it is not associated with its generator's identity

Table 6.1: The identified flaws table for the AS_RPC protocol

6.2.2 Application of the FATI Method to the NSPK Protocol

In this section, we illustrate the FATI method by applying its steps to the NSPK protocol. We identify the possible flaws of the NSPK protocol and we show the identification results

in Table 6.2. By evaluating the results in Table 6.2, we conclude that the NSPK protocol may be subject to the MITM attack. This is because A sends a non-fresh message to B (the first message) and B's response to this message (second message) is not fresh as well.

Action Number	**Message Context**			**Queries**		**Comments**
	Sender	**Receiver**	**Message Components**	**Liveness**	**Freshness**	
1	A	B	$\{A, N_A\}_{pk(B)}$	No	No	The liveness of A from B's perspective is not proved as there is no freshness component generated by B, and this message is not fresh since N_A is not generated by B
2	B	A	$\{N_A, N_B\}_{pk(A)}$	Yes	No	The freshness of this message is not proved as the association of N_B is not shown i.e., it is not associated with its generator's identity
3	A	B	$\{N_B\}_{pk(B)}$	Yes	Yes	

Table 6.2: The identified flaws table for the NSPK protocol

6.3 The proposed Methodology for Analysing Security Protocols

This section presents a methodology for analysing security protocols and developing secure designs for them. It extends the simulation based attack scenarios method in [129] via:

1) avoid wasting unnecessary efforts in simulating all the specified attack scenarios by prior identification of the expected attack types;

2) presenting our implementation in ASM methodology;

3) modelling a "clever" intruder that will generate acceptable messages only; and

4) considering the commutative property of encryption mechanism during protocol analysis.

Our methodology assumes the following:

(1) There is a specific format for each message, and the honest participants only accept the messages conforming with this format.

(2) The intruder is presumed to be as powerful as possible, where it can (a) intercept and eavesdrop any sent message; (b) generate new messages according to its knowledge and the allowed messages' format and content; (c) decrypt an encrypted message when it knows the decryption key or when the execution of a protocol depends on an algebraic property of the encryption method.

Figure 6.1 shows an overview of our methodology which includes the following phases:

(1) **The Manual Analysis Phase.** This phase is mainly aimed at avoiding the waste of effort unnecessarily for simulating all the specified attack scenarios, that the majority of will not lead to attacks. It is worth to carry on a simple manual analysis that helps to identify the possible attack types before undertaking the simulation task. As a result, this phase applies the procedure of the FATI method to produce the types of expected attacks against the protocol under analysis.

(2) **The Automatic Analysis Phase.** This phase is devoted to the accomplishment of analysis for security protocols in an automatic way. It is made up of two stages:

   (a) **The Formal Specification Stage.** This stage represents the base for the next stage. It comprises four aspects:

   - **The Protocol Aspect:** represents a modular model for any protocol allowing to add different participants required for it.
   - **The Intruder Aspect:** represents a general model for intruder behaviour, taking into account updating knowledge and generating message capabilities.
   - **The Attack Scenarios Aspect:** stands for the built-in specifications of attack scenarios, which are inspired by [129], into the intruder model to sketch its behaviour. The specifications are used according to their related attack types which are identified by the first phase.
   - **The Invariant Security Properties Aspect:** represents the specification of the invariant constraints that must be met by the resulting protocol model.

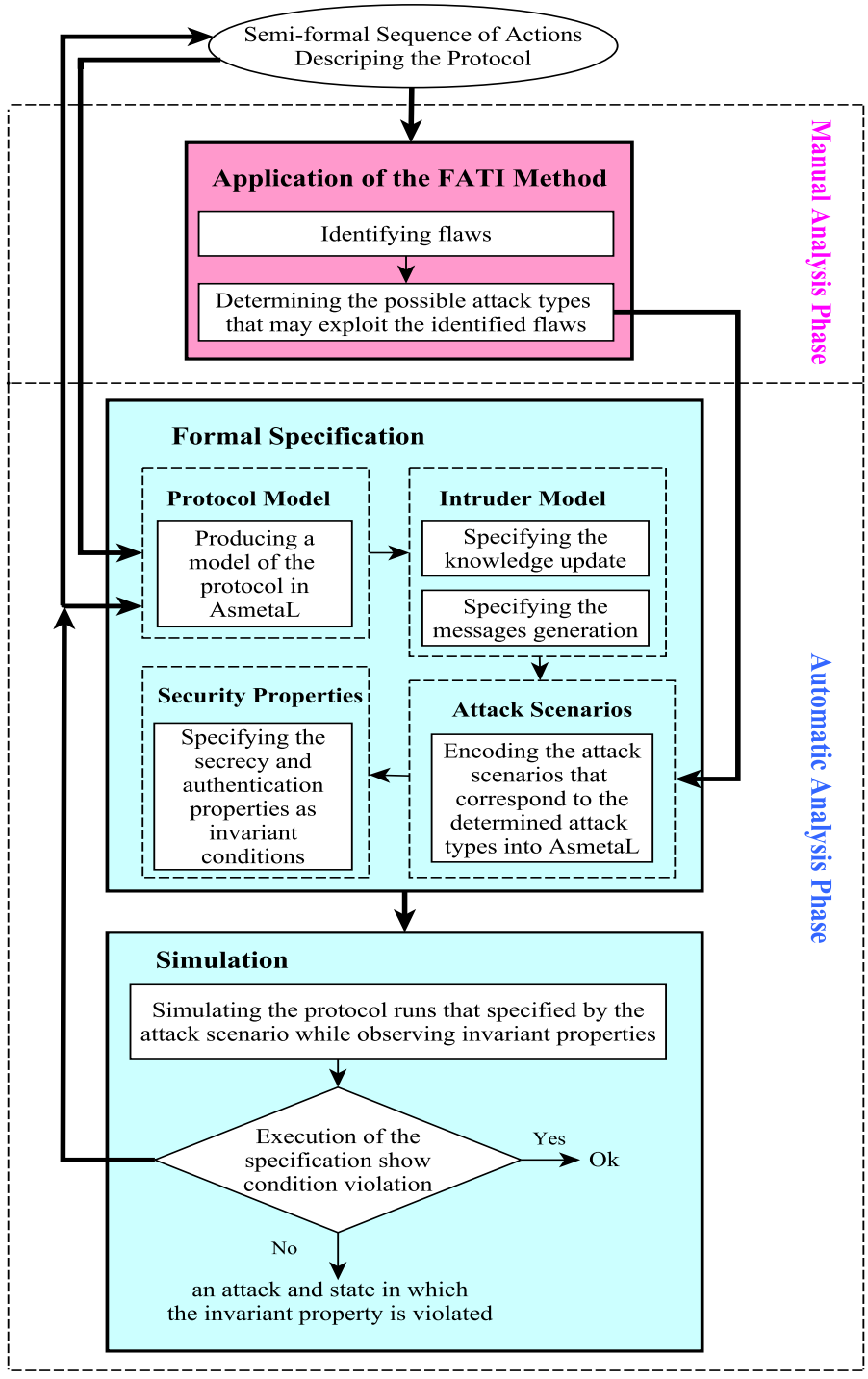

Figure 6.1: The safe design development methodology

For any protocol, the intruder and the attack scenario aspects can be used directly without any amendment, while the rest can be used only after a simple modification based on the protocol's requirements. Each aspect is illustrated in details in Sections 6.3.1-6.3.4.

(b) **The Simulation Stage.** It employs the AsmetaS tool for simulating the obtained specifications from the previous stage. This simulation produces the protocol runs which are specified by an attack scenario and shows whether there is a violation of the invariant properties or not. In the violation case, we can go back to our protocol specification to correct it.

In the following subsections, we detail the aspects of the formal specification stage.

6.3.1 The Protocol Aspect

This aspect represents modelling the protocol participants and their main operations needed for message construction as AsmetaL rules.

6.3.1.1 The Protocol Participants

In this section, we model the NSPK protocol in AmetaL. We define the domain Protocol={*NS, TP, DS, RPC, KSL*} to enumerate the protocol names, note that *NS* refers to the NSPK protocol, and *RPC* refers to the AS_RPC protocol. As each participant has a specific role, we construct a rule that corresponds to each role, such as *r_Initiator* rule for the initiator and *r_Responder* rule for the responder, see Code 6.1 and Code 6.2, respectively.

The first step to build the rules for protocol participants is defining the domain Id={*AA, BB, II*} that represents the set of identities for the participants shared in the NS protocol, including the intruder.

The second step is stating the protocol's control flow, e.g., identifying when to send a message. Each participant can be in two states: *SEND* and *RECEIVE* for sending and receiving a message, respectively. The protocol's session starts when the initiator wants to initiate, i.e. when the Boolean function *wantToInit(Id)* is *true*. Each message has a number stored in the *msgNo* controlled function (a controlled function is equivalent to a variable in standard programming languages and a static function is equivalent to a constant). Similarly, the running session number is stored in the *sNo* function. To indicate

```
rule r_Initiator($InId in Id, $ResId in Id)=
 if state=SEND then
  par
    if wantToInit($InId) and msgNo=1 then
      seq
        start(NS, sNo, 1):=true
        r_GenerateNc[]
        ncInt($InId, $ResId):=toString(n)
        r_Encrypt[[toString($InId), n], pk($ResId)]
        r_Send[[toString(cypher)], $InId, $ResId]
        wantToInit($InId):=false
      endseq
    endif
    if msgNo=3 and start(NS, sNo, 3)=false then
      seq
         start(NS, sNo, 3):=true
         r_Encrypt[[at(plainText, 1n)], pk($ResId)]
         r_Send[[toString(cypher)], $InId, $ResId]
      endseq
    endif
 endpar
 else
   if inBox($InId)!=undef and msgNo=2 and finish(NS, sNo, 2)=false then
    seq
      r_Decrypt[first(inBox($InId)), prk(pk($InId))]
      inBox($InId):=undef
      if plainText!=undef and
         ncInt($InId, $ResId)=first(plainText) then
            finish(NS, sNo, 2):=true
      endif
    endseq
   endif
 endif
```

Code 6.1: The r_Initiator of the NSPK protocol

a message related to a particular session is about to be sent, the *start* function is set to be *true*. To indicate a message has been received, the function *finish* is set to *true*.

The next step shows how to define and deal with each message. The message is defined as a sequence of Strings. To reach a specific element in this sequence, we use the predefined function in the standard library of AsmetaL which is *at*. This function takes as inputs a sequence of a determined domain and an index of the element that we want to reach, and it returns the desired element. The participant realizes that there is a received message when its inbox has a message. Whereas a constructed message is sent by updating the *outBox* function with this message. To build a message, the participant can generate a nonce by the *r_GenerateNc* rule or encrypt a message by the *r_Encrypt* rule or decrypt it by the *r_Decrypt* rule.

```
rule r_Responder($ResId in Id)=
  if state=RECEIVE and inBox($ResId)!=undef then
    if (msgNo=1 and finish(NS, sNo, 1)=false) or
       (msgNo=3 and finish(NS, sNo, 3)=false) then
      seq
        r_Decrypt[first(inBox($ResId)), sk(pk($ResId))]
        if plainText!=undef then
          if msgNo=1 then
            choose $id in Id with first(plainText)=toString($id) do
              par
                finish(NS, sNo, 1):=true
                claimed_Init:=$id
              endpar
          else
            if msgNo=3 then
              if first(plainText)=toString(ncRsp($ResId, claimed_Init)) then
                finish(NS, sNo, 3):=true
              endif
            endif
          endif
        endif
        inBox($ResId):=undef
      endseq
    endif
  else
    if (msgNo=2) and start(NS, sNo, msgNo)=false and finish(NS, sNo, 1)=true then
      seq
        start(NS, sNo, msgNo):=true
        r_GenerateNc[]
        ncRsp($ResId, claimed_Init):=n
        let ($nb=toString(ncRsp($ResId, claimed_Init))) in
          r_Encrypt[[at(plainText, iton(1)), $nb], pk(claimed_Init)]
        endlet
        r_Send[[toString(cypher)], $ResId, claimed_Init]
      endseq
    endif
  endif
```

Code 6.2: The r_Responder of the NSPK protocol

6.3.1.2 The Operations Performed by the Participants

We model the encryption and nonce generation operations in an abstract way, by employing the *extend* construct that expands an abstract domain for the operation outputs with a new element. For instance, we define the abstract domain *Nonce* to represent a universe for all nonce values, and the generated nonce will be an element returned by the *extend* rule, see Code 6.3.

```
rule r_GenerateNc=
  extend Nonce with $n do
    n:=$n
```

Code 6.3: The r_GenerateNc rule

The specification for encryption and decryption operations utilize a similar idea, see the specification for these operations in Code 6.4.

```
dynamic abstract domain Cipher
dynamic abstract domain Key
enum domain Id={AA | BB}
controlled plainText: Seq(String)
controlled cipher: Cipher
controlled key: Cipher-> Key
controlled plain: Cipher-> Seq(String)
static pk: Id-> Key
static sk: Key-> Key
static ssk: Prod(Id, Id)-> Key
static pKA: Key
static pKB: Key
static sSKAB: Key
function pk($id in Id)=
  switch $id
    case AA: pKA
    case BB: pKB
  endswitch
function ssk($id1 in Id, $id2 in Id)=
 if (($id1=AA) and ($id2=BB)) or (($id1=BB) and ($id2=AA)) then
    sSKAB
 endif
function sk($k in Key)=
  switch $k
    case pKA: sKA
    case pKB: sKB
    case sSKAB: sSKAB
  endswitch

rule r_Encrypt($m in Seq(String), $k in Key)=
 choose $c1 in Cipher with (plain($c1)=$m and key($c1)=$k) do
     cipher:=$c1
 ifnone
   extend Cipher with $c2 do
     par
         cipher:=$c2
         plain($c2):=$m
         key($c2):=$k
     endpar
rule r_Decrypt($c in String, $k in Key)=
  choose $c1 in Cipher with (toString($c1)=$c and $k=sk(key($c1))) do
     plainText:=plain($c1)
  ifnone
     plainText:=undef
```

Code 6.4: The encryption and decryption specification

In Code 6.4, first, we introduce the specification signature, which is as follows:

- The Cipher is an infinite domain for the ciphertext
- The Key is an infinite domain for the cryptographic keys.
- The unary function key represents the key for a given Cipher element.

- The unary function plain is a sequence of Strings that stands for the plaintext of a given *Cipher* element.
- The *cipher* function is a sequence of String representing the encryption output.
- The function ssk represents the shared key between two participants.
- The function pk is the public key for a participant.
- The function sk returns the same given key if it is a symmetric one, while if it is a public key, it returns a corresponding private key. This has the advantage of making our specification applicable to both public and symmetric key cryptographic paradigms.

After the signature, we specify two rules: the r_Encrypt rule for converting the presented plain message into an encrypted one using the determined key, and the r_Decrypt rule which transforms the encrypted message into a plain text one using a specific decryption key. The r_Encrypt rule, firstly, makes sure that the given message has not been encrypted before by choosing an element in the Cipher domain, such that the plain text for this element is equal to the given message. This element represents the encrypted form of the given message. When choosing an element returns nothing (the presented message has not been encrypted previously), this rule will generate a new ciphertext (encryption outcome), given its plain text and key, by extending the Cipher domain.

While the r_Decrypt rule chooses from the Cipher domain an item that is equal to the given ciphertext (encrypted message), and when the sk function applied to its key, it will yield a key identical to the given decryption key. If the item was found, the rule would return the item's plaintext; otherwise, an undefined decryption output would not be returned.

In addition to the specification of public and symmetric key encryption, we can specify the algebraic property of the encryption function, such as commutativity. As mentioned in Chapter 4, an encryption function is called commutative if it satisfies the following condition: $\{\{m\}_{k_1}\}_{k_2} = \{\{m\}_{k_2}\}_{k_1}$, where k_1 and k_2 are encryption keys. Notice that ASM specifications are abstract *imperative* programs and do not allow to simply state commutativity property in declarative way. It rather has to be supported by the imperative executable semantics of ASM rules.

The specification for the commutative encryption is presented in Code 6.5. The declarations of functions used in the signature for this code are similar to those in Code 6.4,

except the key function declaration, which is defined as a set of keys to keep track of all the keys that are used to encrypt a given message. The encryption function is implemented with respect to its commutative property by including or excluding the encryption key from the keys set which must initially be empty.

```
dynamic abstract domain Cipher
dynamic abstract domain Key
controlled key: Cipher-> Powerset(Key)
controlled plain: Cipher-> Seq(String)
controlled cipher: Seq(String)
rule r_GenNew($n in Seq(String),$k in Powerset(Key))=
  extend Cipher with $e do
    par
      cipher:=[toString($e)]
      plain($e):=$n
      key($e):=$k
    endpar
rule r_comEncrypt($m in Seq(String), $k in Key)=
 choose $c in Cipher with length($m)=1 and toString($c)=first($m) do
  if contains(key($c), $k) and isEmpty(excluding(key($c), $k)) then
        cipher:=plain($c)
  else
   if contains(key($c), $k) then
    choose $c1 in Cipher with plain($c1)=plain($c) and
                    key($c1)=excluding(key($c), $k) do
         cipher:=[toString($c1)]
    ifnone
     r_GenNew[plain($c), excluding(key($c), $k)]
   else
    choose $c2 in Cipher with plain($c2)=plain($c) and
                    key($c2)=including(key($c), $k) do
       cipher:=[toString($c2)]
    ifnone
      r_GenNew[plain($c), including(key($c), $k)]
   endif
  endif
 ifnone
   choose $c3 in Cipher with plain($c3)=$m and key($c3)=including({}, $k) do
     cipher:=[toString($c3)]
   ifnone
      r_GenNew[$m, including({}, $k)]
```

Code 6.5: The commutative encryption specification

After the signature we specify two rules: the r_GenNew and the r_ComEncrypt. The r_GenNew rule generates a ciphertext for a given message by extending the Cipher domain with one element. The *r_ComEncrypt* rule is responsible of returning the encryption output that satisfies the commutative property, by examining the following conditions:

1. If the message that we want to encrypt is *plaintext* then either a new encryption output is returned by r_GenNew, if this message has not been encrypted before, or

the previous calculated encryption output is returned.

2. If the message that we want to encrypt is *ciphertext* (encrypted) then there are two situations:

   (a) The encryption key *does not belong to* the set of keys for this ciphertext. In this case, the encryption key will be included in the set of keys, and the Cipher domain is checked if it contains an element with a set of keys equals to the obtained one, then this element represents the encryption output; otherwise the r_GenNew rule is called.

   (b) The encryption key *belongs to* the set of keys. Then, the encryption key will be excluded (deleted) from this set. Later, the resultant set will be checked if it is an empty set, then the plaintext for the given ciphertext is returned as an encrypted output, else the Cipher domain is checked to obtain an element with the same obtained set of keys, otherwise the r_GenNew rule is called.

6.3.2 The Intruder Aspect

In the classic Dolev-Yao intruder model [91], the intruder can generate and send any message based on its knowledge. This can lead to an issue of the growing message space with redundant messages that may be rejected by the receiver. We address this issue by only considering the generation and sending messages that have contents and type format based on the receiver's expectations. For example, sending $\{N_A\}_{pk(B)}$ as a third message in the NSPK protocol, will be rejected by the responder, because it expects that it will receive an encrypted nonce which has a value equal to the one sent before.

To do that, we first define an enumerative domain, called Type, which contains the possible message's component types:

Type=$\{ID, NC, PUBK, SECK, SHRK, ENCR, ANY\}$

where:

- *ID* is an identity;
- *NC* is a nonce;
- *PUBK* is a public key;
- *SECK* is a private key;

- *SHRK* is a symmetric key;
- *ENCR* is an encrypted component; and
- *ANY* is any component.

Next, we represent every protocol's message as a syntax tree to capture the analysis performed by the intruder. By the analysis, we mean either decomposing the message into its components, or constructing a new message based on these components. The tree representation is supported by the following two definitions which are illustrated in Figure 6.2.

Definition 1. *Let m be a message which is a sequence of components, then the syntactic tree of this message has the following properties:*

1. *its root is labelled with the message itself, while its other nodes are labelled with the message's sub components;*
2. *a node in this tree, which is a sequence of more than one component, has a number of children;*
3. *the node, which is a sequence of one encrypted component, has one child which is a sequence of its unencrypted components;*
4. *its leaves are single sequences of unencrypted component;*
5. *the root is numbered with a number equal to message number;*
6. *the children for each node are assigned a left-to-right ordering numbers, concatenated to the parent number.*

Definition 2. *Let T_i be a tree for a message that has number i, $i = \{1, ..., n\}$. The position of the root node for T_i is a single sequence containing only i. The position of other nodes is the position of the parent node concatenated with the child order number for this node.*

The last step is to update the intruder's knowledge with the eavesdropped messages, and to generate the possible accepted messages, based on the messages' tree representation. This step is detailed in the following subsections. Note that, the AsmetaL specifications for updating the intruder knowledge and generating the messages are available in [22].

