MODELLING A MULTICHANNEL SECURITY PROTOCOL TO ADDRESS MAN IN THE MIDDLE ATTACKS

by

Aliaa Mahfooz Ali Alabdali - 091594509

Submitted for the degree of
Doctor of Philosophy

Heriot-Watt University

Department of Computer Science
School of Mathematical and Computer Sciences

November 2017

Abstract

Unlike wired networks, wireless networks cannot be physically protected, making them greatly at risk. This study looks into advanced ways of implementing security techniques in wireless networks. It proposes using model checking and theorem proving to prove and validate a security protocol of data transmission over multi-channel in Wireless Local Area Networks (WLANs) between two sources. This can help to reduce the risk of wireless networks being vulnerable to Man in the Middle (MitM) attacks. We model secure transmission over a two-host two-channel wireless network and consider the transmission in the presence of a MitM attack. The main goal of adding an extra channel to the main channel is to provide security by stopping MitM from getting any readable data once one of these channels has been attacked.

We analyse the model for vulnerabilities and specify assertions for secure data transmission over a multi-channel WLAN. Our approach uses the model analyser Alloy which uses a Satisfiability (SAT) solver to find a model of a Boolean formula. Alloy characterizations of security models are written to analyse and verify that the implementation of a system is correct and achieves security relative to assertions about the model of our security protocol. Further, we use the Z3 theorem prover to check satisfiability using the Satisfiability Modulo Theories (SMT) solver to generate results. Using Z3 does not involve high costs and can help with providing reliable results that are accurate and practical for complex designs.

We conclude that, based on the results we achieved from analysing our protocol using Alloy and Z3 SMT solver, the solvers complement each other in their strengths and weaknesses. The common weakness is that neither can tell us why the model is inconsistent, if it is inconsistent. We suggest that an approach of beginning with modelling a problem using Alloy and then turning to prove it using Z3, increases overall confidence in a model.

Acknowledgements

First of all, praise and thanks be to Allah, the Almighty, for helping me to complete my PhD studies.

I want to show my appreciation and gratitude to my supervisors, Dr Lilia Georgieva and Prof. Greg Michaelson, for their encouragement and patience while helping me throughout the creation of this thesis. I am grateful for their helpful advice and guidance.

I would also like to thank my husband Adel Almutairi who has held my hand throughout my entire studies.

Last but not least, I would like to thank my family: my parents, my sisters and my brothers for supporting me throughout my life and during the years that led to this achievement.

Big and special thank to my all lovely children. I want to dedicate this work to them and tell them how great they are and how grateful I am for their sacrifice and patience during the years I have been away from them. Your impact on me has been greater than words can express. I love you all xxx.

I am truly grateful..

ACADEMIC REGISTRY
Research Thesis Submission

Name*:*	Aliaa Mahfooz Ali Alabdali		
School:	Mathematical and Computer Sciences		
Version: *(i.e. First, Resubmission, Final)*	Final	Degree Sought:	PhD

Declaration

In accordance with the appropriate regulations I hereby submit my thesis and I declare that:

1) the thesis embodies the results of my own work and has been composed by myself
2) where appropriate, I have made acknowledgement of the work of others and have made reference to work carried out in collaboration with other persons
3) the thesis is the correct version of the thesis for submission and is the same version as any electronic versions submitted*.
4) my thesis for the award referred to, deposited in the Heriot-Watt University Library, should be made available for loan or photocopying and be available via the Institutional Repository, subject to such conditions as the Librarian may require
5) I understand that as a student of the University I am required to abide by the Regulations of the University and to conform to its discipline.
6) I confirm that the thesis has been verified against plagiarism via an approved plagiarism detection application e.g. Turnitin.

* *Please note that it is the responsibility of the candidate to ensure that the correct version of the thesis is submitted.*

Signature of Candidate*:*	[illegible]	Date:	11/07/2017

Submission

Submitted By *(name in capitals):*	ALIAA MAHFOOZ ALI ALABDALI
Signature of Individual Submitting:	[illegible]
Date Submitted:	11/07/2017

For Completion in the Student Service Centre (SSC)

Received in the SSC by *(name in capitals):*			
Method of Submission *(Handed in to SSC; posted through internal/external mail):*			
E-thesis Submitted (***mandatory for final theses)***			
Signature:		Date:	

Please note this form should be bound into the submitted thesis.
Academic Registry/Version (1) August 2016

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

Software systems have become increasingly important in our daily lives, which increases the demand for ensuring correctness of their behaviour. This applies not just to safety critical domains such as medical and traffic systems; software failures also have a detrimental effect on the economy as a whole.

The size and the complexity of software systems have grown in recent years [105]. Recently, developers have become able to achieve the goal of building systems that implement reliably in spite of their complexity, by using methods that are required to provide an accurate specification of the system and its validity properties. To achieve this goal, *formal methods*, which are mathematically based languages, techniques, and tools for specifying and verifying a system are utilized [89]. Formal methods, when utilized in the design stages of a system's development, provide more accurate results at a lower cost [97] than traditional techniques.

The formal method approach has gained more and more acceptance in industry over the last few years and this is shown clearly, for instance, by the last standard of the European Committee Standardization for railway control and protection systems, where formal methods are "highly recommended" for the safety integration [3].

Moreover, a formal approach is strongly affected as well by the nature and expressive power of the specification language. Its underlying logic is that the more the required behaviour, which is naturally on a higher abstraction level, can be expressed directly, the more specification errors can be avoided, and the properties can be captured and verified.

Formal methods are required to check the behaviour of software. Formal semantics is used for the formulation of the expected behaviour as well as for the description of the software system itself; a framework and a methodology are used for gaining confidence about software correctness, which can range from random checks to a complete proof of correctness.

This thesis uses formal methods to transform software system together with the required behaviour into a logical formula proof in a formal language, such that the formula is valid if the software properties satisfy the required behaviour. The approach in which the software system is checked against a finite number of tests has an ability to give (mathematical) proofs of correctness and is thus the method of choice for safety critical systems.

In this thesis, we explore the use of typed first-order logic (FOL) to model and analyse a multichannel wireless protocol using the model analyser Alloy. Further we use the Z3 SMT solve to prove satisfiability of the properties of the multichannel wireless protocol. We provide two complementary approaches in our case study which uses Alloy and Z3 and validate the proposed methodology of our multichannel security protocol.

Alloy provides a suitable set of methods to easily and directly describe software systems together with their desired behaviours. Alloy descriptions of software systems are called *models*, while Alloy descriptions of required behaviours are called *assertions*. In the verification context, we refer to an Alloy model together with an Alloy assertion as an Alloy specification.

The main significant reason for the success and popularity of Alloy is the fully automatic *Alloy Analyser*. It checks Alloy specifications by looking for a counterexample. The models analysed by Alloy are small. Their default scope is three [83]. This gives the user the ability to check the specification of a model in small scope to detect any errors simply before developing the model to be complex with increasing the number of scopes gradually. However, checking is implemented with respect to a bounded scope in which only a small number of values is considered for each type. Thus, despite the Alloy Analyser's ability to find counterexamples effectively, it has no ability to establish the validity of the proof. Thus we need a complementary formal method such, as an SMT solver, choosing Z3.

According to [22] Z3 is a state-of-the art theorem prover from Microsoft Research. It can be used to verify the satisfiability is logical formulas automatically over more than one theory. Z3 provides a good match for verification components and software analysis since many similar constructs of software map directly into its supported theories. Z3 is used efficiently for logical solving and modelling.

1.2 Aims And Objectives

The principal goal is to compare the techniques of Alloy and Z3 for specifying properties of wireless network protocols through the analysis of Man in the Middle Attacks over a single channel and then an alternative multichannel protocol. Considering this main aim of the study, the objectives are as follows:

- Propose and build a modelling and verification approach using typed first-order logic as a specification language for data transmission over a WLAN.
- Find a model for a given principal goal using the Alloy Analyser [92]. Alloy allows us to specify the flow of communication, characterise the participants in the communication, formulate assertions which would guarantee security, and check the validity and security of the protocol.
- Prove the modelling and verification properties utilising a theorem prover. For the purpose of this study, Z3 will be used to find a proof for a given principal goal using SMT to generate its results.
- Explore out how model analysis compares with theorem proving.
- Make a detailed comparison of both Alloy and Z3 including this relative strength and limitations.