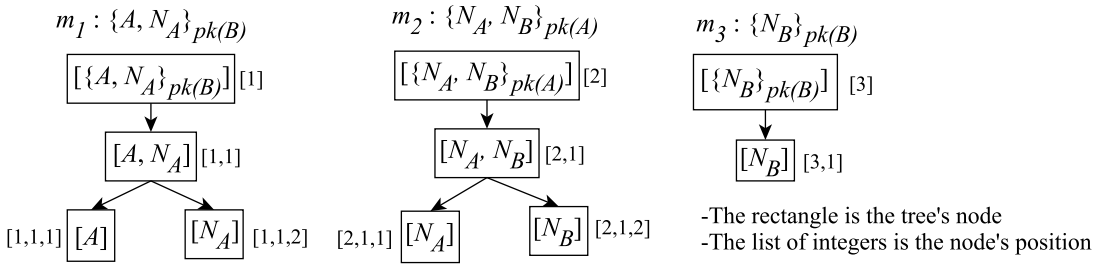

Figure 6.2: Message Tree Representation Examples for the NSPK Protocol

6.3.2.1 Updating the Intruder's Knowledge

To save all the eavesdropped messages and their components, which are sequences of String, in the intruder knowledge, we define the function k: Seq(Seq(String)). This function is updated by the r_UpdateTheKnowledge rule.

Typically, the r_UpdateTheKnowledge rule starts by saving the whole eavesdropped message in the k function. Then it saves each element of this message in k, and meanwhile checks whether the type of this element is encrypted, to decrypt it (if possible) and add the decrypted components to k. The type for each component is specified as a constant for our model by the static function type: Prod(Protocol, Seq(Integer))−> Type. This function yields the type for the node that has a given position in a given protocol. For example, for the third message in NSPK protocol that has two nodes in its tree representation, we have the following type(NS, [3])=[$ENCR$], and type(NS, [3,1])=[NC]. While k is updated, the position of each node will be saved in the following function: position: Prod(Seq(String), Integer, Integer))−> Seq(Integer). This function permits easy access to the desired node of any message, by just giving the node's value, the session number, and the message number.

6.3.2.2 Generating Messages

We restrict the intruder's ability to generate messages by considering the type format of the message and its expected content. By type format, we mean the component types assumed by the receiver, e.g., for the second message in the NSPK protocol, the correct type format would be an encrypted message by a public key. By the expected content, we mean the values of the components that the receiver would expect, e.g., the second message in addition to being encrypted under the receiver's public key, the value of the first component, which is a nonce, inside encryption must be equal to the nonce sent in

the first message for the NSPK protocol. Notice that, in all protocols the first message has no expected content.

Accordingly, we define the following static functions facilitating messages generation: (the first two functions are related to each node in the message's tree representation, while the rest are related to the encryption key)

(1) type: Prod(Protocol, Seq(Integer))−> Type: is previously stated;

(2) posExpVal: Prod(Protocol, Seq(Integer)–>Seq(Integer): returns the position of the expected value (if it exists) for the node that has a given position. For example, the nonce node of the third message for the NSPK protocol, which is at the [3,1] position, has an expected value the same as that at [2,1,2] position, i.e., posExpVal(*NS,* [3,1])=[2,1,2]. As a result, to generate the this message, the k sequence will be checked if it contains a node with a position returned by the posExpVal function.

(3) posKeyId: Prod(Protocol, Seq(Integer)–>Seq(Integer): returns a specific identity for the key of the given encrypted component/node's position;

(4) keyType: Prod(Protocol, Seq(Integer)–>Seq(Type): says what is the type of the key for the given encrypted component/node's position;

(5) posKeyVal: Prod(Protocol, Seq(Integer)–>Seq(Integer): returns the expected key's position for the given position of the encrypted component.

Based on the all above functions, we specify the *r_GenerateMsg* rule for generating the potential accepted messages.

6.3.3 The Attack Scenarios Aspect

As mentioned in Chapter 4, the attack scenarios concept was previously presented in [129], to reduce the number of protocol's runs and to guarantee the termination of the analysis process using simulation. This concept was based on designing generic algorithms for five attack schemes, including MITM, REFL, DoS_REPL, Simple_REPL, and INTRL, to return a scenario of an attack for the given attack scheme and protocol's parameters. The algorithms also need to define an assignment of the participant's identity and the intruder impersonation to each role. At the beginning, the produced scenario is written in Common Syntax [132] and then is translated into an Estelle specification [65], to return the

concrete simulated scenario. Despite that this work helps to reduce the protocol's runs, the intruder in this work generates messages based only on matching the types of message's components, and this can still increase the runs with unacceptable messages by the honest receiver. Furthermore, in this work, the efforts are wasted in simulating all the attack scenarios even those whose simulation will definitely not lead to an attack, and hence the simulation needs to consider only the attck scenarios that will likely lead to producing attacks.

In our work, we apply the FATI method to the protocol under analysis in order to identify the possible attack types upon it, and hence this gives guidance of which attack scenario that needs to be simulated. We also encode the attack scenarios directly into AsmetaL, and we restrict the messages generation action of the intruder with the expected message content and types as well. Our AsmetaL specification of a scenario for a given attack type, depends on the following static functions, that facilitate the assignment identification:

p: Prod(Integer, Integer, Task) –> Id, with Task={SENDER, RECEIVER}, and

imp: Prod(Id, Integer) –> Boolean

The first function restores the identity for the given task: SENDER, RECEIVER for the participant's role at the given session and message numbers, while the second function says whether the given identity for a participant participated in a given session number is impersonated. The detailed specifications for the five scenarios are available in [22].

6.3.3.1 Man in The Middle (MITM) Attack Scenario

The MITM attack aims to secretly intercept, replay, alter messages between two participants and leaving them to believe that they are only communicating with each other. The scenario for this attack is laid out as follows: an odd message number is put in a sequence of session numbers arranged in the increasing order, while an even message number is put in a sequence of session numbers ordered in the decreasing order. For example, if we have 2 sessions and 3 messages, then the order of steps will be as follows: (1.1)(2.1)(2.2)(1.2)(1.3)(2.3). This scenario is built using one of the three possible assignments illustrated in Table 6.3, which can cause MITM attack. Note that, if a protocol has a third participant playing server role, then this participant will not be impersonated as this attack aims to create a fake conversation between two participants. In Table 6.3, the **No** represents the number of possible assignments (assignment of the participant identities and the intruder impersonations to each participant role). The assignment which has number

2 in Table 6.3 causes the attack on NSPK protocol, which is shown in Figure 4.2(b). All the tables mentioned in this chapter are based on the defined assignments in [129].

No	Odd Sessions			Even Sessions		
	Initiator	Responder	Server	Initiator	Responder	Server
1	A	I(B)	S	I(A)	B	S
2	A	I	S	I(A)	B	S
3	A	I(B)	S	I	B	S

Table 6.3: The possible assignments for MITM attack scenario

```
rule r_ExchangeMsgs($p in Protocol)=
let ($s=p(sNo, msgNo, SENDER)) in
 let ($r=p(sNo, msgNo, RECEIVER))in
   par
     if state=SEND then
       par
        if $s=II or
           imp($s, sNo, msgNo)=true then
             r_Intruder[NS]
        else
           par
             r_Initiator[$p, $s, $r]
             r_Responder[$p, $s]
           endpar
        endif
        state:=RECEIVE
       endpar
     endif
     if state=RECEIVE then
      par
        if $r=II or
           imp($r, sNo, msgNo)=true then
           par
              r_UpdateTheKnowledge[$p]
              sNo:=sNo+dir
           endpar
        else
         par
           r_Initiator[$p, $r, $s]
              r_Responder[$p, $r]
           endpar
         endif
         state:=SEND
       endpar
      endif
    endpar
  endlet
endlet
rule r_MITM_Scenario($p in Protocol)=
  seq
    stop:=totalSno
    begin:=1
    dir:=1
    while msgNo<=totalMsgNo and
            existAttack=true do
     seq
       while ((stop=totalSno and sNo<=stop)
              or (stop=1 and sNo>=stop)) and
                 (existAttack=true) do
            r_ExchangeMsgs[$p]
       dir:=(-1)*dir
       par
         stop:=begin
         begin:=stop
       endpar
       msgNo:=msgNo+1
       sNo:=begin
     endseq
  endseq
```

Code 6.6: The r_ExchangeMsgs and the r_MITM_scenario rules

The AsmetaL specification for the MITM attack scenario is shown in the Code 6.6. In this Code, there are two rules: the r_MITM_Scenario rule, which models the scenario arrangement, and the r_ExchangeMsgs rule, which is called by the r_MITM_Scenario rule to send and receive a message by the participants. In r_MITM_Scenario rule, the totalSno function is a number of sessions, the totalMsgNo function is a number of messages, and

stop, begin, and dir functions help achieving the increasing and decreasing order of the session's numbers.

In r_ExchangeMsgs rule, two situations are in parallel execution. First, when state=SEND, the SENDER's identity is checked, if it is *II*, or it is impersonated, then the intruder will send a message. Otherwise, in parallel the r_Initiator/ r_Responder rules are fired to send the current message by either the initiator or the responder depending on the SENDER identity of this message. Second, when state=RECEIVE, the RECEIVER's identity is checked here, to either update the intruder's knowledge or to receive the current message by the initiator/responder. Sending a message by the intruder can be achieved by generating all the possible messages, and sending these messages one by one until one of them is accepted. If not, then there is no such attack scenario.

6.3.3.2 Reflection (REFL) Attack Scenario

The constructed scenario of this attack, with respect to the pattern of ordering the message and session numbers, is the same as that of MITM attack. While they are different in regard to the assignment of participant identities and intruder impersonations, see Table 6.4. In this table, we can see that the initiator of any odd session will be the responder of the corresponding even session. In addition, the intruder either impersonates both the responder of an odd session and the initiator of the corresponding even session, or it participates as both legal responder of an odd session and legal initiator of the corresponding even session without impersonation. Also, the server is not impersonated at the odd sessions in order to use its messages in the even sessions.

No	Odd Sessions			Even Sessions		
	Initiator	**Responder**	**Server**	**Initiator**	**Responder**	**Server**
1	A	I(B)	S	I(B)	A	I(S)
2	A	I	S	I	A	I(S)

Table 6.4: The possible assignments for REFL attack scenario

6.3.3.3 Interleaving (INTRL) Attack Scenario

As the INTRL attack is launched at protocols with repeated authentication part, the INTRL attack scenario starts from the step of beginning the second authentication part. In general, this scenario can be described as follows. The first authentication part is implemented without an intruder impersonation. After that, the second authentication

part proceeds in a way that every two actions of a session are followed by the same two actions of the following session until the last session. The assignment for this scenario is presented in Table 6.5. The specification for this scenario is introduced in Code 6.7.

No	Odd Sessions			Even Sessions		
	Initiator	Responder	Server	Initiator	Responder	Server
1	I(A)	B	S	I(A)	B	S
2	I(A)	B	S	A	I(B)	S
3	I(A)	B	S	B	I(A)	S
4	I(A)	B	I(S)	I(A)	B	S
5	I(A)	B	I(S)	A	I(B)	S
6	I(A)	B	I(S)	B	I(A)	S

Table 6.5: The possible assignments for INTRL attack scenario

```
rule r_ExchangeWithAttacker($p in Protocol)=
 let ($s=p(sNo, msgNo, SENDER)) in
   let ($r=p(sNo, msgNo, RECEIVER))in
     while existAttack do
       seq
         if state=SEND then
           if ($s=II) or
              (imp($s, sNo, msgNo)=true)  then
             r_Intruder[$p]
           else
               par
                 r_Initiator[$p, $s, $r]
                 r_Responder[$p, $s]
                 r_Server[$p]
                 state:=RECEIVE
               endpar
           endif
         endif
         if state=RECEIVE then
           par
             if ($r=II) or
                (imp($r, sNo, msgNo)=true)  then
               par
                 r_Overhear[$p]
                 go:=false
               endpar
             else
                par
                  r_Initiator[$p, $r, $s]
                  r_Responder[$p, $r]
                  r_Server[$p]
                endpar
             endif
               state:=SEND
           endpar
         endif
       endseq
    endlet
```

```
rule r_Intrl($p in Protocol)=
  seq
    if rstep>1 then
      while msgNo<rstep do
        seq
          r_ExchangeMsgs[$p]
          msgNo:=msgNo+1
        endseq
    endif
    actstep:=rstep
    while actstep<=totalMsgNo do
      seq
        while sNo<=totalSno do
          seq
            msgNo:=actstep
            x:=1
            while x<=2 and msgNo<=totalMsgNo do
              seq
                go:=true
                r_ExchangeWithAttacker[$p]
                msgNo:=msgNo+1
                x:=x+1
              endseq
            if sNo=1 and msgNo>=totalMsgNo then
               sNo:=totalSno+1
            else
               sNo:=sNo+1
            endif
          endseq
        actstep:=actstep+2
        sNo:=1
      endseq
  endseq
```

Code 6.7: The r_ExchangeWithAttacker and the r_Intr rules

In Code 6.7, there are two rules: the r_Intrl rule, which specifies the scenario pattern, and the r_ExchangeWithAttacker rule, which is called by the r_Intrl rule to send and receive

messages between the honest participants and the intruder during the second authentication part. In the r_Intrl rule, the rstep function represents the beginning step of the second authentication part.

6.3.3.4 Denial of Service Replay (DoS_REPL) Attack Scenario

This scenario can be described as follows. First, the initial session is performed only between honest participants in order to save the message that can be replayed in the next sessions. Second, the next sessions include eliminating all initial actions in which the responder does not participate, i.e., the next sessions start from the first action at which the responder receives a message. The DoS_REPL attack scenario has only one possible assignment shown in Table 6.6.

```
rule r_InterceptSend($p in Protocol)=
 let ($s=p(sNo, msgNo, SENDER)) in
   let ($r=p(sNo, msgNo, RECEIVER))in
     seq
       par
         r_Initiator[$p, $s,$r]
         r_Responder[$p, $s]
       endpar
       choose $id1 in Id, $id2 in Id
       with ($id1!=II) and ($id1!=$id2)
       and (outBox($id1, $id2)!=undef) do
             outBox($id1, $id2):=undef
       r_Intruder[$p]
       par
          r_Initiator[$p, $r, $s]
          r_Responder[$p, $r]
       endpar
       state:=SEND
     endseq
   endlet
 endlet
```

```
rule r_SimpleReplay($p in Protocol)=
 let ($s=p(sNo, msgNo, SENDER)) in
   seq
     msgNo:=1
     sNo:=1
     state:=SEND
     reply:=4
     existAttack:=true
     while msgNo<=totalMsgNo do
       r_ExchangeMsgs[$p]
     sNo:=sNo+1
     msgNo:=1
     wantToInit($s):=true
     while sNo<=totalSno do
       seq
         while msgNo<reply do
           r_ExchangeMsgs[$p]
         while msgNo<=totalMsgNo do
           seq
             r_InterceptSend[$p]
             msgNo:=msgNo+1
           endseq
         sNo:=sNo+1
         msgNo:=1
       endseq
   endseq
 endlet
```

Code 6.8: The r_FirstSession and the r_Dos_Reply rules

No	First Session			Next Sessions		
	Initiator	Responder	Server	Initiator	Responder	Server
1	A	B	S	I(A)	B	I(S)

Table 6.6: The possible assignments for DoS_REPL attack scenario

This scenario is illustrated in Code 6.8. In Code 6.8, there are two rules: the r_DoS_Replay rule, which specifies the scenario arrangement, and the r_FirstSession rule, which is called by the r_Dos_Replay rule to perform the first session without intruder impersonation and to save the message which will be replayed in the intruder knowledge .

6.3.3.5 Simple Replay (Simple_REPL) Attack Scenario

This scenario is similar to the DoS_REPL attack scenario with respect to executing the first session without impersonation, but it differs from it by implementing the next sessions without omitting any message. In addition, this scenario needs an input parameter called *repl* that identifies the starting step which includes replaying a message by the intruder. Messages before this step are exchanged only between honest participants, then at the *repl* step the sent message is intercepted, and a message from the first session is replayed to the receiver by impersonating its original sender. This means the impersonation occurs just at this step. The possible assignment for this scenario is shown in Table 6.7.

No	First Session			Next Sessions		
	Initiator	Responder	Server	Initiator	Responder	Server
1	A	B	S	A	B	S
2	A	B	S	I	B	S

Table 6.7: The possible assignments for Simple_REPL attack scenario

This scenario is illustrated in Code 6.9. In Code 6.9, there are two rules: the r_SimpleReplay rule, which specifies the scenario arrangement, and the r_InterceptSend rule, which is called by the r_SimpleReplay rule to intercept a message by the intruder and replay a message from previous session to the receiver.

```
rule r_InterceptSend($p in Protocol)=
 let ($s=p(sNo, msgNo, SENDER)) in
   let ($r=p(sNo, msgNo, RECEIVER))in
     seq
       par
         r_Initiator[$p, $s,$r]
         r_Responder[$p, $s]
       endpar
       choose $id1 in Id, $id2 in Id
       with ($id1!=II) and ($id1!=$id2)
       and (outBox($id1, $id2)!=undef) do
             outBox($id1, $id2):=undef
       r_Intruder[$p]
       par
          r_Initiator[$p, $r, $s]
          r_Responder[$p, $r]
       endpar
       state:=SEND
     endseq
   endlet
 endlet
```

```
rule r_SimpleReplay($p in Protocol)=
 let ($s=p(sNo, msgNo, SENDER)) in
   seq
     msgNo:=1
     sNo:=1
     state:=SEND
     reply:=4
     existAttack:=true
     while msgNo<=totalMsgNo do
       r_ExchangeMsgs[$p]
     sNo:=sNo+1
     msgNo:=1
     wantToInit($s):=true
     while sNo<=totalSno do
       seq
         while msgNo<reply do
           r_ExchangeMsgs[$p]
         while msgNo<=totalMsgNo do
           seq
             r_InterceptSend[$p]
             msgNo:=msgNo+1
           endseq
         sNo:=sNo+1
         msgNo:=1
       endseq
   endseq
 endlet
```

Code 6.9: The r_InterceptSend and the r_SimpleReply rules

6.3.4 The Invariant Security Properties Aspect

In order to detect whether a protocol is vulnerable to an attack following a particular attack scenario, we use invariant checking for the security properties at simulated states of the protocol run. We are interested in inspecting two security constraints: *secrecy* and *authentication*.

The secrecy constraint is related to confirming that no secret information is obtained by others. For example, in NSPK protocol we have the following constraint:

invariant inv_sec over ncRsp, k: not(contains(k, append([], ncRsp(*BB, AA*))))

It means that the intruder knowledge does not contain the nonce of BB which is sent to AA.

The authentication property is defined in [238] based on the correspondence assertion style, as follows "when an authenticating principal finishes its part of the protocol, the authenticated principal must have been present and participated in its part of the protocol". This can be formulated as invariant constraint over *start* and *finish* functions, such as in NSPK protocol we have:

invariant inv_authAB over start, finish: implies(finish(*NS*, 2, 1), start(*NS*, 2, 1))

It informally means that if a participant successfully finishes receiving the first message in the second session, then this implies that sending this message has been started previously in this session. This constraint helps to prove the authentication of AA to BB regarding this message, as the first message is sent to BB from AA. For the reason that AA also sends the third message in the NSPK protocol, we have to assert the authentication of AA to BB regarding this message as well by changing the message number in the above constraint.

Recall that the NSPK protocol is intended to achieve mutual authentication, we also have to check the authentication of BB to AA, by replacing the position of the start and finish functions. More precisely, authenticating BB to AA can be checked through the following:

invariant inv_authBA over start, finish: implies(start(*NS*, 1, 1), finish(*NS*, 1, 1))

Informally this constraint asserts that when a participant has been started sending the first message in the first session, then this message will be received successfully in this session. Remember that the *start* and *finish* functions are updated only by the honest participants, not by the intruder.

6.4 Results and Discussion

In our experiments, we are firstly focused on applying the FATI method to the protocol under analysis in order to produce the possible attack types upon it, following that we are concerned with simulating the protocol runs, which are specified by the scenario of the expected attack type while observing the invariant properties. If there is no violation of any stated invariant property, there will be no successful attack related to these properties and the style of modelled attacks when its related scenario is simulated. We have analysed five protocols, including NSPK, TP, DS, AS_RPC, and KSL protocols, as they well-known case studies, and they exhibit different patterns of attack scenarios related to the considered attack types; in particular, MITM, REFL, DoS_REPL, Simple_REPL, and INTRL attacks, respectively[1].

As our methodology consists of two distinct phases, we will discuss each phase separately in the following subsections.