1.3 Thesis Structure

- Chapter 1: The Introduction

This chapter provides the context for our case study. It also describes the main objectives that led to this work being conducted in the Wireless Networks field, using the Alloy and Z3.

- Chapter 2: General Background and Related Work in Multichannel Wireless Security Protocol and Formal Methods

In this chapter, we discuss studies of using multichannel wireless networks for transmitting data securely, the strength, issues and applicability of Alloy for improving models, and the accurate use of Z3 to prove the satisfiabilities of theories. In addition, the chapter gives research results on the strength, efficiency, and the accuracy of results when using formal methods.

- Chapter 3: Foundations: Alloy

This first foundation chapter provides a general introduction to the Alloy specification language and its analysis using the Alloy Analyser to model an ATM system as an example.

- Chapter 4: Foundations: Z3 SMT Solver

This second foundation chapter provides a general introduction to the Z3 SMT solver using the same ATM system as an example. At the end of chapter 4, a comparison between SAT and SMT results, and SAT and SMT tools is provided. These comparisons include decidability, time, limitations, accuracy, counterexample, capabilities, and quantifiers.

- Chapter 5: Problem Specification and Case Study, and Multichannel Security Protocol Modelling and Analysis

This chapter considers work related to the case study in the area of securing data transmission over a multichannel. It also looks into the details of the particular issues that have been chosen for this study. Further, this chapter discusses the reason behind presenting the model for our case and how it can help with achieving secure communication over wireless networks. This chapter also discusses how data being transmitted over wireless networks can be secured and safeguarded from MitM attacks by making use of a multichannel instead of a single channel for data transfer. The security of a multichannel protocol depends on analysing the transmitted message into two parts, random letters and the indices for each letter, and then transmitting each part over different wireless networks, and changing the MAC address for each channel. This chapter finally gives a description of the proposed model that has been developed.

- Chapter 6: Multichannel Security Protocol Modelling Using Alloy

This chapter presents the Alloy framework for modelling and checking the validity of properties of the provided protocols using: (i) predicate-running which is applied to Alloy problems with a predicate and results in a visualization if the model is consistence, and (ii) assertion-checking, which is applied to Alloy specifications in which properties have been constrained as facts.

- Chapter 7: Multichannel Security Protocol Proving Using Z3

In this chapter, we present the results that we found after proving the satisfiability of the properties of the protocols using Z3. Then we compare and discuss the results, and how Alloy and Z3 complement each other in terms of strengths and weaknesses and whether working together strengthens the capacity of them in terms of automation and scalability. We also explore whether both model checking and theorem proving could provide the validity and the satisfiability of the security of our protocol for the same property. In chapter 7, we noticed that, after making the corresponding transformation of model from Alloy into Z3, when checking the assertion results a counterexample in Alloy, it also results in a counterexample in Z3 under the same properties. Also, when checking the assertion results in no counterexample in Alloy, it also results in the same in Z3 under the same constraints on properties .

- Chapter 8: Systematic Translation Rules A First Step Towards An Automated Translator

In chapter 8 we show how small and simple model in Alloy maybe translated manually into a large and complex (less readable) formulas in Z3 including declaring types, subtypes, abstraction, extension, multiplicity constraints, relations, facts, assertions, formulas, and analysis. This chapter presents and discusses abstract syntax for the core Alloy logic and SMT2 language; the tool integration and methodology that we have worked on with the Alloy Z3 hybrid for verification; and systematic translation rules to describe how each piece of Alloy syntax is translated into Z3.

In this chapter we show how to translate a specification that is written in the Alloy specification language into a satisfiability-equivalent SMT problem using Z3 SMT logic (FOL) and solved by an SMT solver such that if there is a counterexample in Alloy in finite scopes, it maybe supposed to be a counterexample in Z3 in infinite scopes and vice versa.

- Chapter 9: Conclusion and Future Work

In this final chapter we present conclusions of our approach and future work.

1.4 Contributions

Our contributions in this thesis include:

- A model of a multichannel protocol built using Alloy.
- A model of the protocol using Z3 including all of the aspects that were included in Alloy.
- A comparative analysis of the the results found through Alloy and Z3.
- Identifying and comparing the strengths, limitations, and advantages of using theorem proving and model analysis.

Chapter 2

General Background and Related Work for Security Protocol in Multichannel Wireless and Formal Methods

2.1 Context

Within the last ten years, the software engineering industry has seen globally a growing amount of susceptible information being produced, used, and accessed by web-based applications using wide area networks and wireless technology from financial institutions, healthcare organizations, online payment agencies among other transactions conducted online [16]. Transactions conducted online and personal security now depend on the safety and protection that are incorporated into web-based software programs. It is important to note that even one design flaw or execution error in a program can be used by malicious criminals to embezzle, change, or falsify the confidential information of guiltless system end-users. Such network and web application attacks are now a common problem and are having a negative impact on a number of organizations and system users [16].

The dangers of installing software with security issues are too big to accept with no assurance, but programs are getting extremely big for security engineers and professional experts to scrutinize manually, and computerized verification and validation programs are not able to scale. At the moment, we don't have any single extensively used online program or network that can be certified as safe and with a high level of security, even alongside a small group of hacks. In fact, development and execution faults are still established in commonly disseminated and comprehensively examined protection systems and network operating systems developed by excellent professionals and executed by specialists [13].

2.1.1 Wireless Protocol

Wireless is a technology that facilitates networking by linking many computer users to share resources in a home or business environment simultaneously over a single channel with no need for any additional wiring or cabling. Wireless networks have become pervasive. They allow millions of users across the globe to exchange information, conveniently and flexibly, among a variety of devices including notebook computers, PDAs, tablet PCs, mobile phones, digital cameras, and hand-held games. The intensive use of wireless networks over single-channel wireless networks reveal various vulnerabilities: the wireless channel can be eavesdropped; the data transmitted over the network can be altered; the identities of the communicated parties can be impersonated; and the communication channel can be overused or jammed [27, 70, 136, 68].

The rapid commercialisation of new products and services that use wireless networks does not allow enough time for the design of robust security architectures [132]. Consequently, the risk of losing and misusing data transmitted over a wireless network is high [2, 31]. As a result, an investigation to ensure information security on a wireless network by using a strong key or transmitting data over pairs of nodes and encryption has been identified as a critical need. [124].

In the context of wireless networks, preserving information security has three main goals [132]:

- *Confidentiality*: ensuring that nobody but the receiver can see the information.
- *Integrity*: ensuring that the information has not been modified.
- *Availability*: ensuring that the information is available whenever it is needed.

One type of threat to a single channel wireless networks is known as Man in the Middle (MitM). In MitM over a single channel, the attacker takes advantage of weak network communication protocols in order to convince a host to route the information through the attacker instead of through the normal router. The attacker intercepts messages in a public key exchange and then resends messages after substituting their own public key for the requested public key [133]. Hence, the attacker makes two hosts believe that they are communicating with one another, while the attacker controls and modifies the communicated messages.

The attacker attacks in real time by splitting the original Transmission Control Protocol (TCP) connection into two new connections and acting as a proxy where they can read, insert, and modify the data in the intercepted communication [120].

Our work focuses on modelling a multichannel wireless network communication protocol using two formal methods Alloy and Z3. Our challenge is to compare the technique of Alloy and Z3 for specifying properties of wireless network protocols through the analysis of Man in the Middle Attacks over a single channel and then a multichannel alternative protocol.

2.1.2 Security Protocols

Today, active information and communicating like e-commerce, mobile technologies, and the Internet, have made the use of security protocols an absolute necessity. It is therefore essential for systems designers to have confidence in the security protocols they use [97].

The main purpose of security protocols is securing communications on insecure networks, such as the Internet. Security protocols are required to be used everywhere, such as in electronic commerce (e.g., the protocol TLS [46], used for https:// URLs), bank transactions, mobile phone, and wireless networks. However, the design of security protocols is error-prone; this was clearly demonstrated by the famous Needham-Schroeder public-key protocol [118], when the design flaw of its security protocol was detected 17 years after its publication [103]. Thus, security flaws cannot necessarily be discovered using functional software testing because they appear just in the presence of a malicious attack. For this reason automatic tools such as formal methods are required for ensuring that security protocols are correct during the earlier stages of development. Research in the verification of security protocols has been very active since the 1990s, and is still very active [109].