[1]The specification for the five protocols with their related attack scenarios are available in [22]

6.4.1 Manual Analysis Phase

The main idea of the manual analysis phase is to apply the FATI method to security protocols using queries and guidelines, in order to identify the potential flaws and their possible associated attack types. These queries and guidelines are useful for a concise and instructive analysis of protocol's flaws and attack types, as illustrated in its application to security protocol examples in Sections 6.2.1-6.2.2, and they are also useful for saving efforts needed in the next phase.

To evaluate the FATI method, we compare it to VIA and REA methods illustrated in Chapter 3. Both of these methods were invented to identify vulnerabilities in security protocols and to elicit requirements that counteract these vulnerabilities. The VIA method uses guide words to identify the potential vulnerabilities, e.g., the *omission* guide word is used to identify the vulnerability of no message sent, though, such guide words will only identify vulnerabilities at an abstract level without precise recognition of the vulnerability nature. Furthermore, the VIA method suffers from repetition in addressed vulnerabilities.

The REA method decomposes only the negated protocol's goal (undesirable event) into its possible sub-events that contribute to creating vulnerabilities. The decomposition is performed by asking a query "what do we mean by [the event under consideration]?", which is answered with respect to satisfying the security properties, such as availability, secrecy, etc. Since this query is asked several times until a vulnerability is reached, REA method consumes time and efforts. On top of that, the results of this method are not complete, due to considering only the negated goal. Furthermore, both of these methods do not identify possible attack types against security protocols.

Our method, on the other hand, is more determinant and precise at recognizing the vulnerabilities (or flaws) by showing their nature (freshness or liveness) and their positions (at which protocol action they occurs). In addition, it gives an insight into the possible attack types that could compromise the security of a protocol. In spite of that, our method is not suitable for analysing protocols that have not been designed yet, while the other two methods are more convenient for that.

6.4.2 Automatic Analysis Phase

This phase uses simulation as a process of validation for security protocols. Through the simulation process, two checks are performed in every simulated scenario: the violation of invariant constraints and the acceptance of messages sent by the intruder. Typically, a

produced scenario from the simulation is equivalent to an attack, when the invariant constraints are violated, and all messages sent by an intruder are accepted by their intended receivers. For example, by simulating the NSPK protocol using the MITM scenario with two sessions, we obtained four scenarios. Only one of them has messages that conform to the attack shown in Figure 4.2b and its invariant properties presented in Section 6.3.4 are violated. During this simulation, the intruder sends messages to honest participants three times. By the addition of the expected content check, the number of the generated messages each time is less than that obtained with the type matching method [129] . For example, in our method, the intruder generates 6 messages at the first step of second session: $\{A, N_A\}_{pk(B)}$, $\{B, N_A\}_{pk(B)}$, $\{I, N_A\}_{pk(B)}$, $\{A, N_I\}_{pk(B)}$, $\{B, N_I\}_{pk(B)}$, $\{I, N_I\}_{pk(B)}$. While 24 messages are generated based on the type matching method. Fewer messages are generated since the intruder is restricted with the expected responder public key, instead of all the known keys, including $pk(A), pk(B), pk(I), prk(I)$.

Note that, the attack scenarios give guidance exploring all runs with correct message formats and types while looking for attacks, and this can reduce the number of checked runs. As the number of protocol runs is positively impacted by the number of executed sessions and the number of steps in each session, reducing the generated messages by the intruder will eventually minimize the protocol runs. Combining the attack scenarios with the idea of reducing the intruder's messages can dramatically decrease the protocol runs.

Concerning the TP protocol, we obtain the attack in Figure 4.5b by executing the REFL scenario with two sessions. In fact, our method is flexible in a way that it allows adding more checking abilities to the participants for easy detection of the attack in Figure 4.5b by themselves, i.e., checking that the received message is not equal to the one sent in a session before. The TP protocol is a challenging case study as it's specification depends on the commutative encryption property. Commonly, this property is modelled as an equation directed from left to right (where the right-hand side is either a constant or a rigid subterm of the left-hand side) by tools like Proverif and Tamarin, in which case it can cause non-terminated analysis. We avoid this by modelling this property in a procedural way that captures the commutative equivalence condition.

Although understandably we cannot claim any form of completeness for our method, we can say that it detects attacks with given specified scenarios, including MITM, REFL, DoS_REPL, Simple_REPL, and INTRL, on protocols vulnerable to these attacks, though it cannot detect other attacks with undefined scenarios, such as type flaw and collaborative attacks.

In this methodology, we have modeled any protocol using the basic ASM instead of the distributed ASM in order to check the simulation outputs (update sets) for every protocol participant which has a rule equivalent to its behavior. Note that, to encode an AametaL model of a distributed system or multi-agents system, we have to specify the main rule for the main ASM which imports all the ASM modules that correspond to each agent of the system, and the AsmetaS separately shows the outputs for each specified module.

6.5 Related Work

Different formal methods have been used for analysing security protocols. Model checking tools, such as Spin and NuSMV tools, have been applied to the automatic verification of security protocols, such as [47, 155]. These tools successfully prove that insecure states are unreachable but with a bounding assumption for generated fresh components or protocol sessions.

With respect to the unbounded analysis issue, ProVerif [51], is a well-known tool that has emerged to address this. ProVerif employs Prolog style rules and executes an abstract representation technique to analyse an unbounded sessions. Its specification language is applied pi calculus. ProVerif reasonably proves secrecy and correspondence properties. However, for some protocols, e.g. TP protocol, and algebraic properties, ProVerif may go into an infinite loop which leads to a stack overflow. Recent work related to unbounded analysis [168], presents the Tamarin tool as a protocol verification tool in a symbolic model checking style. The specification language for Tamarin is based on multiset rewriting rules. Tamarin's language is more expressive than that of ProVerif in specifying security properties, as it supports direct modelling for temporal properties. It nevertheless shares with ProVerif the same non termination limitation.

The ASM method has been theoretically used for specifying and analysing security protocols, taking into account modelling intruder capabilities [45, 46]. However, this work does not deal with expected content during generating messages.

The papers [130], and [129] are closest to our work. The first paper restricts the intruder's ability to generate only the messages of a particular type, which can be accepted by some agents. Depending only on type matching still may produce unacceptable messages. In [129] a methodology of simulating security protocols using scenarios designed for different attack schemes is presented. This is implemented in the Estelle language. The results showed it is an effective approach to reduce the number of protocol runs. However,

in this work, the intruder generates messages based on method of [130]. In addition, this work does consider the algebraic properties of the encryption mechanism.

6.6 Conclusion

This chapter presents a methodology for analysing security protocols and developing secure designs for them. This is achieved by extending the simulation based attack scenarios method in [129] via 1) avoiding to waste unnecessary efforts in simulating all the specified attack scenarios by prior identification of the expected attack types; 2) specifying the scenarios in AsmetaL and simulaing them using AsmetaS tool 3) modelling a "clever" intruder that will generate acceptable messages only by use two conditions: the expected message content and type matching, to limit the intruder's generated messages only to those that meet these conditions; and 4) considering the commutative property of encryption mechanism during protocol analysis.

Chapter 7

Clarifying Ambiguous Requirements: SASL Case Study

7.1 Introduction

In Chapter 6, we presented a methodology to analyse security protocols in order to identify the main design flaws in these protocols and how the intruder can exploit these flaws to lunch attacks. Chapter 6 also highlights the fact that flaws do not necessarily emerge from mistakes in the design itself, rather they might come from silent assumptions or from missing or non clarified details in the protocol requirements. To overcome that, the requirements must be clarified precisely before performing the analysis process. As a result, this chapter shows how to clarify informal requirements for the Simple Authentication and Security Layer (SASL) case study using Abstract State Machines.

SASL is a framework that can be used by application protocols to perform authentication, and to optionally supplement it with what is called security layer services, including integrity and confidentiality. SASL was firstly described in Requests for Comments (RFC) 2222 [173], and then in RFC 4422 update [169], using technical natural language. Unfortunately, the RFCs, being stated in natural language that intrinsically has associated informality and imprecision, are sometimes ambiguous. There is an Oracle implementation of SASL [177], its documentation also includes textual explanations and some underspecified aspects of the RFCs. In addition, as the source code of this implementation is not publicly available, their details are unknown.

To overcome the imprecision and ambiguity problems, formal methods can be used

as they are based on mathematical foundations [109]. Among these methods, we choose the Abstract State Machine (ASM) method, since it can be used to specify systems in a rigorous mathematical, understandable, and scalable way.

Consequently, we formally analyse the ambiguities of the SASL textual explanations in the RFCs and Oracle implementation documents using the ASM method. This is achieved by implementing two strategies of the ASM method. First, the ground model directly captures the informal SASL behavior in an understandable and concise but precise enough manner. Second, the refinement strategy allows us to precisely explicate and re-elaborate the under-defined notions in the ground model. The refined specification is written in the executable ASMETA Language (AsmetaL), since it is close to the ASM mathematical concepts, and it directly permits us to test specification errors.

The main contributions of this chapter are:

- clarifying the ambiguities of the SASL informal descriptions in RFCs and Oracle implementation documents clearly in terms of ASMs;
- presenting a methodology for clarifying ambiguity that starts with the RFC document to capture its informal description, via the ASM ground model, then it explicates the potential description ambiguities depending on other document sources by using ASM refinement;
- highlighting the main differences between RFCs and the Oracle documents.

The rest of this chapter is organized as follows. Section 7.2 introduces the SASL framework. Section 7.3 describes the ASM formal specification, and highlights how this specification elucidates the main ambiguities of SASL. Section 7.4 discusses our results. Section 7.5 presents some related work. Finally, Section 7.6 concludes the chapter.

7.2 Simple Authentication and Security Layer

In this section, we describe the Simple Authentication and Security Layer (SASL) example. SASL was initially introduced in RFC 2222 [173], and later updated in RFC 4422 [169], as a framework for providing authentication support with an optional security layer service, such as integrity or confidentiality, to connection-oriented protocols, via substitutable mechanisms. Providing these services is achieved through using a shared abstraction layer which has a structured interface between intended protocols and mechanisms. With this

layer, any SASL supported protocol, such as IMAP [215], SMTP [216], etc., can exploit any SASL supported mechanism, such as PLAIN [243], DIGEST-MD5 [144], etc.

Based on RFC 2222/4422 [173, 169], the client and server of the SASL protocol application launch a negotiation about the selection of a suitable mechanism, then they negotiate the authentication. Essentially, the client requests to connect with the server using SASL. Then, the server replies with a list of supported authentication mechanisms. Next, the client selects the best mechanism. After that, the authentication is started by the client via sending an authentication command, which involves the selected mechanism and optionally authentication data, to the server. The authentication exchange continues until the authentication succeeds, fails, or is aborted by the client or the server. During the authentication exchange, when the selected mechanism supports the security layer, the client and server negotiate the use of a security layer. If they both agree about using it, then both sides must negotiate the maximum size for the cipher text buffer, that each side is able to receive. The RFCs specification, however, leaves open a number of questions, in particular:

- how the server advertises its mechanisms' list;
- how the client selects best mechanism, when the client and server agree about using the security layer and how it can be used; and
- how they negotiate the maximum cipher text buffer size.

Some of these questions also relate to the ambiguity and missing details of the informal description for the API routines in the Oracle implementation documentation [177].

According to the Oracle implementation [177], the application communicates with the structured interface by calling a suitable API routine, which in turn calls a mechanism plug-in interface. One of these routines is: the `sasl_client_start()` which is called by the client to select the best mechanism depending on the security properties. The main properties that restrict mechanism selection are: the **security policies**, such as NOPLAINTEXT, NOACTIVE, NOANONYMOUS, etc., and the maximum **Security Strength Factor (SSF)** [177] for the client, server, and mechanism. The SSF is an integer that denotes the security layer strength, as follows:

- When it is zero, it indicates only authentication.
- If it is one, it means both authentication and integrity.

- If it is greater than one, it denotes authentication, integrity, confidentiality, and at the same time the key length for encryption.

In addition, the server calls the `sasl_listmech()` routine to obtain the mechanisms' list that satisfies the security policies.

7.3 The Formal SASL Specification

In this section, we show how the ASM method has been used to provide formal specifications for SASL. The main aim is to clarify precisely: how the server advertises the available mechanisms, how the client selects the best mechanism, how the client determines the maximum size for the cipher buffer, and how and when the security layer is negotiated. As the SASL framework has detailed and complex behavior, we separate the SASL into three phases: the mechanism negotiation phase, the authentication negotiation phase, and the security layer negotiation phase. In each phase, we will present (if necessary) the ground model for both client and server sides, that is depicted via the control state ASM, then we will focus only on refining the rules to clarify ambiguities in the RFCs and Oracle implementation documentation[1]. Each refined rule is expressed in AsmetaL. The mapping from the graphical notation of the control state ASM to the AsmetaL notation is done according to the mapping shown in Figure 2.5 that is presented in Chapter 2.

7.3.1 The Mechanism Negotiation Phase

Figure 7.1 shows the ground model at the client side for this phase.

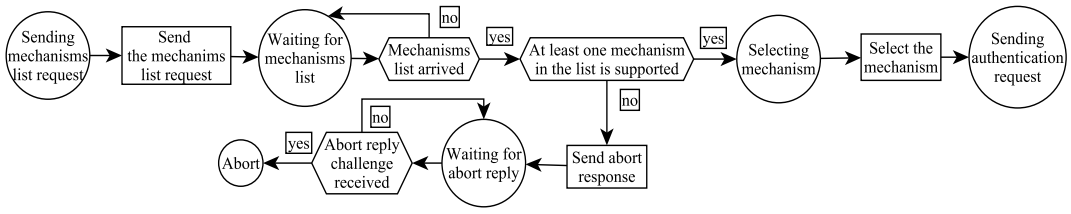

Figure 7.1: Client side for mechanism selection phase - ground model

[1]All the rules for the refined model that is based on RFC 2222/4422 are available in [24], while those for the refined model which is based on the description of Oracle implementation documentation are available in [23]

This figure is a direct interpretation of RFC 2222/4422. The client starts this phase by sending a request in order to ask the server to send its mechanisms' list. Whenever this list is received, the guard *'At least one mechanism in the list is supported'* checks if the client can use one or more mechanisms in the received list. If so, the client selects an acceptable mechanism from the server list and reaches the final state for this phase *'Sending authentication request'*. Otherwise, the client will send an abort response to the server, and waits for an abort reply from it. When the abort reply is received, the client aborts this exchange, by entering the *Abort* state.

The server side for this phase is also based on RFC 2222/4422, see Figure 7.2.

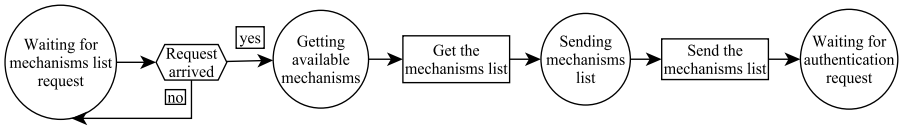

Figure 7.2: Server side for mechanism selection phase - ground model

The ground model depicted in Figure 7.2, schedules the main steps taken by the server for this phase. Initially, the server keeps waiting at the *'Waiting for mechanisms list request'* state until it receives a mechanism list request from the client. When this request arrives, the server obtains the available mechanisms' list in order to send it to the client. After sending this list, the server goes to the final state for this phase, which is *Waiting for authentication request*.

It is not clear from Figure 7.1, how the client chooses the desired mechanism. As stated by RFC 4422 [169], determining the best mechanism is the client's choice. This is specified in the r_selectmech rule shown in Code 7.1a. In this Code, the mechanism selection is performed in an interactive manner with the client via the monitored function insertMechanism (the monitored function is a function whose value written by the environment of the machine). The selected mechanism should be any mechanism in the arrived mechanisms' set, which is represented by the arrivedMechList. An arbitrarily chosen mechanism is stored in selMech.

On the other hand, the explanation of the Oracle implementation documentation [177] states that the client selects the best mechanism, depending on the maximum mechanism SSF and client's security policy. This can be re-elaborated by refining the rule in Code

7.1a into the one shown in Code 7.1b. In the refined rule, we added further modelling vocabulary. Precisely, let the ssf($m) function be the SSF value for each mechanism $m in the domain Mechanisms={PLAIN, DIGESTMD, ANONYMOUS, EXTERNAL}, and policies($m) be the security policies set for each mechanism $m, while policies(self) is the security policies' set for client. The 0-ary function self is interpreted by the client agent as itself. Each policies set can be one or more elements from the domain Policies={NOPLAINTEXT, NOANONYMOUS, NOACTIVE, MUTUALAUTH, NODICTIONARY}. The mHasGreatestSSF function returns true if the selected mechanism has the greatest SSF value. The refined rule picks the best mechanism from the arrivedMechList, such that the security policies set of the selected mechanism includes all the elements in the client's set, and this mechanism hasan SSF value, which is equal or greater than all the SSF values of the mechanisms that their sets include the client's policies set.

```
rule r_selectmech=
  if contains(arrivedMechList, insertMechanism) then
       selMech:=insertMechanism
  endif
```

(a) The r_selectmech rule according to RFC 4422

```
function mHasGreatestSSF($m in Mechanisms, $c in Client)=
 forall $x in arrivedMechList with
   (($x!=$m) and allin(policies($x), policies($c)) implies ssf($m)>=ssf($x))

rule r_selectmech=
  choose $m in arrivedMechList with mHasGreatestSSF($m, self)=true and
                                    allin(policies($m), policies(self)) do
        selMech:=$m
```

(b) The refined r_selectmech rule according to Oracle implementation document

Code 7.1: The r_selectmech rule

On the server side in Figure 7.2, getting the available mechanisms' list needs elucidation. As indicated by RFC 4422 [169], the server just advertises the available mechanisms' list. This is specified in the r_getmechs rule shown in Code 7.2a. In this code, let the mList($c, self) be a set of the advertised mechanisms which will be sent to the $c client. The saslmechs(self) set contains one or more SASL mechanisms for server use. The server, in the r_getmechs rule, will simply make a copy of all the elements in the saslmechs(self) set and pass it to the mList($c, self), which is initially empty set, to represent the advertised mechanisms' list.

```
rule r_getmechs($c in Client)=
  mList($c,self):=saslmechs(self)
```

(a) The r_getmechs rule according to RFC 4422

```
rule r_getmechs($c in Client)=
  let ($i=0) in
    while $i<size(saslmechs(self)) do
      seq
       let($m=at(asSequence(saslmechs(self)), iton($i))) in
         if (exist $p in policies(self) with
             contains(policies($m), $p)=true) then
           mList($c,self):=including(mList($c,self),$m)
         endif
       endlet
       $i:=$i+1
      endseq
  endlet
```

(b) The refined r_getmechs rule according to Oracle implementation document

Code 7.2: The r_getmechs rule

Obtaining the available mechanisms' list is described in the Oracle implementation documentation [177], as "The server can call sasl_listmech() to get a list of the available SASL mechanisms that satisfy the security policy". In this quoted statement, it is not obvious whether there is a specific policy for the SASL mechanisms and what is meant to satisfy this policy. In the Java security guide provided by Oracle [178], it says that there is a particular policy set for each SASL mechanism, such as the PLAIN, EXTERNAL, and DIGEST-MD5 mechanisms have certain policy sets illustrated in Table 7.1.

Mechanism	**Policy Set**
PLAIN	NOANONYMOUS
EXTERNAL	NOPLAINTEXT. NOACTIVE, NODICTIONARY
DIGEST-MD5	NOPLAINTEXT, NOANONYMOUS

Table 7.1: The security policies for authentication mechanisms

As an attempt to understand the exact meaning of 'satisfy the security policy', we analyse the server's reply (sending the available mechanisms' list to the client) in some SASL mechanism examples. For instance, in the DIGEST-MD5 mechanism example [144], the server sends the {PLAIN, DIGEST-MD5} list. We can see that these two mechanisms share the NONANONYMOUS policy. This means that the server adopts the NONANONYMOUS

policy and it sends the mechanisms which satisfy this policy. Similarly, in the EXTERNAL mechanism example [169], the server sends the {DIGEST-MD5, EXTERNAL}. Again, these mechanisms in the list share the NOPLAINTEXT policy, which is supported by the server. Accordingly, the r_getmechs rule in Code 7.2a can be refined into the rule in Code 7.2b.

In the refined r_getmechs rule, the server gets a mechanism from the saslmechs for the server use, which satisfies the following condition: the policy set for this mechanism contains a policy of the server's policies set. In other words, the policy set for every mechanism in the advertised mechanisms' list supports at least one server's policy.

7.3.2 The Authentication Negotiation Phase

This phase is the longest phase in SASL. As a result, we divide the ground model for both client and server into two parts: one for achieving an initial step in this phase, and one for performing the later step(s). The number of the later steps is determined by the selected mechanism. In this section, we provide the ground model for achieving only the initial step by the client and server as this step is not clear, though the ground models of the rest steps are presented in Appendix C.

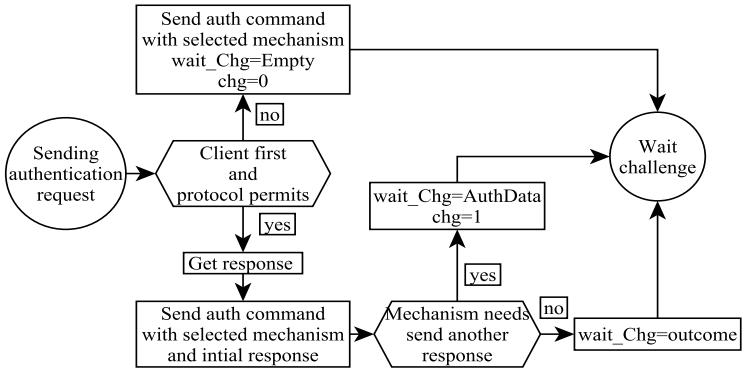

Figure 7.3: Client side for performing an initial step in the authentication negotiation phase - ground model

Figure 7.3 shows the ground model for performing an initial step in this phase by the client. The client starts this phase checking by the guard *'Client first and protocol permits'*, to

inspect whether the selected mechanism determines that the client must send the first response and the protocol allows the client to do that. If so, the client will *'Get response'* to send it together with authentication command and the chosen mechanism to the server. Then, the flow goes further to the *'Mechanism needs send another response'* guard. If this guard is satisfied, the client will keep waiting for the server authentication data. Otherwise, the client will wait for an authentication outcome. However, if the guard *'Client first and protocol permits'* is not satisfied, the client will send only the authentication command with the chosen mechanism. Then, the client goes to the *'Wait challenge'* state, and keeps waiting for an empty challenge.

On the server side, Figure 7.4 displays the ground model for achieving an initial step in this phase by the server.

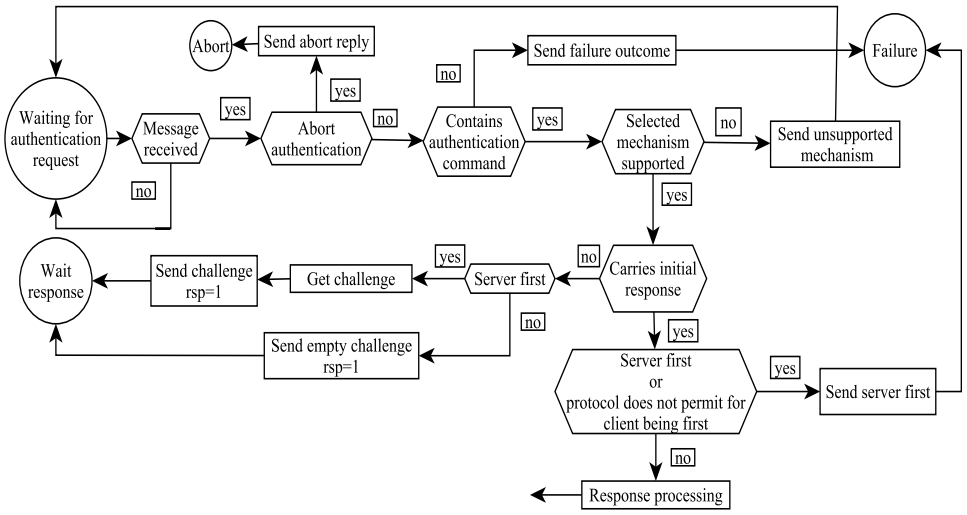

Figure 7.4: Server side for performing an initial step in the authentication negotiation phase - ground model

In Figure 7.4, the server initiates this phase, when it receives a message from the client. At the receiving time, the server checks three guards: the message is not an *'Abort authentication'* response, it *'Contains authentication command'*, and the *'Selected mechanism is supported'*. When these guards are satisfied, the server checks if the message *'Carries an initial response'*. If so, the server checks if the mechanism specifies that the server is the

first, or it specifies that the client is the first but the protocol does not allow for it to be the first. In case that the mechanism specifies that the server must send the first message, but the client does, or the protocol does not allow the client to send the message, then the server sends the server must be the first and enters the *'Failure'* state, otherwise, it will process the response by verifying it and determining the next step. When the message does not have an initial response, the server checks if the *'server is the first'*, to *'get its initial challenge'* and sends it to the client. Otherwise, the server just sends an empty challenge. In both cases, the server goes to the *'Wait response'* state.

One underspecified aspect of this phase, is the negotiation about using the security layer and the maximum cipher buffer size, which are involved in the Get response and Get challenge rules on the client and server sides, respectively. In RFC 4422 [169], it was stated that when the selected mechanism supports a security layer, then a negotiation about using this layer must be carried out, but how this negotiation takes place is not defined. However, RFC 2222 [173] defines this by stating that the negotiation includes exchanging a **bit-mask** (1: no security layer, 2: integrity, and 4: privacy), which corresponds to a security layer level. This bit-mask ignores the confidentiality service.

On the other hand, in the explanation of the Oracle implementation documentation [177], the SSF value (0: authentication, 1: authentication and integrity, and > 1: authentication, integrity, confidentiality and the key length), is used instead of a bit-mask. However, it is not clear how the client and server agree about using a security layer.

The Java security guide provided by Oracle [178] states that the selected mechanism, when its SSF value is greater than or equal to 1, tells the server to send its supported **Quality of Protection** (QOP) list, which includes one or more items from the following: **auth** (authentication), **auth-int** (authentication and integrity), and **auth-conf** (authentication, integrity, and confidentiality). Later, the client selects a protection value from this list according to its SSF value, and sends it to the server. The server verifies that the client's protection value is within its list, to save the session SSF value which is equivalent to the client's protection value. The saved SSF value represents the agreed security layer service. However, in this guide, there is insufficient detail about how the client and server determine the maximum buffer size when they agree about using the confidentiality service.

In the DIGEST-MD5 SASL mechanism example [144], it was stated that when the server sends its supported maximum buffer size (if desired), the client will check the availability of buffer size value in the received challenge. If it exists, the client will calculate the size of the buffer for this session, by firstly determining the minimum value of the

received buffer size and its supported size, then from the determined value, the 16 bytes are subtracted to produce the buffer size for the current session. If it is not available, the client will determine that the buffer size is equal to the default value 65536. Following this description, we present in Code 7.3 the specification of how the client determines the maximum buffer size.

```
if contains(receivCh(self), "maxbuf")=true then
  choose $max in Maxbuf with
  eq(at(receivCh(self), iton(indexOf(receivCh(self), "maxbuf")+1)), toString($max)) do
      if $max<maxBuf(self) then
          maxBufDetermined:=$max-16
      else
          maxBufDetermined:=maxBuf(self)-16
      endif
else
    maxBufDetermined:=65520
endif
```

Code 7.3: Specifying the maximum buffer size in the client side

In Code 7.3, the receivCh(self) is a sequence of String that represents the received challenge from the server, the integer domain Maxbuf contains the following possible values for the buffer's size {65535, 131071, 262143, 16777215}, and the maxBuf(self) is the client's maximum buffer size. First of all, the client checks if the receivCh(self) contains the server's maximum buffer size, to calculate the buffer size, or to set it to the default value. Setting to a default value which is 65535, will be when the server does not send its supported buffer size, i.e., the receivCh(self) contains the server's maximum buffer size. Otherwise, the client chooses an integer value from the Maxbuf domain, since the receivCh(self) is a sequence of String, which is equal to the string value contained in the receivCh(self). Then the chosen value is compared with the client's buffer size to determine the buffer size, which is stored in maxBufDetermined.

7.3.3 The Security Layer Negotiation Phase

This phase is an optional phase. Performing this phase depends on the negotiation in the previous phase. Based on RFC 2222 [173], this negotiation includes exchanging a bit-mask in order to decide on the use of this layer. In case that this layer is agreed upon, then the messages related to this layer are exchanged between the client and server. Figure 7.5 the ASM ground model for negotiating the security layer service on the client side.

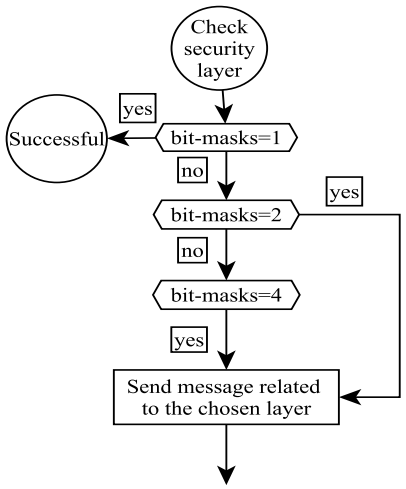

Figure 7.5: Client side for security layer negotiation phase - ground model based on RFC 2222

In Figure 7.5, it is not clear what are the messages that should be exchanged between the agents to achieve the service of this layer, and how to finish this phase. Furthermore, the bit-mask does not define the confidentiality service. The server side shares the same ambiguities for the client side. RFC 2831 for the DIGEST-MD5 SASL mechanism [144] clearly shows the specification of messages related to achieve integrity and confidentiality protection. In addition, instead of the bit-mask, the Oracle documentation [177] uses SSF value to distinctly indicate which layer is negotiated. Furthermore, RFC 4422 [169] illustrates how to finish this phase. As a result, we specify this phase relying on three references: the Oracle implementation documentation [177], the RFC 4422 [169], and the RFC 2831 for the DIGEST-MD5 SASL mechanism [144].

Starting from the client side to clarify this phase, Figure 7.6 illustrates the clarified ASM ground model for negotiating the security layer service on this side. In this figure, the client starts this phase by checking the SSF value, that was agreed by both client and server in the authentication phase. If this value is zero, then the client will reach the final state *Successful*. This state indicates that the client has been authenticated successfully, and there is no security layer. If the SSF value is one, this means that the subsequent protocol messages must be integrity protected. Therefore, the flow goes to execute the *Get client integrity protected message*, to obtain a test message that appends with the

computed Message Authentication Code (MAC) for the message sequence number and the message itself [144]. While if the SSF value is greater than one, then the following protocol messages must be confidentiality protected (encrypted). As a result, the client executes the *Get client confidentiality protected message*, to encrypt a test message together with its computed MAC [144].

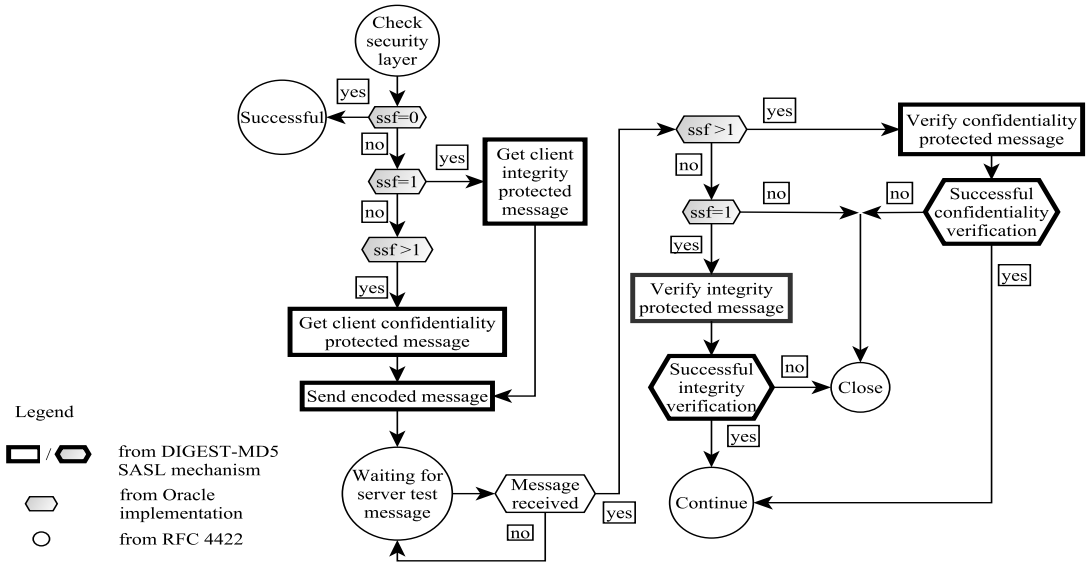

Figure 7.6: Client side for security layer negotiation phase - ground model based on three references

The encryption is done according to the selected cipher, which is one of the following: rc4-40 (40 bit key), rc4-56 (56 bit key), rc4 (128 bit key), and aes-ctr (128 bit key). Later, the client sends the protected message to the server, and changes its state to *Waiting for server test message*. Note that, the two consecutive rectangles in this figure, i.e, the *Get client confidentiality protected message* and *send encoded message* are executed sequentially using **seq end seq** construct. Whenever the client receives a protected message from the server, it will check the agreed SSF value. When this value is greater than one, the client will perform confidentiality verification (decrypt the message, compute the MAC and compare it with the received one). While, if the SSF value is one, the client will perform integrity verification (compute the MAC and compare it with the received one). In case that the

verification succeeds, the client reaches the final state *Continue*, which means the client can continue the interactions after SASL. Whereas, the client terminates the connection with the server and changes its state to *Close*, when the verification fails. As the ground model in Figure 7.6, clearly shows how the client uses a security layer service, and how the SSF value guides the client to determine whether a security layer has been negotiated, we do not show the refinement for this model. We annotate the main information for specifying this phase in Figure 7.6.

On the server side, the security layer negotiation, which is shown in Figure 7.7, is similar to that on the client side. However, the server here keeps waiting for a protected message from the client before sending its message. Note that, the two consecutive rectangles in Figure 7.7, i.e, the *Get server confidentiality protected message* and *send protected message* are executed sequentially using **seq end seq** construct.

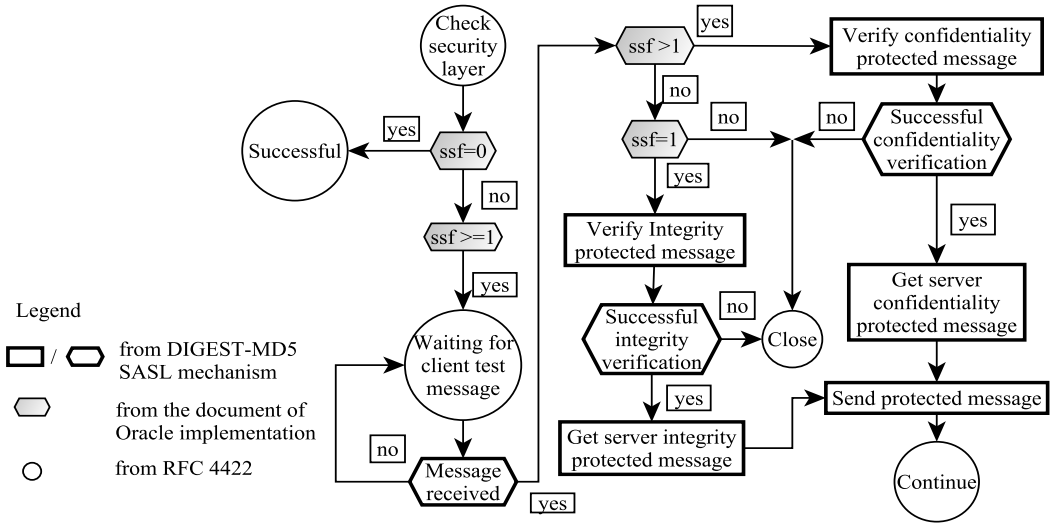

Figure 7.7: Server side for security layer negotiation phase - ground model based on three references

As in this phase we need to encrypt and decrypt the message with regards to a suitable cryptographic method, we modify our encryption and decryption specification, which is presented in Chapter 6, such that it takes into account the encryption and decryption key, and the named of the method for performing the encryption and decryption besides, see

Code 7.4. In Code 7.4, the method is a cipher method that has been used to encrypt the plain message. The r_Encrypt rule for converting the presented plain message into an encrypted one using the determined key and method, and the r_Decrypt rule which transforms the encrypted message into a plain text one using a specific key and method. In other word, the r_Encrypt rule generates a new ciphertext, given its plain text, key, and method, by extending the `Cipher` domain. While r_Decrypt rule choose a ciphertext item from the Cipher domain, in such a way that this item equals to the given ciphertext, and the key and the method for this item are equal to the given ones, in order to return the plain text of this item. If there is no such item, this rule will return undefined output.

```
dynamic abstract domain Cipher
dynamic abstract domain Key
controlled plainText: Seq(String)
controlled cipher: Cipher
controlled key: Cipher-> Key
controlled plain: Cipher-> Seq(String)
controlled method: Cipher-> String

rule r_Encrypt($m in Seq(String), $k in Key, $method in String)=
 choose $c1 in Cipher with (plain($c1)=$m and key($c1)=$k and method($c1)=$method) do
     cipher:=$c1
 ifnone
   extend Cipher with $c2 do
     par
         cipher:=$c2
         plain($c2):=$m
         method($c2):=$method
         key($c2):=$k
     endpar
rule r_Decrypt($c in String, $k in Key, $method in String)=
  choose $c1 in Cipher with (toString($c1)=$c and $k=key($c1) and method($c1)=$method) do
     plainText:=plain($c1)
  ifnone
     plainText:=undef
```

Code 7.4: Abstract specification for encryption and decryption with respect to the employed encryption method

7.4 Results and Discussion

The main aim of this chapter is to provide clarification of ambiguities in SASL using ASMs. Our methodology starts with reflecting the textual description in RFCs, using ground model notion, then it re-elaborates this description using other document sources by exploiting the refinement notion. Table 7.2 outlines the main ambiguities that have been investigated, and the source documents for both the ambiguity itself and its formal clarified specification.

From Table 7.2, we can see the following:

No.	The ambiguity	The document source for ambiguity	The clarified specification	The document source for clarification
1	The client selects the best mechanism	RFC 4422 [169]	Code 1 (b)	Oracle implementation document [177]
2	The server advertises the available mechanisms' list	RFC 4422 [169], and Oracle implementation document [177]	Code 2 (b)	DIGEST-MD5 SASL mechanism [144], Oracle implementation document [177], and its Java security guide [178]
3	Determining the maximum cipher text buffer size	RFC 4422 [169], and Oracle implementation document [177]	Code 3	DIGEST-MD5 SASL mechanism [144]
4	How and when the security layer is negotiated	RFC 2222 [173]	Ground model in Fig. 7.6	Oracle implementation document [177],DIGEST-MD5 SASL mechanism [144], and RFC 4422 [169]

Table 7.2: The source document for each ambiguity and its formal clarified specification

(1) selection of the best mechanism is ambiguous in RFC 4422 [169], as it just states that the client selects the best one. We try to elucidate this using the description of the Oracle implementation [177], which states that the client selects the best mechanism with the maximum SSF, and according to its security policy;

(2) advertising the available mechanisms' list is not clear in both RFC 4422, which only states that the server advertises the list, and the Oracle implementation, which states that the server advertises the mechanisms that satisfy the security policy. We convert the informal description of Oracle into a formal one, based on analysing the server reply in the document sources shown in Table 1. We conclude that satisfying the security policy means at least one server's policy must be supported by every mechanism in the advertised list;

(3) determining the maximum buffer size is under-defined in RFC 4422 [169] and the Oracle implementation. For explicating that, we use the explanation that is provided by the DIGEST-MD5 SASL mechanism [144];

(4) using the security layer in RFC 2222 needs more explication, as it states that using this layer relates to the agreed bit-mask, which does not consider the confidentiality service. Therefore, we rely on the Oracle implementation, that uses the SSF instead of a bit-mask, to show when this layer is used. Also, we rely on the DIGEST-MD5 SASL mechanism [144], to show how the client and server negotiate this layer.

This paper shows how the ASM formalism is valuable in clarifying the ambiguity, especially with its ground model and the refinement notions. The ground model can first

capture the informal specification in understandable way and at the desired level of details. Then, it can be evolved via stepwise refinement into a precise and enhanced mathematical specification. For the reason that we focus on employing the stepwise refinement for explicating the ambiguous behavioral aspects in the ground model, we have not proven the model refinement correctness. However, proving the relationship between the ground and the refined models can be addressed in the future by using the AsmetaRef tool [34], within ASMETA framework, which automatically facilitates proving the refinement correctness.

As we construct a formal specification and provide links between it and informal or under-defined resources, we could prove properties of the development specification using a suitable tool in the ASMETA framework.

In our ASM specification, the timing aspects for SASL are not considered, since neither of RFC 2222/4422 and Oracle implementation documents give specification for that.

7.5 Related Work

Our work elucidates ambiguities in the informal description for SASL, based on the ASM method. Therefore, in this section we discuss other work related to either elucidating ambiguity or to the ASM method. In [45], the ASM formalism is used to get a formal model of the Kerberos Authentication System which is based on the Needham and Schroeder authentication protocol. The formal model is used as a basis to locate the minimum assumptions to guarantee the correctness of the system and to analyse its security weaknesses.

In [72], the ASM ground models of a content adaptation system employed for the interactions between different client devices and the Cloud, is presented. This work is extended in [37], by refining the initial model into a more detailed one, through focusing on the interactions between the client and the middleware server to retrieve information relating to the client's device. Furthermore, the modelling process has been supported by validation and verification activities which are integrated within the ASMETA framework.