2.1.3 Multichannel Protocols

The concept of using multichannel protocols is not new. In cryptography, the initial idea of using multichannel protocols was to use two channels: one providing more security during sending and receiving the symmetric key, and the other sending the ciphertext. Transmitting data over multichannel has been used to transmit Pretty Good Privacy (PGP) fingerprints, which are attested through authentic channels with another channel that is used to send and receive a PGP public key [156].

Today there are many viewpoints about research directed towards multichannel because it helps with avoiding protocol design defects and it is the main approach to resolving security requirements, channel properties, and the attacker model [156]. Keeping a multichannel communication secure is essential and is of particular interest for pervasive computing [156]. The multichannel data communication concept aims to make multiple-paths appear as a strong single path, secure and authoritative communication link between at least two parties.

A multichannel protocol facilitates data transmission securely and effectively in WLANs, especially in the presence of a Man in the Middle [156]. However, using software to build such a security protocol increases the need to combine verification techniques such as model checking and theorem proving.

2.2 Background in Security Protocols to Transmit Data Securely Over Multichannel Wireless Networks

2.2.1 Introduction

Data confidentiality over mobile devices is not easy to secure due to a lack of strong encryption components. However, new devices have multi wireless interfaces with a variety of channel capacities and security components [158]. Data confidentiality is the main priority of security solutions for mobile environments and wireless networks are in particular amenable to threats such as eavesdropping [158].

As it is known [158], the traditional solution for protection is through data encryption. However, many link layer encryption schemes are weak, such as WEP [45] in WiFi home or small business networks. Even strong end-to-end encryption techniques such as TLS [47] are not considered an adequate protection solution because many web sites do not support them, and they might be too difficult for deployment and too expensive for maintenance [158].

The research below in wireless security addressed the vulnerabilities in WLANs as follow.

2.2.2 Security Protocols for Data Transmission Over Multichannel Wireless Networks

Interest in multichannel security protocols is growing [156, 73]. For example, the multichannel MAC Protocol has been studied by [159]. This protocol uses a technique called Cognitive Radio (CR) to prevent Denial-of-Service (DoS) attacks by radio jamming in a wireless network. This kind of attack takes place if the device uses single channel communication, making it easy for eavesdroppers to send and receive radio signals or packets to confuse the original client transmission, which leads to DoS. The multichannel MAC protocol is one of the suggested solutions for that problem. It utilises CR which has two technologies that are responsible for obstructing the jamming attack. The first is real time spectrum sensing and the second is fast channels switching. The disadvantage is that using many channels in this protocol can cause a higher channel sensing time [159].

Multi-hop communication with access to the multichannel MAC (medium access control) protocol was used by [157] to reduce the problems in a Wireless Mobile Ad hoc Network (MANET), such as conflict and collision caused by increasing the number of mobile hosts quickly and decreasing the performance of the network. This protocol was designed for wireless networks, which assume a joint channel exchanged by mobile hosts. The protocol works by utilising static channel activity and applying IEEE 802.11 in the sending and receiving operations on every channel. This protocol faced problems with hidden terminals and exposed terminals

[157].

A hidden terminal, as defined in [157] is when there are two hosts X and Y, and host X is sending to host Y. Y's receiving activity could be destroyed when another host Z cannot sense the signals from host X which leads to Z's transmission activity being mathematically overheard by host Y. An exposed terminal, as defined in [157] is, host X is sending to Y. Later, host Z plans to send to host D, but because Z can sense X's signals, Z will wait until X's transmission activity terminates. In fact, the communications from X to Y and from Z to D can happen simultaneously.

To eliminate these problems, in the multichannel MAC protocol called dynamic channel assignment (DCA) [157], bandwidth is separated into two types of channel.

A single control channel (control transceiver) is used to solve connection problems and to allocate data channels to mobile hosts. Then a number of data channels (data transceivers) are used to submit data packets and acknowledgement through switching between data channels. This protocol has the ability to support mobile communication between two hosts and has comparable simplicity to our approach which is transmitting data over two different channels with the message separated into two parts and each part transmitted over a channel.