In [202], abstract encryption and decryption is specified using the language AsmL. This specification is based on the object-oriented features and constructs, and thus it diverts from the theoretical model of ASMs.

The researchers in [49] use Higher-order logic (HOL4) to develop a rigorous post-hoc specification for TCP, UDP, and the Sockets API, that reflects the behavior of different implementations, include: FreeBSD 4.6, Linux 2.4.20-8, and Windows XP SP1. They validate their specification against several thousand traces captured from these implementations,

to test whether they meet this specification. This paper is notable in the context of our work as its authors are motivated by increasing clarity and precision over ambiguous informal specifications of the RFC, that may result in inconsistent implementations. In this paper, we do not consider validating that the implementation meets the specification. We focus on clarifying ambiguities in the RFC description, and on elucidating uncertainty in the textual explanation of the implementation. Furthermore, our specification is expressed using the ASM method, because it is accessible, as it requires a minimum of notational coding, unlike HOL4, which requires extensively annotating the mathematical definitions side-by-side with informal specification [49].

7.6 Conclusion and Future Work

We have provided the ASM specifications that elucidate ambiguities in the SASL framework. We have focused on the ambiguous parts in RFC 2222/4422 and Oracle implementation documents, including mechanism selection, providing mechanisms' list, defining when and how the security layer can be used, and determining the maximum cipher buffer size.

We have showed how the comprehensible specification has been achieved based on two ASM notions: a ground model and stepwise refinement. The ground model enabled us to reflect the desired behavior, which is explained in RFCs, in an understandable way. While the stepwise refinement helped us to explicate the ambiguous part of the desired behavior in an accurate way, using other document sources to inform us.

We convert the informal specification into formal one by expressing it in the ASM formalism, which is mathematically well-defined, precise, and easily understood.

To further our research we are planning to consider the security of the SASL, to show whether the SASL specification is secure by using a suitable verification technique.

Chapter 8

Conclusion and Future Research Works

This chapter presents an overall summary of the work described in this thesis, a review of the main ndings in terms of the research question and associated research issues identied in Chapter 1, and some potentially fruitful directions for future research.

8.1 Summary

In this thesis, we have proposed two systematic methodologies to analyse safety-critical systems and security protocols based on combining a formal method with an informal safety analysis technique. In particular, during analysing safety aspects of critical systems, we combined the ASM supported model-checking method with the safety requirements yielding from applying the STPA technique to the system under analysis. While when analysing security aspects of protocols, we combined the ASM supported simulation method with the attack types obtained from a developed STPA technique, called FATI. Also, when using the SASL example for protocols whose informal requirements contain ambiguities and hidden details, we showed how we could obtain precise specifications that clarify these ambiguities and hidden details based on employing two ASM notions: ground model and stepwise refinement.

After the introductory chapter, the next three chapters overviewed the necessary background and the related work to the areas of our contributions, namely, the Abstract State Machines, the safety and security analysis techniques, and security protocols. Our contri-

butions were covered in chapters 5, 6, and 7 detailing the proposed analysis methodologies and clarifying the requirements for SASL.

In more detail, Chapter 5 presented the methodology that combines the ASM method with the STPA technique to analyse critical systems and develop safe designs for them. The fundamental idea was to link the model-checking based verification verification based supported by the ASM method with the STPA requirements which need to be formalized. The STPA requirements were formalized using the if and only if, and disjunction operators. In comparison to related work [17, 19] our formalization process captured the temporal aspects of the STPA requirements adequately and reduced the number of requirements to be verified. Our analysis methodology also conducted the simulation and validation activities before moving into the more expensive activity, which is verification, to gain enough confidence in the specified model. The effectiveness of this methodology was demonstrated by indicating the main unsafe aspects and conditions in the insulin pump control and train door controller systems. This motivated the development of another methodology that adopts a similar combination idea but in a way benefiting the security protocols.

Chapter 6 considered the proposed methodology for analysing security protocols. It imitated the previous methodology for critical systems but it replaced the STPA safety requirements with possible attack types and the model-checking with simulation in the verification process. The possible attack types relied on a proposed method called Flaws and Attack Types Identification (FATI). The FATI method is STPA-like as it is based on a set of queries to identify the possible protocol security flaws and their related attack types, in particular, liveness and freshness flaws that can be exploited by MITM, REFL, Simple_REPL, DoS_REPL, and INTRL attacks. Compared with the related work in [108], FATI is more specific and accurate at identifying flaws by showing their nature (freshness or liveness) and their positions (at which protocol action they occurs). The identified attack types were employed to guide the simulation-based attack scenarios process to look only for these attack types upon protocol under analysis. We cannot claim any form of completeness for our methodology, though, we can say that it does not miss attacks against the protocol examples we had analysed by simulating the specified scenarios for the external attacks, including MITM, REFL, DoS REPL, Simple REPL, and INTRL.

The main limitation with critical systems and security protocols analysis methodologies is the lack of automation for eliciting safety requirements and for identifying the flaws and attack types, and this can be considered as future work.

Finally, chapter 7 covered the ASM specifications that elucidate the main ambiguities in

the informal requirements for SASL case study. However, these specifications need further future work in concern with verifying whether they meet some properties that SASL should guarantee.

8.2 Main Findings and Contributions

This section outlines the main findings and contributions from the work presented in this thesis. The principal motivation for the work was represented in a single research question as follows: *Can ASM be combined with STPA to analyse safety aspects of critical systems and security aspects of protocols?*. The answer to this question required providing answers to a number of supplementary questions. Thus, this section is arranged by presenting the main findings in view of each supplementary questions, and then in terms of the main research question as follows:

(1) *Can the STPA technique help to improve the ASM specifications concerning safety issues? How can we formulate the STPA safety requirements to be used in further verification?*. The STPA technique can help to improve the ASM specification by feeding the STPA safety requirements to the model-checker which produces guided counter-example in case of violation. The four STPA requirements which are ordinarily represented by using 'provided', 'not provided', 'provided too early', and 'provided too late' phrases, were formulated into one formula using the $\leftrightarrow$, and the $\vee$ operators. By the $\leftrightarrow$ operator, we put a strong condition on providing the action, i.e., the action will not be provided with other combinations or with the later/earlier satisfying determined combinations. Furthermore, the $\vee$ together with $\leftrightarrow$ help to reduce the number of properties to be verified. The STPA requirements aided at improving the ASM model for train door controller example by highlighting the condition in which the model does not consider cancelling the open action after the door is fully opened, and for insulin pump control system, by highlighting the conditions wherein the model delivers a manual dose even if it exceeds the available insulin, and the model does not give an alarm if the available insulin in the reservoir is less than the sum of 4 maximum single doses.

(2) *Can the STPA technique be developed to analyse security protocols?*. To answer this question and based on the attacks reviewed in Chapter 4 together with the main flaws they had exploited, we developed FATI method which is an inspired form of STPA

that asks queries on each protocol action to determine the possible protocol flaws and attack types. We found that direct flaws identification can be realized by asking queries about participants' *liveness* and messages' *freshness*. The reason for choosing these two queries is the fact that attacks against security protocols essentially aim to compromise the authentication and/or secrets distribution goals. We developed guidelines to answer these questions based on the presence of an encrypted sent message containing a freshness component which is generated by the receiver and whose association is clearly stated. We have shown how the FATI method helps to determine the possible flaws and attack types for AS_RPC and NSPK protocols.

(3) *Given the answer to the above, in which analysis activity (simulation, verification), should the outcomes of the developed STPA technique be considered to serve the analysis of security protocols?*. The answer is simulation because the efficacy of its use to analyse security protocols has been proven in [129] with respect to the designed attack scenarios. However, it needs a guidance in deciding to simulate which attack scenario instead of simulating all the scenarios even those not definitely compromising the protocol under analysis. Therefore, we developed security protocols analysis methodology that applies the FATI method first to produce the expected attack types, then it performs simulation using scenarios specified for these types. Our methodology extended the simulation based method in [129] by reducing the number of explored runs more through decreasing the number of intruders generated messages via considering: the expected message content and type matching. Furthermore, the algebraic property for commutative encryption has been dealt with during analysing a protocol that depends on that property. Our methodology has been successfully detected known attacks against five protocols including NSPK, TP, AS_RPC, DS, and KSL protocols.

(4) *How to clarify ambiguous requirements for security protocols using the ASM method?*. The issue of imprecision and ambiguities in textual requirements of security protocols was shown with respect to the Simple Authentication and Security Layer (SASL) case study which is stated in RFCs documentations. Addressing that issue was through the implementation of two ASM essential notions: ground model and stepwise refinement. The ground model notion reflects the textual description in a concise but precisely enough manner, while the stepwise refinement notion allows us to accurately explicate and re-elaborate under-defined aspects in the ground model. The

uncertainty of textual explanation for SASL was elucidated starting with capturing the informal description of the RFC document via the ASM ground model, then the potential ambiguous description was explicated depending on other document sources by using ASM refinement. The ASM notions helped to achieve compressible specifications for the following ambiguous behavioural parts for SASL: mechanism selection, providing mechanisms list, defining when and how the security layer can be used, and determining the maximum cypher buffer size. In addition, the main differences between RFCs and the Oracle documents were highlighted.

Returning to the main research question of this thesis, *Can ASM be combined with STPA to analyse safety aspects of critical systems and security aspects of protocols?*, it can be concluded that the combination of the ASM method and the STPA technique is possible straightaway in the safety-critical system, while this combination is possible in security protocols only after modifying the STPA technique in a way that helps to identify the possible flaws and the expected attack types against protocols. As STPA technique was designed for safety, so no surprise that it directly works for safety-critical systems. Furthermore, the STPA charming feature of inspecting how each action affects the whole system function can be utilized for security protocols, though, its focus mainly on timing conditions affects its applicability to protocols. The ASM method was a good choice for this combination due to its generality feature which is associated with simple syntax and accurate semantics to specify any system. Also, it is equipped with automatic analysis tools that facilitate our analysis goal for both critical systems and security protocols.

The main contributions for the work presented in this thesis are restated from Chapter 1 as follows:

1) A systematic methodology that combines the ASM method and the STPA technique for analysing and developing safety-critical systems.

2) The FATI method for identifying the possible protocol flaws and attack types.

3) The analysis methodology for security protocols that guides the simulation based attack scenarios with the identified attack types.

4) The precise specifications for SASL.

8.3 Future Work

During the research work of this thesis, a number of possible directions for future work have been identified. Below is a list for some of these directions concerning both safety-critical systems and security protocols.

- Precise formal semantics for STPA requirements elicitation process which can be used to build an automatic tool to support this process.
- Extending and formalizing the STPA requirements in terms of Allens interval algebra [27] which is a set of thirteen binary relationships between pairs of intervals, including *proceed*, *meet*, *overlap*, *during*, *begin*, *end*, and *equal*. These relationships can be used instead of 'too early' and 'too late' to elicit requirements about interval actions.
- An automatic tool for producing the ASM specification of the protocol under analysis directly from its abstract specification which is written in Common Syntax. This will greatly contribute to simplify the modelling and analysis processes.
- Considering the specifications of other scenarios for unaddressed attacks, such as a type-flaw attack in which honest participants misinterpret the message types, and a collaborative attack where multiple intruders can launch an attack together.
- Verifing the SASL specifications against the RFC requirements and validate these specifications against the traces produced from the Oracle implementation, using a suitable model-based testing technique.

Appendices

Appendix A

The Application of Safety and Security Techniques

This appendix explains the principle work of safety and security analysis techniques by applying the safety techniques to the Train Door Controller (TDC) example [227], while the security analysis techniques have been applied to our given example. This example is an authentication protocol for two participants: initiator and responder. The protocol aims to authenticate the responder to the initiator through exchanging two messages. In the first message, the initiator sends its identity together with a randomly chosen nonce to the responder, i.e., the initiator tells the responder, *This is my identity and I want to communicate with you. The nonce is my challenge.* Upon receipt, the responder encrypts the nonce with a secret key shared between it and the initiator. Once the second message is decrypted and the nonce checked by the initiator, this will confirm that the responder is an authenticated participant. We call this protocol is a Single Authentication Protocol (SAP).

Safety Analysis Techniques Below are the safety analysis techniques that have been described in Chapter 3.

Fault Tree Analysis (FTA). Figure A.1 shows an example of FTA for Train Door Controller system from Chapter 2, which has a door that may accidentally crush a person.

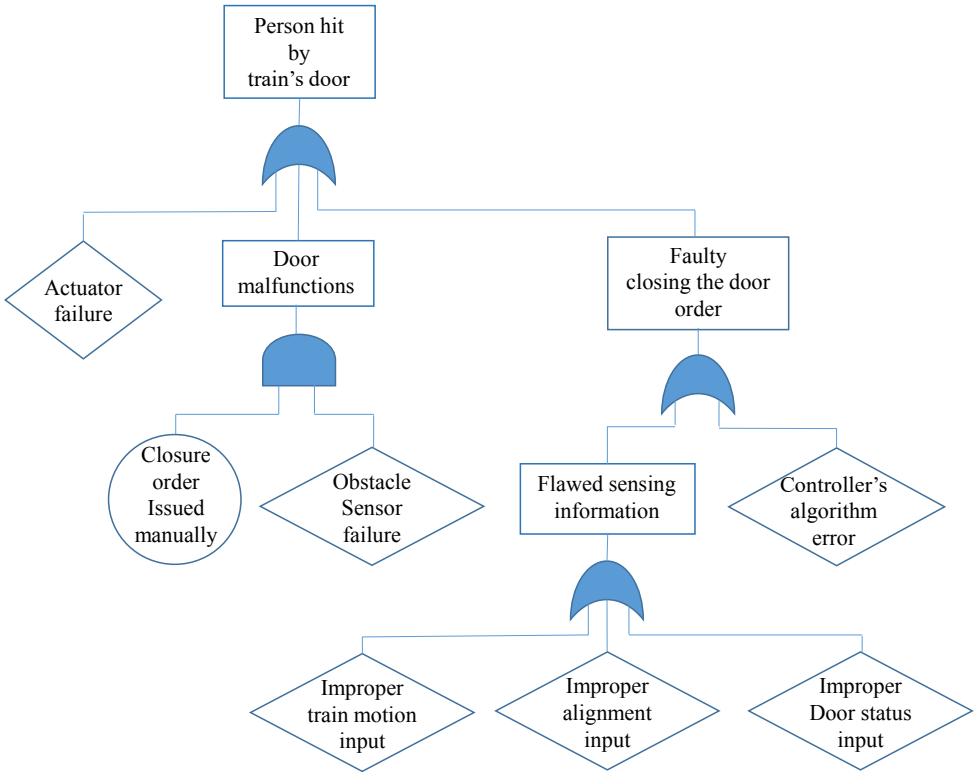

Figure A.1: Example FTA diagram

Failure Modes and Effect Analysis (FMEA). Table A.1 presents an example FMECA worksheet that gives an outline of the analysis for TDC example.

Component	Failure mode	Cause	Effect	Severity	Criticality	Recommendations
Door	Fault closure	Obstacle sensor malfunction	A person being hit by a door	Critical	Probable	The system must be provided with more than one sensor, and an alarm
Door	Stuck opening	mechanical defect	A passenger falls out of the train while moving	Catastrophic	Occasional	The train passengers must be warned at this situation

Table A.1: Example FMECA worksheet

HAZards and OPerability Analysis (HAZOP). The HAZOP analysis worksheet for

TDC example is shown in Table A.2.

Item	Parameter	Deviation-Guide Word	Cause	Consequence	Risk	Recommendations
Door	Opening	**No** opening	Mechanical or sensor defect	Passengers can not leave the train	Marginal Occasional	A door must be designed with a supervised manual door opening
Door	Opening	**Reverse** opening while a person in a doorway	Obstacle sensor defect	A person being crushed by a door	Critical Probable	The system must be provided with more than one sensor, and an alarm
Door	Opening	The door is opened **before** the train is aligned with a platform	Train position sensor malfunction	Passengers might fall out of the train	Critical Occasional	A door must not be opened while the train is not completely stopped
Door	Opening	**Late** opening of a door when there is an emergency	Emergency influencing actuator operation	Passengers could have died	Catastrophic Probable	Developing a suitable emergency safety strategy, e.g., exit choice from a window
Door	Closing	Only **part of** closure completed	Obstacle sensor defect	Passengers could fall down while train is moving	Critical Probable	A door must be designed with a supervised manual door opening
Door	Closing	Door closing **after** train moving	Door status sensor malfunction	Passengers might fall out of the train	Critical Probable	Provide the driver with a visual door status
Door	Closing	**Late** closing of the door, while the train starts to move	Door status and train motion sensors malfunction	Passengers might fall out of the train	Critical Probable	A door must be designed with a supervised manual door opening

Table A.2: Example HAZOP worksheet

Security Analysis Techniques Below are the security analysis techniques that have been described in Chapter 3.

Attack Tree (AT). The attack tree for the single authentication protocol is constructed with a goal of breaching the authentication, see Figure A.2.

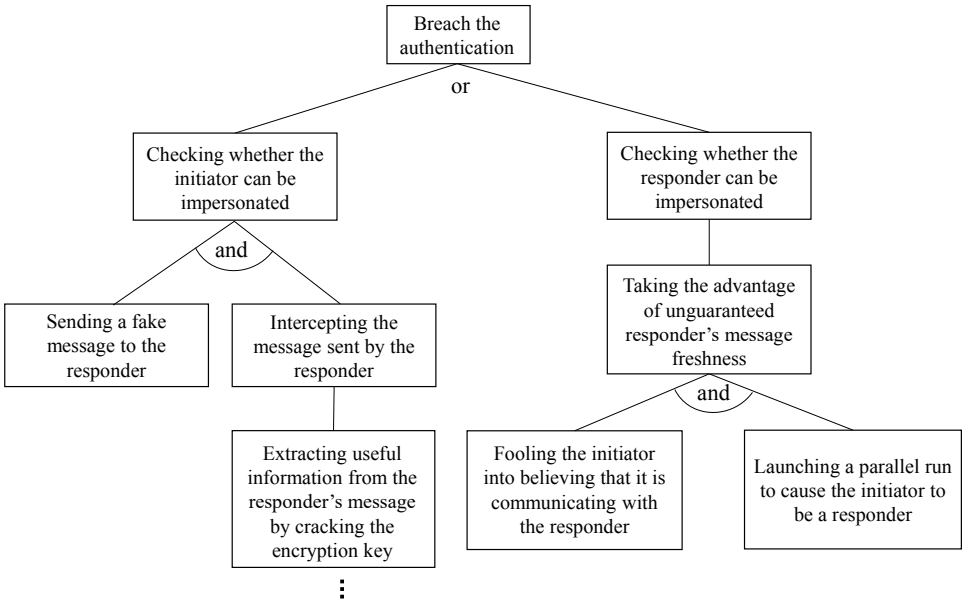

Figure A.2: An AT example

Vulnerability Identification and Analysis (VIA). The VIA process is applied to the single authentication protocol. The first step of this process is identifying the high level functional requirements. These requirements for the single authentication protocol are:

- The none and the initiator's identity must be sent to the responder.
- The responder should securely send back an encrypted nonce to the initiator.
- The initiator approves that the responder is an authenticated participant.

The next setp is analysing each requirement separately. The analysis for the first requirements is shown in Table A.3.

Guide word-Deviation	Causes	Effects	Recommendations
Omission **No nonce and identity sent by the initiator**	1) The initiator sends an incorrect message 2) The network blocks the message 3) The attacker blocks the message	1) The initiator does not know that the responder has not received the message 2) The responder does not know that the message has been sent 3) The message is blocked	Using a determined time-out by the initiator and the responder at the waiting for messages
Inclusion (spurious) **The responder unexpectedly receives the nonce and the identity**	1) The attacker sends a fake message 2)The network misdirects a message	The responder may provide authenticated information to a participant has an invalid identity	1) The responder's identity should be explicitly stated 2)The responder should be able to authenticate the received message
Inclusion (repetition) **the message that includes identity and nonce is repeated**	1) The message is replayed by the attacker 2) The message is duplicated by the network	1) The responder believes that it is communicating with a true participant 2) The attacker may be able to breach the authentication	The responder should have a mechanism to limit the number of the received messages
Value **The value of the identity and/or the nonce is changed**	1) The attacker takes advantage of sending the identity and the nonce as a plaintext message 2) The value is corrupted during the transmission	The attacker may be able to trick the responder into using incorrect challenge for authentication purpose	The responder should carry out message integrity test
Disclosure (external) **The identity and the nonce are disclosed to external participant**	The identity and the nonce are not protected when created	The attacker is able to read the nonce	The identity and the nonce should be encrypted
Early (absolute) **The identity and the nonce are received early**	See Inclusion(spurious)		
Late (absolute) **The identity and the nonce are received late**	1) Intercepted by the attacker and then sent back 2) The network causes delayed	1) The responder may use the nonce to authenticate itself to the claimed identity 2) The attacker may impair the authentication	The responder should carry out freshness checks by encrypting the none

Table A.3: Analysis of VIA for the requirement: The nonce and the initiator's identity must be sent to the responder

Requirements Analysis and Elicitation (REA). We apply the REA tree process to the single authentication protocol, see Figure A.3. The root for the tree in Figure A.3 tree is: The authentication is not guaranteed, which is obtained by negating the main protocol goal. The oval shape in this tree represents the assumption which is used for the analysis.

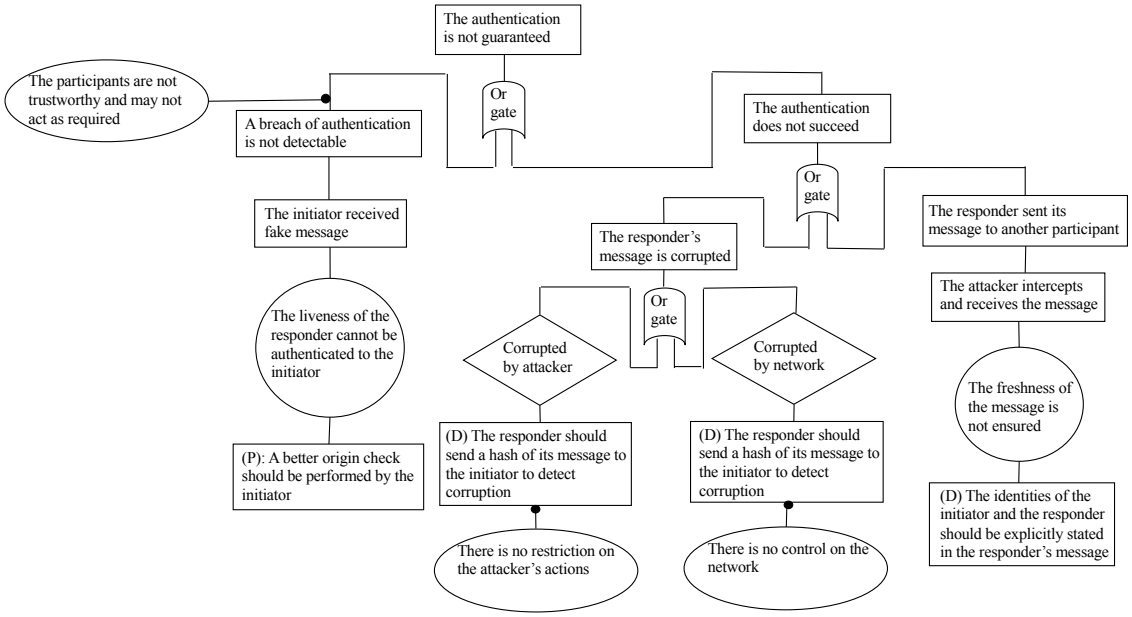

Figure A.3: REA tree analysis for the requirement: The authentication is not guaranteed

Appendix B

The Corresponding Requirements for the IPCS Scenarios

This appendix details, in Tables B.1 and B.2, the informal requirements which correspond to the scenarios that have been used to validate the model for the IPCS, and the main values that have been checked through these scenarios besides.

Scenario no.	Corresponding Requirements	Checked Values
s0	No insulin dose must be delivered when the current glucose value is less than the minimum safe limit	The dose=0 when the current glucose value is 4, i.e, cR=4 ((cR=4)<(sMin=6))
s1	The minimum dose must be delivered when the current glucose value is greater than the maximum safe limit, the glucose level is stable, and the cumulative dose is less than the maximum daily dose	The dose=1 when the current glucose value is 16, i.e, cR=16 ((cR=16)>(sMax=14)) and sugar level is stable cR=pR=16, where pR is a reading 10 minutes before
s2	If the current glucose value is greater than the maximum safe limit, the glucose level is increasing, the rounded division of the glucose level by the maximum single dose is greater than zero, and the cumulative dose is less than the maximum daily dose then the delivered dose must be equal to the rounded division result	The dose=round((cR-pR)/4) when the cR=19, and glucose level is increasing (cR=19)>(pR=14), and round((19-14)/4)>0
s3	If the current glucose value is greater than the maximum safe limit, and the glucose level is increasing, the rounded division of the glucose level on the maximum single dose is equal to zero, and the cumulative dose is less than the maximum daily dose then the delivered dose must be equal to the minimum dose	The dose=1 when the cR=15, and glucose level is increasing (cR=15)>(pR=14), and round((15-14)/4)=0
s4	No insulin dose must be delivered when the current glucose value is greater than the maximum safe limit, the glucose level is declining, and the decrease rate is rising	The dose=0 when the cR=17, and glucose level is decreasing (cR=17)<(pR=18), and the decrease rate is rising ((cR=17)-(pR=18))<= ((pR=18)-(oR=19)), where oR is the reading 20 minutes before
s5	Minimum dose must be delivered when the current glucose value is greater than the maximum safe limit, the glucose level is declining, the decrease rate is declining, and the cumulative dose is less than the maximum daily dose	The dose=1 when the cR=15, and glucose level is falling (cR=15)<(pR=16), and the decrease rate is decreasing ((cR=15)-(pR=16))<= ((pR=16)-(oR=18))
s6	No insulin dose must be delivered when the current glucose value is within the safe limits, and the glucose level is steady or declining	The dose=0 when the (cR=13)>=(sMin=6), and (cR=13)<=(sMax=14), and glucose level is satble (cR=13)<=(pR=13)

Table B.1: Corresponding requirements of the successful scenarios for the IPCS case study-Part 1

Scenario no.	Corresponding Requirements	Checked Values
s7	No insulin dose must be delivered when the current glucose value is within the safe limits, the glucose level is rising, but the increase rate is declining	The dose=0 when the (cR=13)>=(sMin=6), and (cR=13)<=(sMax=14), and glucose level is increasing (cR=13)>=(pR=12), and the increase rate is decreasing ((cR=13)-(pR=12)< ((pR=12)-(oR=10))
s8	Minimum dose must be delivered when the current glucose value is within the safe limits, the glucose level is rising, the increase rate is rising, the rounded division of the glucose level on the maximum single dose is equal to zero, and the cumulative dose is less than the maximum daily dose	The dose=1 when the (cR=12)>=(sMin=6), and (cR=12)<=(sMax=14), and glucose level is increasing (cR=12)>=(pR=10), and the increase rate is rising ((cR=12)-(pR=10)>= ((pR=10)-(oR=14)), and round((12-10)/4)=0
s9	If the current glucose value is within the safe limits, the glucose level is rising, the increase rate is rising, the rounded division of the glucose level on the maximum single dose is greater than zero, and the cumulative dose is less than the maximum daily dose then the delivered dose must be equal to the rounded division result	The dose=round((14-9)/4) when the (cR=14)>=(sMin=6), and (cR=14)<=(sMax=14), and glucose level is increasing (cR=14)>=(pR=9), and the increase rate is rising ((cR=14)-(pR=9)>= ((pR=9)-(oR=4)), and round((14-9)/4)>0
s10	If the summation of the computed dose plus the current cumulative dose is greater than the maximum dose per day, then the delivered dose must be equal to subtraction the cumulative dose from the maximum dose per day	The dose=mDD-cD when the (cR=34)>=(sMax=14), and (cD=23), and comD=(34-18)/4=4, and (comD=4)+(cD=23) >(mDD=25), where mDD is the maximum dialy dose, cD is the cumulative dose, and comD is the computed dose
s11	The delivered dose must be equal to the computed dose, if delivering it will not exceed the maximum daily dose, and the computed dose itself is less than or equal the maximum single dose	The dose=comD when the (comD=3)<=(mSD=4), and (comD=3)+(cD=18)< (mDD=25) where mSD is the maximum single dose
s12	If the summation of the computed dose plus the current cumulative is less than the maximum dose per day, and the computed dose itself is greater than the maximum dose per delivering then the delivered dose must be equal to the maximum dose per day	The dose=mSD when the (comD=5)>(mSD=4), and (comD=5)+(cD=0)< (mDD=25) where comD=((cR=24)-(pR=14))/4

Table B.2: Corresponding requirements of the successful scenarios for the IPCS case study-Part 2

Appendix C

The Ground model for the Rest Steps of the Second Phase for SASL

This appendix presents the ground models for the rest steps performed by both client and server for SASL example in the authentication negotiation phase.

Client-Side. Figure C.1 shows the ground model for performing the rest steps in the authentication negotiation phase. The client starts that when it receives a challenge from the server. If so, the client will check if it is waiting for one of the following: outcome, *AuthData*, *abortReply*, or *addAuthData*. When it is waiting for *outcome*, i.e., the outcome of sending the authentication command together with the selected mechanism and the initial response, the client will check whether it received a successful outcome or not. If not, the client goes to the *'Failure'* state. While if it is successful, the client inspects that the selected mechanism supports the security layer and the quality of the protection field in the challenge contains *auth-int* or *auth-conf*, to goes to the security layer phase; otherwise, the client goes to the *'Successful'* state to indicate that the authentication phase is finished successfully.

When the client is waiting for authentication data (*AuthData*), then the client will implement the *'Challenge processing'* rule to process the received data, which we will detail it down. While in case that it is waiting for an abort reply *'abortReply'*, as it sent in the mechanism negotiation phase an abort response when there is no supported

mechanism by it in the received list, the client will check if it is received an abort reply challenge. If so, then the client goes to the *'Abort'* state; otherwise, it will return back to the *'Wait challenge'* state.

In case that the client is waiting for additional authentication data (*addAuthData*), the client will inspect whether the received challenge contains additional data. if so, the client will verify them, when the verification output is successful, the client checks that the additional authentication data is attached with successful authentication outcome. If true, the client goes to the security layer phase, but if it is not attached, the client sends an empty response to the server and waits for an authentication outcome from the server. When the received challenge does not contain additional authentication data or it contains that but the verification of data failed, the client sends an abort reply to the server.

The *Challenge processing* rule is shown in Figure C.2. The client processes the receiver challenge by firstly checking that this challenge is an *Abort reply*, *Server first*, or *Failure decoding*, in order to release the connection and go to the *'Abort'* state. When the challenge is not of any the previous types, the client makes sure that it must process the first challenge. If so, the client checks if it receives a challenge indicating that the server does not support the mechanism that has been selected by the client. When this indication is received, the client will select another mechanism and send another authentication command by repeating the initial step in the authentication negotiation phase. When the received challenge does not contain unsupported mechanism, the client checks if it is an empty challenge, then it sends its initial response and it waits for the authentication outcome when the mechanism does not need more authentication data; otherwise, it waits for an authentication challenge. In case that the client is not waiting for an empty or the first challenge, then the mechanism must be checked whether it needs to very the receiver challenge. If true, then the client verifies the received challenge. If the challenge is not authenticated, then the client sends an abort reply. While, when it is authenticated or the mechanism does not require to perform the verification process, the client sends a response to the server and waits for an authentication challenge when there are more challenges to exchange. However, when the client must wait for a last challenge from the server, then this challenge must be either an additional authentication data or authentication outcome depending on whether the mechanism supports the additional authentication data or not, respectively.

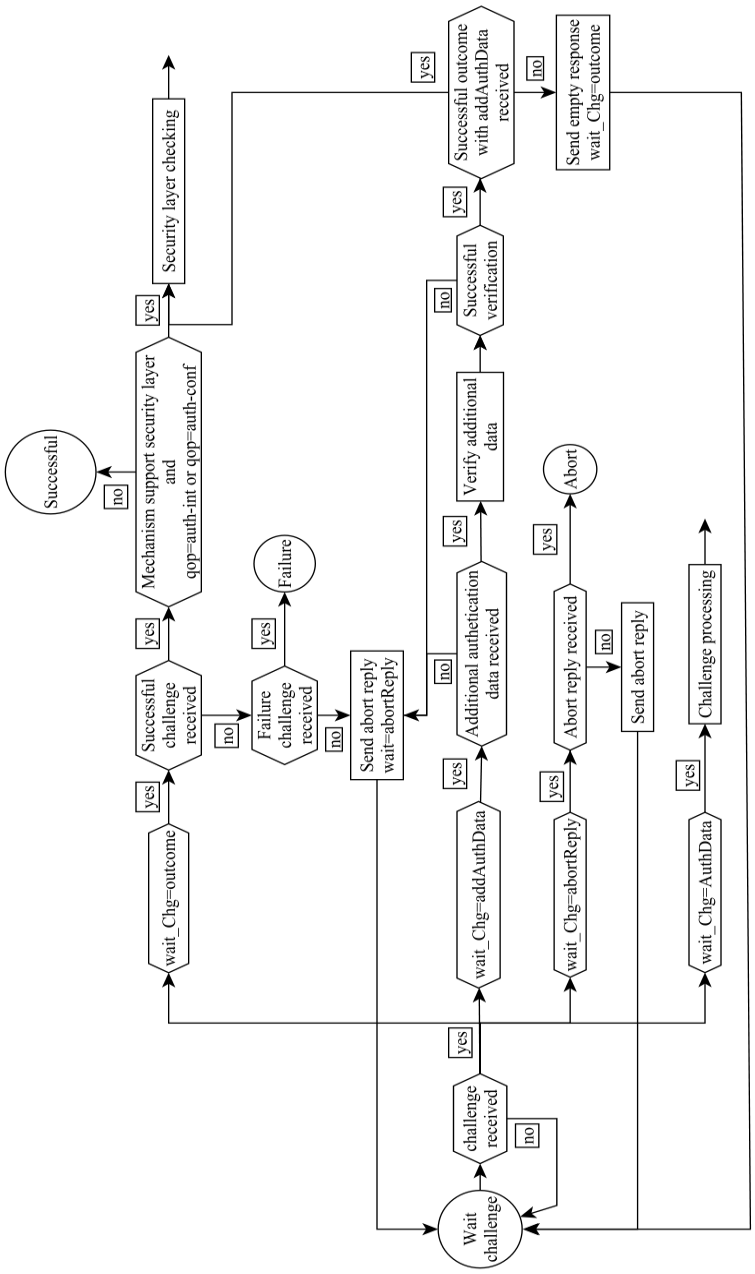

Figure C.1: Client side for performing rest steps in the authentication negotiation phase - ground model

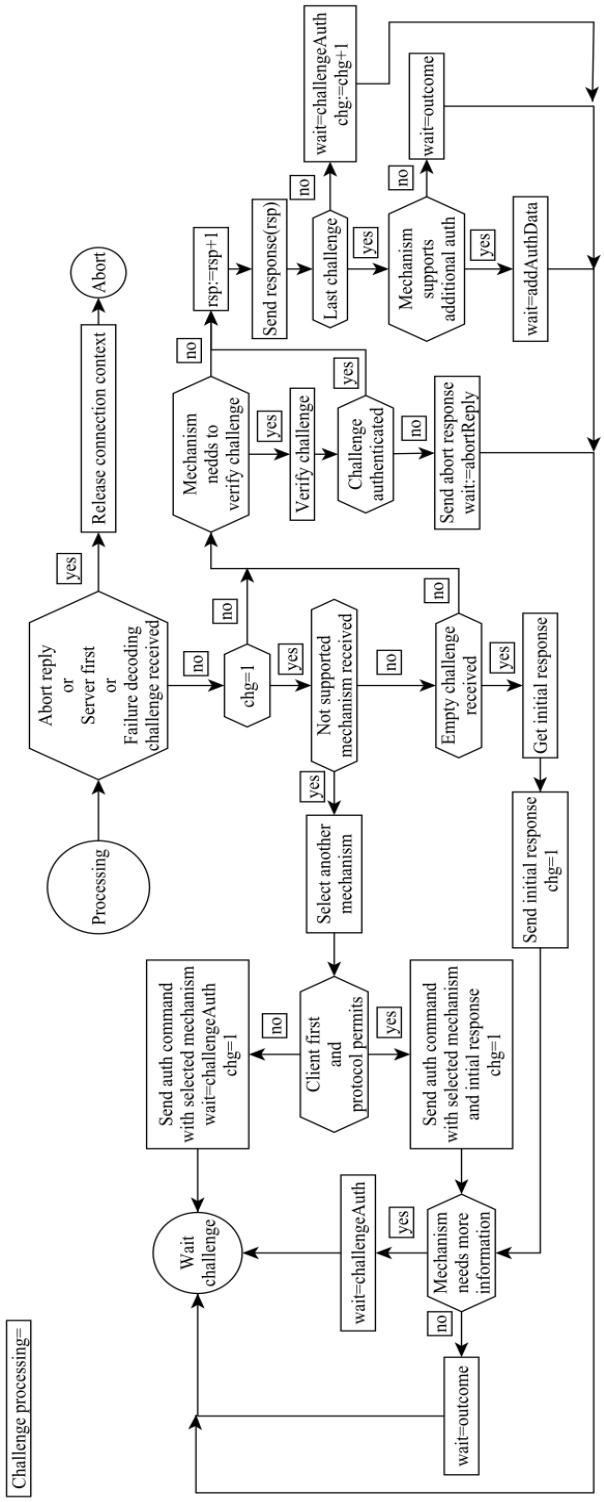

Figure C.2: The *Challenge processing* rule - ground model

Server-Side. Figure C.3 shows the ground model for performing the rest steps in the authentication negotiation phase. The server starts these steps when it receives a response from the client. At receiving the time, the server checks if the response is an *'Abort authentication'*, then the server in parallel sends an abort reply and releases the connection to enter the *'Abort'* state. However, when the received response is empty and the mechanism specifies that the server must wait for an empty response, the server will send a successful outcome and go to the security layer negotiation phase. In case that the mechanism does not specify that the server must wait for an empty challenge, the server will send a failure outcome and go to the *Failure'* state. when the reserver does not receive an empty or abort authentication response, it will call the *'Response processing'* rule to handle the received response.

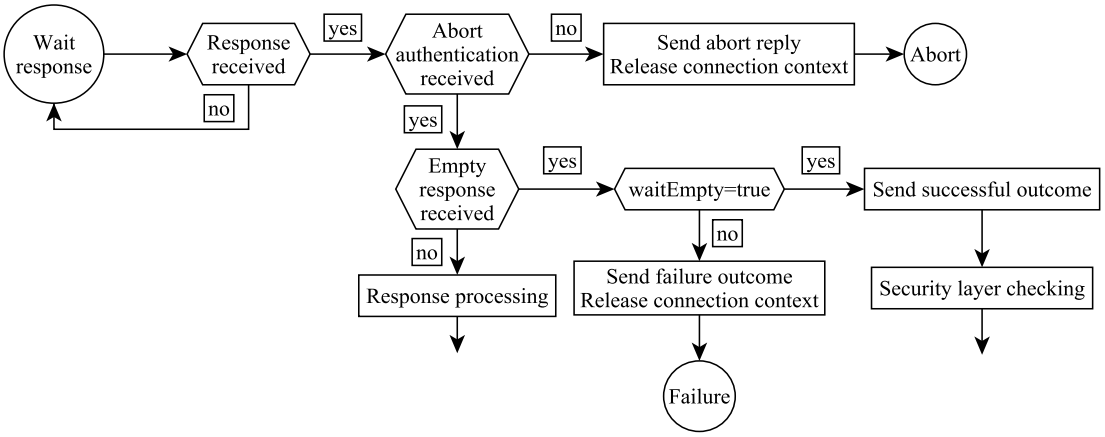

Figure C.3: Server side for performing rest steps in the authentication negotiation phase - ground model

The *Response processing* rule is shown in Figure C.4. The server processes the received response by firstly decoding this response. When the server fails to decode the receiver response, it will in parallel send failure decoding challenge to the client and release the connection to enter the *'Abort'* state. In case that the server succeeds at decoding the response, it will verify the response. When the verification failed, the server sends failure outcome to the client, releases the connection, and enter the *Failure'* state. However,

when the verification succeeded, the server checks if the mechanism determines to send a challenge. If so, the server sends the challenge and enters the *'Wait response'* state. When the mechanism does not determine to send a challenge, the server checks that the mechanism does not require to send addition authentication data, or it requires that but the protocol does not permit, then it sends successful outcome and goes to the security layer negotiation phase. While when the mechanism requires to send addition authentication data and the protocol permits to perform that, then the server sends additional authentication data to the client and goes to the *'Wait response'* state.