A multichannel technique is also used to prevent Wireless Mobile Ad hoc Networks from jamming attacks using a protocol called Multi-path Multi-Channel Protocol. The goal behind using multichannel in this protocol is to prevent jamming attacks [153], especially jamming attacks focusing on the transmitted channel itself. This protocol depends on designing the transmitted channels to start as a single channel, with the ability to add more channels if a jamming detection system detects a jamming attack.

The technique of Multi-path Multi-Channel Protocol starts in the MAC layer with 802.11 control traffic sent to each node: each node at each configurable W second calculates the average control traffic (CT) sent of neighbour nodes. If CT is greater than T, which is the total average of submitted control traffic from adjoining nodes, then an alarm packet will be generated and broadcast to warn other nodes to change their common control channel and increase the number of transmitted channels [153].

Relay attacks are studied by [149, 44, 23]. A relay attack is a kind of MitM attack. The attacker captures some of the data packets on their way to the intended destination and then attempts to re-use this information to attack the victim's network without modifying data. MitM has ability to observe, modify, and/or block data after intercepting the data, making the parties think they are connecting to each other.

According to [149], encryption alone may not protect all communication between two parties from MitM attacks. They propose multichannel protocols. One solution for preventing relay attacks is to add another channel so two endpoints can establish if they are connecting to each other securely or if there is a MitM intercepting them. The main characteristic of this extra channel is that it is difficult for it to be relayed. The technique is based on a distance bounding protocol to provide connection approval with a low force security guarantee, provided there are no other principals within a specified distance of the verifier. The main goal of the distance bounding protocol is to give only the genuine provider the ability to respond to the challenge, whereas the MitM never responds to the challenge because MitM does not have any idea about the shared secret.

Our approach is similar; also we model secure message exchange over a multichannel network that makes similar assumptions to [5]. These assumptions are that we should not depend entirely on encryption to guarantee the security, and that adding another channel to transmit a message may increase the security of the transmitted message, which in turn prevents MitM.

An analysis of a relay attack [44] based on speed, and calculating how much time sending and receiving data takes, depends on the theory of the speed of light. This defensive technique is limited and works by establishing whether the provider is further away than predicted. An analysis of a relay attack is reported by [23] but with a high security low level protocol with a pre-calculation of a one bit challenge and a response which transmits as fast as the channel allows. In contrast, we are not concerned with the speed of the message exchange. Instead we only capture the intended sender and receiver and the relevant properties of description for a secure exchange to take place.

In a multichannel key agreement, using an encrypted public key exchange, secure communication is achieved by sending and receiving messages that are easy to eavesdrop or change through public channels. To make a fresh strong cryptographic key agreement between two PDAs, Diffie-Hellman (DH) key exchange may be used to guarantee security, confidentiality, integrity, and authenticity in data transition [100, 48].

The DH key exchange [58] is a method that provides security when symmetric cryptography cannot eliminate eavesdropping. The protocol provides robust security for data transmission over two channels. Its technique works by merging long values submitted via radio channel with short values submitted via the data origin authentication channel.

The DH key agreement protocol accomplishes similar goals to those achieved by [58], but there are further goals that this protocol has achieved, such as neglecting for how long the hacker tries to hack the second channel. This protocol could decrease the ability to hack the second channel data transition to less than 1 in 1,000,000 by taking into account the use of the bandwidth of the second channel. Another goal is that the protocol requires the data transferred over the second channel to be utilized in calculating subsequent protocol values, instead of merely to be checked for equality [30]. This is another difference to [58].

A new security protocol called Multichannel Encryption Overlay (MEO) using multiple channels to transmit data securely has been introduced by [158]. This protocol is similar to ours in terms of increasing the number of options for wireless broadband. However, in this protocol the options for wireless broadband can be used simultaneously with no need to disconnect from one and reconnect to the other. Our protocol requires this disconnection and reconnection to change the MAC addresses using "Technitium MAC Address Changer" software, which strengthens the security.

The goal of the MEO protocol is to increase data confidentiality by splitting data transmission over multiple channels. This protocol is based on two ideas: first, removing any information to frustrate decryption; and second, to reduce the rate of the information, helping to increase the time available to stop those attacks which rely on sniffing encrypted text. These goals are achieved by, splitting the source into two or more channels. The majority of the data is carried by the first channel, and the rest, after some additional encryption, on the second channel. The first channel essentially carries the first part of data, and data on the second channel is low rate, and encrypted. Before reaching the final destination, the split data can be reassembled at a network proxy location [158].