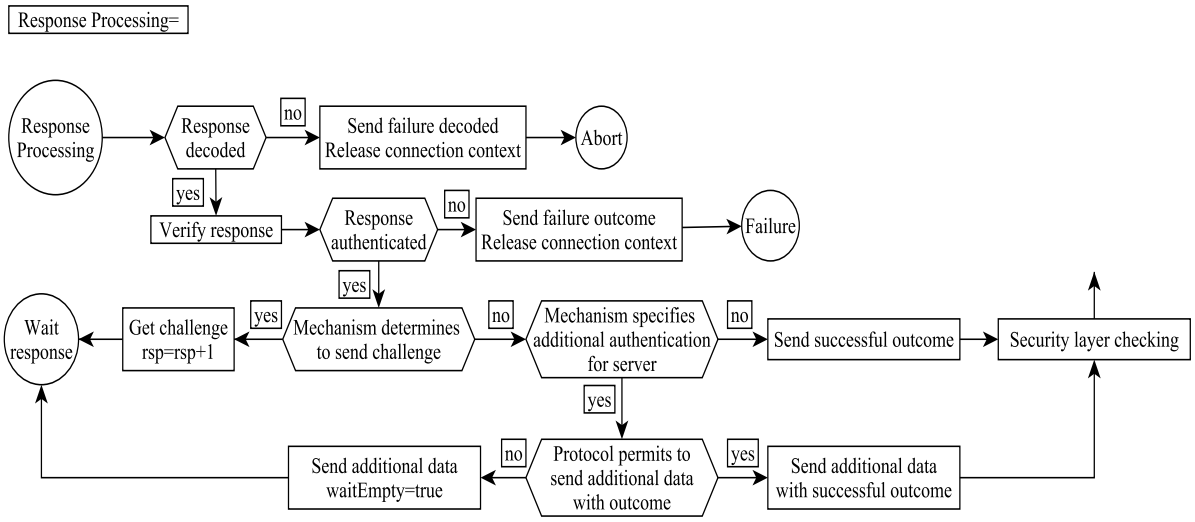

Figure C.4: The *Response processing* rule - ground model

Bibliography

[1] Eclipse Platform Technical Overview. https://www.eclipse.org/articles/Whitepaper-Platform-3.1/eclipse-platform-whitepaper.pdf. Accessed: 2019-02-10.

[2] The ASM Workbench. http://www.uni-paderbom.de/cs/asm/ASMToolPage/asm-workbench.html. Accessed: 2019-02-17.

[3] The ASMETA framework. http://asmeta.sourceforge.net/. Accessed: 2019-02-21.

[4] The AsmL Compiler. http://www.codeplex.com/AsmL. Accessed: 2019-02-1.

[5] The CoreASM Project. http://www.coreasm.org. Accessed: 2019-02-20.

[6] The XASM Open Source Project. http://www.xasm.org. Accessed: 2019-02-19.

[7] The Object Management Group (OMG). http://www.omg.org. Accessed: 2019-02-21.

[8] The ProVerif Project. https://prosecco.gforge.inria.fr/personal/bblanche/proverif/. Accessed: 2019-05-12.

[9] The Tamarin Prover. https://tamarin-prover.github.io/. Accessed: 2019-05-14.

[10] The TASM Toolset. http://esl.mit.edu/tasm. Accessed: 2019-01-25.

[11] The Timed ASM Simulator Project. http://vpax.narod.ru/asmsim.html. Accessed: 2019-02-15.

[12] Procedures for performing a failure mode, effects and criticality analysis. Military Standard MIL-STD-1629A, Department of Defense, Washington, DC 20301, November 1980.

[13] Executing ASM specifications with AsmGofer. `http://www.tydo.de/AsmGofer`, 1999. Accessed: 2019-02-20.

[14] Mearns A. Fault tree analysis- The study of unlikely events in complex systems. In *System Safety Symposium*. University of Washington and the Boeing Company, Seattle, Washington, 1965.

[15] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In *Acm Sigplan Notices*, volume 36, pages 104–115. ACM, 2001.

[16] Asim Abdulkhaleq and Stefan Wagner. Experiences with applying STPA to software-intensive systems in the automotive domain. 2013.

[17] Asim Abdulkhaleq and Stefan Wagner. Integrated safety analysis using systems-theoretic process analysis and software model checking. In *International Conference on Computer Safety, Reliability, and Security*, pages 121–134. Springer, 2015.

[18] Asim Abdulkhaleq and Stefan Wagner. XSTAMPP: An extensible STAMP platform as tool support for safety engineering. In *2015 STAMP Workshop, MIT, Boston, USA*. Stuttgart University, 2015.

[19] Asim Abdulkhaleq and Stefan Wagner. A Systematic and Semi-Automatic Safety-Based Test Case Generation Approach Based on Systems-Theoretic Process Analysis. *arXiv preprint arXiv:1612.03103*, 2016.

[20] Jean-Raymond Abrial. *The B-book: Assigning programs to meanings*. Cambridge University Press, 1996.

[21] Shreya Adyanthaya, Shilpa Rukmangada, Amrish Tiwari, and Sanjay Singh. Modeling freshness concept to overcome replay attack in Kerberos protocol using NuSMV. In *Computer and Communication Technology (ICCCT), 2010 International Conference on*, pages 125–129. IEEE, 2010.

[22] Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon. The AsmetaL specifications for five security protocols and five attack scenarios. `https://doi.org/10.5281/zenodo.2628743`.

[23] Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon. The refined model for the SASL case study based on Oracle implementation documentation. `https://doi.org/10.5281/zenodo.1204242`.

[24] Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon. The refined model for the SASL case study based on RFC 2222/4422. `https://doi.org/10.5281/zenodo.1204257`.

[25] Farah Al-Shareefi, Alexei Lisitsa, and Clare Dixon. The insulin pump control system: Specification and scenarios. `https://doi.org/10.5281/zenodo.3242126`. Accessed: 2019-06-9.

[26] Homa Alemzadeh, Daniel Chen, Andrew Lewis, Zbigniew Kalbarczyk, Jaishankar Raman, Nancy Leveson, and Ravishankar Iyer. Systems-theoretic safety assessment of robotic telesurgical systems. In *International conference on computer safety, reliability, and security*, pages 213–227. Springer, 2014.

[27] James F Allen. Maintaining knowledge about temporal intervals. In *Communications of the ACM*, volume 26, page 832843, 1983.

[28] Matthias Anlauff. XASM-An extensible, component-based Abstract State Machines language. In *Abstract State Machines-Theory and Applications*, pages 69–90. Springer, 2000.

[29] Blandine Antoine. *Systems Theoretic Hazard Analysis (STPA) applied to the risk review of complex systems: an example from the medical device industry.* PhD thesis, Massachusetts Institute of Technology, 2013.

[30] Paolo Arcaini, Silvia Bonfanti, Angelo Gargantini, Atif Mashkoor, and Elvinia Riccobene. Formal validation and verification of a medical software critical component. In *Formal Methods and Models for Codesign (MEMOCODE), 2015 ACM/IEEE International Conference on*, pages 80–89. IEEE, 2015.

[31] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. AsmetaSMV: a way to link high-level ASM models to low-level NuSMV specifications. In *International Conference on Abstract State Machines, Alloy, B and Z*, pages 61–74. Springer, 2010.

[32] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Automatic review of abstract state machines by meta-property verification. In *NASA Formal Methods Symposium*, pages 4–13. NASA, 2010.

[33] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Modeling and analyzing using ASMs: the landing gear system case study. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 36–51. Springer, 2014.

[34] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. SMT-based automatic proof of ASM model refinement. In *14th International Conference on Software Engineering and Formal Methods*, pages 253–269. Springer, 2016.

[35] Paolo Arcaini, Angelo Gargantini, and Elvinia Riccobene. Rigorous development process of a safety-critical system: from ASM models to Java code. *International Journal on Software Tools for Technology Transfer*, 19(2):247–269, 2017.

[36] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. *Software: Practice and Experience*, 41(2):155–166, 2011.

[37] Paolo Arcaini, Roxana-Maria Holom, and Elvinia Riccobene. ASM-based formal design of an adaptivity component for a Cloud system. *Formal Aspects of Computing*, 28(4):567–595, 2016.

[38] Varsha Awhad and Charles Wallace. A unified formal specification and analysis of the new Java memory models. In *International Workshop on Abstract State Machines*, pages 166–185. Springer, 2003.

[39] Nikhil Balakrishnan. *Fault Tree Analysis for Medical Applications*, pages 83–112. Springer, 2015.

[40] Mike Barnett, Egon Börger, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Using abstract state machines at Microsoft: A case study. In *Abstract State Machines-Theory and Applications*, pages 367–369. Springer, 2000.

[41] Danièle Beauquier and Anatol Slissenko. A first order logic for specification of timed algorithms: Basic properties and a decidable class. *Annals of Pure and Applied Logic*, 113(1-3):13–52, 2001.

[42] Jörg Beckers, Daniel Klünder, Stefan Kowalewski, and Bastian Schlich. Direct support for model checking abstract state machines by utilizing simulation. In *International Conference on Abstract State Machines, B and Z*, pages 112–124. Springer, 2008.

[43] Gerd Behrmann, Alexandre David, and Kim G Larsen. A tutorial on Uppaal. In *Formal methods for the design of real-time systems*, pages 200–236. Springer, 2004.

[44] Christine M Belcastro. Validation and verification techniques and tools. *Encyclopedia of Systems and Control*, pages 1511–1517, 2015.

[45] Giampaolo Bella and Elvinia Riccobene. Formal analysis of the Kerberos authentication system. *Journal of Universal Computer Science*, 3(12):1337–1381, 1997.

[46] Giampaolo Bella and Elvinia Riccobene. A realistic environment for crypto-protocol analyses by ASMs. In *Workshop on Abstract State Machines*, pages 127–138, 1998.

[47] Noomene Ben Henda. Generic and efficient attacker models in SPIN. In *SPIN Symposium on Model Checking of Software*, pages 77–86. ACM, 2014.

[48] Daniel Lee Berre, Anne Parrain, and Lakhdar Sais. SAT4J: A Satisfiability Library for Java. Accessed: 2019-02-5.

[49] Steve Bishop, Matthew Fairbairn, Michael Norrish, Peter Sewell, Michael Smith, and Keith Wansbrough. Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations. pages 55–66. ACM Press, 2006.

[50] Dines Bjrner and Cliff B Jones. *The Vienna Development Method: The Meta-Language*. Springer-Verlag, 1978.

[51] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *14th IEEE Computer Security Foundations Workshop*, pages 82–96. IEEE, 2001.

[52] Bruno Blanchet. Security protocol verification: Symbolic and computational models. In *Proceedings of the First international conference on Principles of Security and Trust*, pages 3–29. Springer-Verlag, 2012.

[53] Bruno Blanchet. Modeling and verifying security protocols with the applied pi calculus and ProVerif. *Foundations and Trends® in Privacy and Security*, 1(1-2):1–135, 2016.

[54] Bruno Blanchet, Ben Smyth, Vincent Cheval, and Marc Sylvestre. ProVerif 2.00: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial. `http://www.proverif.ens.fr/`, 2018.

[55] Silvia Bonfanti, Marco Carissoni, Angelo Gargantini, and Atif Mashkoor. Asm2C++: a tool for code generation from abstract state machines to Arduino. In *NASA Formal Methods Symposium*, pages 295–301. Springer, 2017.

[56] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language User Guide*. Addison Wesley, 1999.

[57] Egon Börger. A Logical Operational Semantics for Full Prolog. Part I: Selection Core and Control. *Lecture Notes in Computer Science*, 440:36–64, 1990.

[58] Egon Börger. A logical operational semantics for full Prolog. Part II: Built-in predicates for database manipulations. In *Mathematical Foundations of Computer Science*, pages 1–14. Springer, 1990.

[59] Egon Börger. Modeling distributed algorithms by Abstract State Machines compared to Petri Nets. In *International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z*, pages 3–34. Springer, 2016.

[60] Egon Börger and Alexander Raschke. *Modeling Companion for Software Practitioners*. Springer, Heidelberg, 2018.

[61] Egon Börger, Elvinia Riccobene, and Joachim Schmid. Capturing requirements by Abstract State Machines: The Light Control Case Study. *Journal of Universal Computer Science*, 6(7):597–620, 2000.

[62] Egon Börger and Dean Rosenzweig. A mathematical definition of full Prolog. *Science of Computer Programming*, 24(3):249–286, 1995.

[63] Egon Börger and Wolfram Schulte. A programmer friendly modular definition of the semantics of Java. In *Formal Syntax and Semantics of Java*, pages 353–404. Springer, 1999.

[64] Egon Börger and Robert Stärk. *Abstract State Machines: A method for high-level system design and analysis*. Springer, Heidelberg, 2003.

[65] Stanislaw Budkowski and Piotr Dembinski. An introduction to Estelle: a specification language for distributed systems. *Computer Networks and ISDN systems*, 14(1):3–23, 1987.

[66] D.J. Burns and R.M. Pitblado. A modified HAZOP methodology for safety critical system assessment. In *Directions in Safety-critical Systems*, pages 232–245. Springer, 1993.

[67] Michael Burrows, Martin Abadi, and Roger Michael Needham. A logic of authentication. *Proceedings of the Royal Society of London. A. Mathematical and Physical Sciences*, 426(1871):233–271, 1989.

[68] Dominique Cansell and Dominique Mery. Tutorial on the event-based B method: Concepts and case studies. In *International Conference on Formal Methods for Networked and Distributed Systems*. FORTE, 2006.

[69] Alessandro Carioni, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A scenario-based validation language for ASMs. In *International Conference on Abstract State Machines, B and Z*, pages 71–84. Springer, 2008.

[70] Steven C Cater and James K Huggins. An ASM dynamic semantics for standard ML. In *International Workshop on Abstract State Machines*, pages 203–222. Springer, 2000.

[71] Alessandra Cavarra and Elvinia Riccobene. Simulating UML statecharts. *Formal Methods and Tools for Computer Science (Proceedings of Eurocast 2001)*, pages 224–227, 2001.

[72] Roxanaä Maria Chelemen. Modeling a web application for cloud content adaptation with ASMs. In *Cloud Computing and Big Data (CloudCom-Asia), International Conference on*, pages 44–55. IEEE, 2013.

[73] Shengbo Chen, Hao Fu, and Huaikou Miao. Formal verification of security protocols using SPIN. In *2016 IEEE/ACIS 15th International Conference on Computer and Information Science (ICIS)*, pages 1–6. IEEE, 2016.

[74] S.K. Chen, T.K. Ho, and B.H. Mao. Reliability evaluations of railway power supplies by fault-tree analysis. *IET Electric Power Applications*, 1(2):161–172, 2007.

[75] Alessandro Cimatti, Edmund Clarke, Fausto Giunchiglia, and Marco Roveri. NuSMV: A new symbolic model verifier. In *International conference on computer aided verification*, pages 495–499. Springer, 1999.

[76] CISHEC. *A Guide to Hazard and Operability Studies*. Chemical Industry Safety and Health Council of the Chemical Industries, 1977.

[77] John Clark and Jeremy Jacob. Attacking authentication protocols. *High Integrity Systems*, 1:465–474, 1996.

[78] Edmund M Clarke and E Allen Emerson. Design and synthesis of synchronization skeletons using branching time temporal logic. In *Workshop on Logic of Programs*, pages 52–71. Springer, 1981.

[79] Edmund M Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT press, 1999.

[80] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996.

[81] International Electrotechnical Commission et al. *Analysis Techniques for System Reliability: Procedure for Failure Mode and Effects Analysis (FMEA)*. International Electrotechnical Commission, 2006.

[82] Kevin Compton, Yuri Gurevich, James Huggins, and Wuwei Shen. An automatic verification tool for UML. *Univ. of Michigan, EECS Dept. Tech. Report CSE-TR-423*, 2000.

[83] Véronique Cortier, Stéphanie Delaune, and Pascal Lafourcade. A survey of algebraic properties used in cryptographic protocols. *Journal of Computer Security*, 14(1):1–43, 2006.

[84] Frank Crawley, Malcolm Preston, and Brian Tyler. HAZOP: Guide to best practice. Guidelines to best practice for the process and chemical industries. European Process Safety Centre. *Chemical Industries Association & Institution of Chemical Engineers. Rugby, England, IChem*, 2000.

[85] Basin David, Cremers Cas, Dreier Jannik, Meier Simon, Sasse Ralf, and Schmidt Benedikt. Tamarin-Prover manual: Security protocol analysis in the symbolic model. `https://tamarin-prover.github.io/manual/index.html`, 2019.

[86] Giuseppe Del Castillo. The ASM workbench: an open and extensible tool environment for Abstract State Machines. In *Workshop on Abstract State Machines*, pages 139–154, 1998.

[87] Giuseppe Del Castillo. Towards comprehensive tool support for Abstract State Machines: The ASM workbench tool environment and architecture. In *International Workshop on Current Trends in Applied Formal Methods*, pages 311–325. Springer, 1999.

[88] Thomas W DeLong. A fault tree manual. Technical report, Industrial Engineering Department, Texas A&M University, 1970.

[89] Dorothy E Denning and Giovanni Maria Sacco. Timestamps in key distribution protocols. *Communications of the ACM*, 24(8):533–536, 1981.

[90] Whitfield Diffie and Martin Hellman. New directions in cryptography. *IEEE transactions on Information Theory*, 22(6):644–654, 1976.

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

[92] Ling Dong and Kefei Chen. Engineering principles for security design of protocols. In *Cryptographic Protocol*, pages 41–81. Springer, 2012.

[93] Ben Donovan, Paul Norris, and Gavin Lowe. Analyzing a library of security protocols using Casper and FDR. In *Proceedings of the Workshop on Formal Methods and Security Protocols*, pages 36–43. Citeseer, 1999.

[94] Saeed Doostali. An Efficient Solution for Model Checking Abstract State Machine Using Bogor. *arXiv preprint arXiv:1404.2155*, 2014.

[95] Clifton A Ericson. Fault tree analysis - a history. In *Proceedings of the 17th International System Safety Conference, Orlando, Florida*, volume 1, pages 1–9, 1999.

[96] Clifton A. Ericson. *Hazard analysis techniques for system safety.* John Wiley & Sons, 2nd edition, 2005.

[97] Roozbeh Farahbod. CoreASM language user manual. *URL: http://coreasm. svn. sourceforge. net/viewvc/coreasm/engine-carma/trunk/doc/user_ manual/CoreASM-UserManual. pdf*, 2006.

[98] Roozbeh Farahbod. *CoreASM: an extensible modeling framework & Tool environment for high-level design and analysis of distributed systems.* PhD thesis, Simon Fraser University, 2009.

[99] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. Design and specification of the CoreASM execution engine. Technical report, SFU-CMPT-TR-2005-02, Simon Fraser University, 2005.

[100] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. CoreASM: An extensible ASM execution engine. *Fundamenta Informaticae*, 77(1-2):71–103, 2007.

[101] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glässer. Executable formal specifications of complex distributed systems with CoreASM. *Science of Computer Programming*, 79:23–38, 2014.

[102] Roozbeh Farahbod, Uwe Glässer, and George Ma. Model checking CoreASM specifications. In *Proceedings of the 14th International ASM Workshop (ASM 2007)*, 2007.

[103] Thomas A Ferguson and Lixuan Lu. Fault tree analysis for an inspection robot in a nuclear power plant. In *IOP Conference Series: Materials Science and Engineering*, volume 235, page 012003. IOP Publishing, 2017.

[104] Michael Fisher. *An introduction to practical formal methods using temporal logic.* John Wiley & Sons, 2011.

[105] Cody Harrison Fleming, Melissa Spencer, Nancy Leveson, and Chris Wilkinson. Safety Assurance in NextGen. Technical Report NASA/CR-2012-217553, NF1676L-13918, Massachusetts Institution of Technology; Cambridge, MA, United States, 2012.

[106] Bradley Fordham, Serge Abiteboul, and Yelena Yesha. Evolving databases: An application to electronic commerce. In *International Database Engineering and Applications Symposium (IDEAS)*, pages 191–200. IEEE, 1997.

[107] Nathalie Foster and Jeremy Jacob. Hazard analysis for security protocol requirements. In *Advances in Network and Distributed Systems Security*, pages 75–92. Springer, 2002.

[108] Nathalie Louise Foster. *The application of software and safety engineering techniques to security protocol development*. PhD thesis, York University, 2002.

[109] Peter Froome and Brian Monahan. The role of mathematically formal methods in the development and assessment of safety-critical systems. *Microprocessors and Microsystems*, 12(10):539–546, 1988.

[110] Angelo Gargantini, Elvinia Riccobene, and Salvatore Rinzivillo. Using Spin to generate tests from ASM specifications. In *International Workshop on Abstract State Machines*, pages 263–277. Springer, 2003.

[111] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Model-driven language engineering: The ASMETA case study. In *International Conference on Software Engineering Advances*, pages 373–378. IEEE, 2008.

[112] Angelo Michele Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based language and a simulation engine for Abstract State Machines. *Journal of Universal Computer Science*, 14(12):1949–1983, 2008.

[113] Li Gong, Roger Needham, and Raphael Yahalom. Reasoning about belief in cryptographic protocols. In *Proceedings. 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, pages 234–248. IEEE, 1990.

[114] Yuri Gurevich. Evolving algebras 1993: Lipari guide. *Oxford University Press*, 324:9–36, 1995.

[115] Yuri Gurevich. May 1997 draft of the ASM guide. 1997.

[116] Yuri Gurevich and James K Huggins. The railroad crossing problem: an experiment with instantaneous actions and immediate reactions. In *International Workshop on Computer Science Logic*, pages 266–290. Springer, 1995.

[117] Yuri Gurevich, Wolfram Schulte, C Campbell, and Wolfgang Grieskamp. AsmL: The Abstract State Machine Language. Technical report, Technical report, Version 2.0, Microsoft Research, Redmond, 2002.

[118] Jameleddine Hassine. Mapping ASM specifications to Spec Explorer: Guidelines, benefits and challenges. pages 380–387, 2012.

[119] Kletz T Hazop and Hazan. Identifying and assessing process industry hazards. *IchernE: Institution of Chemical Engineers,*, 1992.

[120] Miguel Angel Herrera, Aderval Severino Luna, Antonio Carlos Costa, and Elezer Monte Lemes. A structural approach to the HAZOP-Hazard and operability technique in the biopharmaceutical industry. *Journal of Loss Prevention in the Process Industries*, 35:1–11, 2015.

[121] CAR Hoare. *Communicating Sequential Processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[122] Gerard J Holzmann. *Design and Validation of Computer Protocols*. Prentice Hall, 1991.

[123] Gerard J. Holzmann. The model checker SPIN. *IEEE Transactions on software engineering*, 23(5):279–295, 1997.

[124] Alfred Horn. On sentences which are true of direct unions of algebras. *The Journal of Symbolic Logic*, 16(1):14–21, 1951.

[125] Giles Howard, Michael Butler, John Colley, and Vladimiro Sassone. A methodology for assuring the safety and security of critical infrastructure based on stpa and event-b. *International Journal of Critical Computer-Based Systems*, pages 56–75, 2019.

[126] Takuto Ishimatsu, Nancy Leveson, John Thomas, Masa Katahira, Yuko Miyamoto, and Haruka Nakao. Modeling and hazard analysis using STPA. 2010.

[127] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. MIT press, 2006.

[128] Florent Jacquemard. Security protocols open repository, 2003.

[129] Gizela Jakubowska, Piotr Dembiński, Wojciech Penczek, and Maciej Szreter. Simulation of security protocols based on scenarios of attacks. *Fundamenta Informaticae*, 93(1-3):185–203, 2009.

[130] Gizela Jakubowska, Wojciech Penczek, and Marian Srebrny. Verifying security protocols with timestamps via translation to timed automata. In *Proc. of the International Workshop on Concurrency, Specification and Programming (CS&P'05)*, pages 100–115, 2005.

[131] Audun Joesang. Security protocol verification using SPIN. In *First SPIN Workshop*, 1995.

[132] Clark John and Jacob Jeremy. A survey of authentication protocol literature. Technical Report 1.0, University of York, 1997.

[133] Mary Johnson. Using Process FMEA in an Aeronautical Engineering Technology Capstone Course. In *American Society for Engineering Education*. American Society for Engineering Education, 2010.

[134] Amarendra K. and Rao A. Vasudeva. Safety critical systems analysis. *Global Journal of Computer Science and Technology*, 2011.

[135] Angelica M. Kappel. Executable specifications based on Dynamic Algebras. In *Logic Programming and Automated Reasoning*, pages 229–240. Springer, 1993.

[136] Martin Kardos. An approach to model checking AsmL specifications. In *Abstract State Machines*, pages 289–304, 2005.

[137] Jonathan Katz, Alfred J Menezes, Paul C Van Oorschot, and Scott A Vanstone. *Handbook of applied cryptography*. CRC press, 1996.

[138] Axel Kehne, Jürgen Schönwälder, and Horst Langendörfer. A nonce-based protocol for multiple authentications. *ACM SIGOPS Operating Systems Review*, 26(4):84–89, 1992.

[139] Ralf Küsters and Tomasz Truderung. Using ProVerif to analyze protocols with Diffie-Hellman exponentiation. In *2009 22nd IEEE Computer Security Foundations Symposium*, pages 157–171. IEEE, 2009.

[140] Ralf Küsters and Tomasz Truderung. Reducing protocol analysis with XOR to the XOR-free case in the horn theory based approach. *Journal of Automated Reasoning*, 46(3-4):325–352, 2011.

[141] Pascal Lafourcade and Maxime Puys. Performance evaluations of cryptographic protocols verification tools dealing with algebraic properties. In *International Symposium on Foundations and Practice of Security*, pages 137–155. Springer, 2015.

[142] Leslie Lamport and Fred B Schneider. Formal foundation for specification and verification. In *Distributed Systems: Methods and Tools for Specification, An Advanced Course*, pages 203–285. Springer, 1985.

[143] Kim G Larsen, Paul Pettersson, and Wang Yi. UPPAAL in a nutshell. *International journal on software tools for technology transfer*, 1(1-2):134–152, 1997.

[144] Paul Leach, Chris Newman, and A. Melnikov. Using Digest Authentication as a SASL Mechanism. RFC 2831, 2000.

[145] Nancy Leveson. *Safeware: System safety and computers*. Addison-Wesley, 1996.

[146] Nancy Leveson. A new accident model for engineering safer systems. In *Proceedings of the 20th International System Safety Conference*, pages 476–486. International Systems Safety Society Unionville, USA, 2002.

[147] Nancy Leveson. A new accident model for engineering safer systems. *Safety science*, 42(4):237–270, 2004.

[148] Nancy Leveson. Software challenges in achieving space safety. *Journal of the British Interplanetary Society*, 2009.

[149] Nancy Leveson. *Engineering a safer world: Systems thinking applied to safety*. MIT press, 2011.

[150] Nancy Leveson, Cody Harrison Fleming, Melissa Spencer, John Thomas, and Chris Wilkinson. Safety assessment of complex, software-intensive systems. *SAE International Journal of Aerospace*, 5(2012-01-2134):233–244, 2012.

[151] Nancy. Leveson and Peter R. Harvey. Analyzing software safety. *IEEE Transactions on Software Engineering*, (5):569–579, 1983.

[152] Nancy Leveson and John Thomas. An STPA primer. *Cambridge, MA*, 2013.

[153] Nancy Leveson and Clark S Turner. An investigation of the Therac-25 accidents. *Computer*, 26(7):18–41, 1993.

[154] Nancy Leveson, Chris Wilkinson, Cody Fleming, John Thomas, and Ian Tracy. A Comparison of STPA and the ARP 4761 Safety Assessment Process. Technical report, MIT PSAS Technical report, 2014.

[155] Alessio Lomuscio, Charles Pecheur, and Franco Raimondi. Automatic verification of knowledge and time with NuSMV. In *Proceedings of the Twentieth International Joint Conference on Artificial Intelligence*, pages 1384–1389. IJCAI/AAAI Press, 2007.

[156] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information processing letters*, 56(3):131–132, 1995.

[157] Gavin Lowe. Some new attacks upon security protocols. In *Proceedings 9th IEEE Computer Security Foundations Workshop*, pages 162–169. IEEE, 1996.

[158] Gavin Lowe. A family of attacks upon authentication protocols. Technical report, 1997.

[159] Gavin Lowe. Casper: A compiler for the analysis of security protocols. *Journal of computer security*, 6(1-2):53–84, 1998.

[160] Robyn R Lutz. Analyzing software requirements errors in safety-critical, embedded systems. In *[1993] Proceedings of the IEEE International Symposium on Requirements Engineering*, pages 126–133. IEEE, 1993.

[161] George Zi Sheng Ma. Model checking support for CoreASM: Model checking distributed abstract state machines using Spin. Master's thesis, School of Computing Science-Simon Fraser University, 2007.

[162] Paolo Maggi and Riccardo Sisto. Using SPIN to verify security properties of cryptographic protocols. In *International SPIN Workshop on Model Checking of Software*, pages 187–204. Springer, 2002.

[163] Wenbo Mao. An augmentation of BAN-like logics. In *Proceedings The Eighth IEEE Computer Security Foundations Workshop*, pages 44–56. IEEE, 1995.

[164] Wenbo Mao and Colin Boyd. Towards formal analysis of security protocols. In *[1993] Proceedings Computer Security Foundations Workshop VI*, pages 147–158. IEEE, 1993.

[165] J. Massey and J. Omura. A new multiplicative algorithm over finite fields and its applicability in public key cryptography. *EUROCRYPT83 Udine, Italy*, 1983.

[166] James L Massey. An introduction to contemporary cryptology. *Proceedings of the IEEE*, 76(5):533–549, 1988.

[167] Kenneth L McMillan. Symbolic model checking. In *Symbolic Model Checking*, pages 25–60. Springer, 1993.

[168] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The TAMARIN prover for the symbolic analysis of security protocols. In *25th International Conference on Computer Aided Verification*, volume 8044, pages 696–701. Springer, 2013.

[169] A. Melnikov and K. Zeilenga. Simple Authentication and Security Layer (SASL). RFC 4422, 2006.

[170] Ralph C Merkle. A fast software one-way hash function. *Journal of Cryptology*, 3(1):43–58, 1990.

[171] Robin Milner, Mads Tofte, Robert Harper, and David MacQueen. *The definition of standard ML: revised*. MIT press, 1997.

[172] Matthew W Moskewicz, Conor F Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[173] John Myers. Simple Authentication and Security Layer (SASL). RFC 2222, 1997. `http://www.rfc-editor.org/rfc/rfc2222.txt`.

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

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

[176] BBC News. Boeing 737 Max: What went wrong? `https://www.bbc.co.uk/news/world-africa-47553174`. Accessed: 2019-07-27.

[177] Oracle. Writing applications that use SASL. In *Developer's Guide to Oracle Solaris® 11 Security*, chapter 7, pages 126–148. Oracle, 2012.

[178] Oracle. Java SASL API Programming and Deployment Guide. In *Java Platform, Standard Edition Security Developers Guide*, chapter 10, pages 21–28. Oracle, 2016.

[179] Martin Ouimet. *A formal framework for specification-based embedded real-time system engineering*. PhD thesis, Massachusetts Institute of Technology, 2008.

[180] Martin Ouimet, Guillaume Berteau, and Kristina Lundqvist. Modeling an electronic throttle controller using the timed abstract state machine language and toolset. In *International Conference on Model Driven Engineering Languages and Systems*, pages 32–41. Springer, 2006.

[181] Martin Ouimet and Kristina Lundqvist. The TASM Language Reference Manual, Version 1.1. *Cambridge, MA*, 2139, 2006.

[182] Martin Ouimet and Kristina Lundqvist. Automated Verification of Completeness and Consistency of Abstract State Machine Specifications using a SAT Solver. *Electr. Notes Theor. Comput. Sci.*, 190(2):85–97, 2007.

[183] Martin Ouimet and Kristina Lundqvist. The TASM toolset: Specification, simulation, and formal verification of real-time systems. In *International Conference on Computer Aided Verification*, pages 126–130. Springer, 2007.

[184] Martin Ouimet, Kristina Lundqvist, and Mikael Nolin. The timed abstract state machine language: An executable specification language for reactive real-time systems. *RTNS07*, page 15, 2007.

[185] Brandon Owens, Margaret Stringfellow Herring, Nicolas Dulac, Nancy Leveson, Michel Ingham, and Kathryn Anne Weiss. Application of a safety-driven design methodology to an outer planet exploration mission. In *2008 IEEE aerospace conference*, pages 1–24. IEEE, 2008.

[186] Vedat Özyazgan. FMEA analysis and implementation in a textile factory producing woven fabric. *Journal of Textile & Apparel/Tekstil ve Konfeksiyon*, 24(3), 2014.

[187] Tulika Pandey and Saurabh Srivastava. Comparative Analysis of Formal Specification Languages Z, VDM and B. *International Journal of Current Engineering and Technology*, 5(3):2086–2091, 2015.

[188] M Panti, L Spalazzi, and S Tacconi. Attacks on cryptographic protocols: A survey. Technical report, Istituto di Informatica, University of Ancona, 2002.

[189] M Panti, L Spalazzi, and S Tacconi. Using the NuSMV model checker to verify the Kerberos protocol. In *The proceedings of the International Conference on Simulation and Multimedia in Engineering Education*, volume 34, pages 230–236. Society for Modeling and Simulation International, 2002.

[190] James L Peterson. *Petri net theory and the modeling of systems*. Prentice-Hall, 1981.

[191] Simon L Peyton Jones and Philip Wadler. Imperative functional programming. In *Proceedings of the 20th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 71–84. ACM, 1993.

[192] Ludovic Pièтre-Cambacédès and Claude Chaudet. The SEMA referential framework: Avoiding ambiguities in the terms security and safety. *International Journal of Critical Infrastructure Protection*, 3(2):55–66, 2010.

[193] Amir Pnueli. The temporal logic of programs. In *18th Annual Symposium on Foundations of Computer Science (sfcs 1977)*, pages 46–57. IEEE, 1977.

[194] Stephen Pohlig and Martin Hellman. An improved algorithm for computing logarithms over gf (p) and its cryptographic significance (corresp.). *IEEE Transactions on information Theory*, 24(1):106–110, 1978.

[195] DR Prajapati. Application of FMEA in Automobile Industries: A Case Study. *IUP Journal of Mechanical Engineering*, 4(4), 2011.

[196] David John Pumfrey. *The principled design of computer system safety analyses*. PhD thesis, University of York, 1999.

[197] Joh Damon Reese and Nancy Leveson. Software deviation analysis: A safeware technique. In *AIChe 31st Annual Loss Prevention Symposium*. Citeseer, 1997.

[198] Donald J Reifer. Software failure modes and effects analysis. *IEEE Transactions on reliability*, 28(3):247–249, 1979.

[199] Sigitas Rimkevičius, Mindaugas Vaišnoras, Egidijus Babilas, and Eugenijus Ušpuras. HAZOP application for the nuclear power plants decommissioning projects. *Annals of Nuclear Energy*, 94:461–471, 2016.

[200] A William Roscoe. Modelling and verifying key-exchange protocols using CSP and FDR. In *Proceedings The Eighth IEEE Computer Security Foundations Workshop*, pages 98–107. IEEE, 1995.

[201] Bill Roscoe. Model-checking CSP. In *In A Classical Mind, Essays in Honour of CAR Hoare.* Prentice-Hall, 1994.

[202] Dean Rosenzweig, Davor Runje, and Neva Slani. Privacy, abstract encryption and protocols: an ASM model-part I. In *Abstract State Machines 2003*, pages 372–390. Springer, 2003.

[203] D. Russell, K. Blacklock, and M. Langhenry. Failure Modes and Effects Analysis (FMEA) for the Space Shuttle Solid Rocket Motor. In *AIAA/ASME/SAE/ASEE 24 th Joint Propulsion Conference*, 1988.

[204] Israel Barragan Santiago and Jean-Marc Faure. From Fault Tree Analysis to Model Checking of Logic Controllers. *IFAC Proceedings Volumes*, 38(1):86–91, 2005.

[205] Mahadev Satyanarayanan. Integrating security in a large distributed system. *ACM Transactions on Computer Systems (TOCS)*, 7(3):247–280, 1989.

[206] Bryan Scattergood. *The semantics and implementation of machine-readable CSP*. PhD thesis, Citeseer, 1998.

[207] Bastian Schlich and Stefan Kowalewski. [MC] SQUARE: A model checker for microcontroller code. In *Second International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*, pages 466–473. IEEE, 2006.

[208] Joachim Schmid. Introduction to AsmGofer. *Archived web pages at: https://tydo.eu/files/AsmGofer/AsmGoferIntro.pdf*, 2001.

[209] Douglas C Schmidt. Model-driven engineering. *IEEE Computer Society*, 39(2):25–31, 2006.

[210] Eric Schmitt. Army is blaming Patriots computer for failure to stop Dhahran scud. *New York Times*, 20, 1991.

[211] Bruce Schneier. Attack trees. *Dr. Dobbs journal*, 24(12):21–29, 1999.

[212] Bruce Schneier. *Applied cryptography: Protocols, algorithms, and source code in C*. john wiley & sons, 2007.

[213] Bruce Schneier. *Secrets and lies: Digital security in a networked world*. John Wiley & Sons, 2011.

[214] Adi Shamir, Ronald L Rivest, and Leonard M Adleman. Mental poker. Technical Report TM-125, MIT Laboratry for Computer Science, 1978.

[215] R. Siemborski and A. Gulbrandsen. IMAP Extension for Simple Authentication and Security Layer (SASL) Initial Client Response. RFC 4959, 2007.

[216] R. Siemborski and A. Melnikov. SMTP Service Extension for Authentication Initial Client Response. RFC 4954, 2007.

[217] Michael Sipser. *Introduction to the Theory of Computation*, volume 2. Thomson Course Technology Boston, 2006.

[218] Anatol Slissenko and Pavel Vasilyev. Simulation of timed Abstract State Machines with predicate logic model-checking. *J. Universal Computer Science*, 14(12):1984–2006, 2008.

[219] Ian Sommerville. Insulin Pump – Z schemas. `http://iansommerville.com/software-engineering-book/files/2014/07/Insulin-Pump-Z-schemas.pdf`. Accessed: 2019-05-21.

[220] Ian Sommerville. *Software Engineering*. Addison Wesley, 9th edition, 2010.

[221] J. Michael Spivey. *The Z notation: A reference manual*. Prentice Hall International Series in Computer Science, 2nd edition, 1992.

[222] William Stallings. *Cryptography and network security: principles and practice*. Pearson Education, 7th edition, 2017.

[223] R. F. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine: Definition, Verification and Validation*. Springer-Verlag, 2001.

[224] Paul Syverson. A taxonomy of replay attacks. In *Computer Security Foundations Workshop VII (CSFW'94)*, pages 187–191. IEEE, 1994.

[225] Clemens Szyperski, Dominik Gruntz, and Stephan Murer. *Component software: Beyond Object-Oriented Programming*. Pearson Education, 2002.

[226] John Thomas. *Extending and automating a systems-theoretic hazard analysis for requirements generation and analysis*. PhD thesis, Massachusetts Institute of Technology, 2013.

[227] John Thomas and Nancy Leveson. Performing hazard analysis on complex, software- and human-intensive systems. In *29th International System Safety Conference*. International System Safety Society Unionville, VA, 2011.

[228] Simon Thompson. *Haskell: the craft of functional programming*. Addison-Wesley, 2nd edition, 1999.

[229] Elena Troubitsyna. Elicitation and Specification of Safety Requirements. In *Systems, 2008. ICONS 08. Third International Conference on*, pages 202–207. IEEE, 2008.

[230] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with Spec Explorer. In *Formal methods and testing*, pages 39–76. Springer, 2008.

[231] William E Vesely, Francine F Goldberg, Norman H Roberts, and David F Haasl. Fault Tree Handbook. Technical report, Nuclear Regulatory Commission Washington DC, 1981.

[232] Andrija Volkanovski, Marko Čepin, and Borut Mavko. Application of the fault tree analysis for assessment of power system reliability. *Reliability Engineering & System Safety*, 94(6):1116–1127, 2009.

[233] Ton Vullinghs, Wolfram Schulte, and Thilo Schwinn. An introduction to TkGofer. Technical report, Fakultät für Informatik, Universität Ulm, 1996.

[234] Jichuan Wang, Shaoying Liu, Yong Qi, and Di Hou. Developing an insulin pump system using the SOFL method. In *Software Engineering Conference, 2007. APSEC 2007. 14th Asia-Pacific*, pages 334–341. IEEE, 2007.

[235] EP Wentworth. Introduction to Funcional Programming using Gofer. Technical report, Department of Computer Science, Rhodes University, 1994.

[236] K Preston White and Ricki G Ingalls. Introduction to simulation. In *Winter Simulation Conference (WSC)*, pages 1741–1755. IEEE, 20o9.

[237] Michael Willett. Cryptography old and new. *Computers & Security*, 1(2):177–186, 1982.

[238] Thomas YC Woo and Simon S Lam. A semantic model for authentication protocols. In *Research in Security and Privacy, 1993. Proceedings., 1993 IEEE Computer Society Symposium on*, pages 178–194. IEEE, 1993.

[239] Chong Xu, Gershon Kedem, and Fengmin Gong. Categorizing attacks on cryptographic protocols based on intruders' objectives and roles. In *Workshop on Formal Methods and Computer Security*, 2000.

[240] Rui Xue and Deng-Guo Feng. New semantic model for authentication protocols in ASMs. *Journal of Computer Science and Technology*, 19(4):555–563, 2004.

[241] Alexey V Yablokov, Vassily B Nesterenko, Alexey V Nesterenko, and Janette D Sherman-Nevinger. *Chernobyl: Consequences of the catastrophe for people and the environment*, volume 39. John Wiley & Sons, 2010.

[242] W Young and N Leveson. Inside risks-an integrated approach to safety and security based on system theory: Applying a more powerful new safety methodology to security risks. *Communications of the ACM*, 57(2):232–242, 2014.

[243] K. Zeilenga. The PLAIN Simple Authentication and Security Layer (SASL) Mechanism. RFC 4616, 2006.

[244] Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler backends: An ASM-approach. *Journal of Universal Computer Science*, 3(5):504–567, 1997.