The MEO protocol is similar to our protocol in terms of its technique of splitting data into two parts and transmitting each part over different channel; however, we think our protocol is stronger because it uses different channels at different times, changing the MAC address for each channel using "Technitium MAC Address

Changer" software. It also does not depend on submitting the data itself but just random letters and indices.

2.3 Specification and Verification

Verification is the process of identifying whether a system meets its specifications. In the past, the goal of this process has been to verify after a development project's completion whether the implementation of a program (what we got) is equal to the specifications (what we wanted). Now, however, verification occurs in the early stages of the software development cycle, with the goal of verifying the validity of the theoretical model. Verification here has the advantage of evaluating early on whether the product will satisfy system requirements and whether the system will implement as required. It also allows for the detection of errors before they've spread and become more complex [89].

Specifications frequently rely on semantics to understand the system to be developed. Because of the ambiguity of the natural language that is used to specify the software development, it is hard to verify the validity of the system and the verification process becomes less efficient when the software development becomes bigger and more complex. The automation of the procedure is difficult as well because of the informal specifications [89].

2.3.1 Formal Methods

Formal methods are a particular kind of mathematically based technique for the specification, development, and verification of software and hardware systems. They enable accurate verification of specifications and their realization. Moreover, they facilitate the automation of verification and refine its efficiency. Therefore, the utilization of formal methods is expected to raise the quality, reliability, and accuracy of software.

Formal methods are introduced as a method to convert a problem from the informal space into the formal space, where it is easier for computational methods and technologies to be applied. Formal methods are used to represent methods that are used in process or system engineering, and allow for the description of a problem using a method that assists in discovering the solution. Formal methods were utilized on a large scale with software engineering to first determine the goal of the system, and then to design, develop, and to validate the fundamental system [141].

Until formal methods were included in the development process, security solutions were verified via human analysts. On several occasions, some traces were not conducted in the analysis. Thus, relevant errors were not discovered. Formal methods for security protocol analysis as illustrated by [109] represent a mathematical or logical modelling of a system and properties to be proven, together with an efficient procedure for determining whether a model satisfies these properties.

There are two kinds of formal methods: formal specification and formal verification [89].

2.3.1.1 Formal Specification

Formal specifications are utilized to describe a system, to analyse its behaviour, and to help its design by verifying system properties such as functional behaviour, timing behaviour, performance characteristics, or internal structure through strict and efficient reasoning tools [72]. Most analysts and programmers have not been trained in formal specification techniques because formal specifications can take a very long time and it may be impossible to make changes to the specifications once design and implementation has begun. Also, formal specifications describe what a system should do, not how the system should do it, and maybe used to detect errors but do not help solve those errors. Moreover, they are complex because they require a high level of mathematical expertise and analytical skills to understand

and apply them efficiently [72]. Even in a relatively simple system, they might include formulas ranging over 100 pages [139].

2.3.1.2 Formal Verification

Detecting errors manually is very difficult, and the software engineering process is very complex. For those reasons, the trend is moving toward verification techniques to discover design errors.

Since the 1990s, formal verifications are increasingly helpful in proving the correctness of systems such as security and cryptographic protocols, combinational circuits, and digital circuits with internal memory [141, 51, 19].

The goal of much of the research in formal verification is designing decidable formalisms in which targeted properties of computing systems can be formulated, so that the truth and falsity of such formulas can be checked through decision procedures [89, 141].

2.3.1.3 Distinction Between Formal Specification and Formal Verification

In computer science, both formal specification and verification are examples of formal methods which use mathematical techniques to help document, specify, design, analyze, or certify software and hardware. Formal verification is concerned with mathematical models of both: a system and its requirements [143].

For a complex system both formal specification and verification can be utilized to manage its complexity. At the beginning, the system is modelled utilizing formal techniques to get a specification. And then, for this specification, verification techniques are applied to verify the correctness of the system [102].

For example the Transmission Control Protocol (TCP) is a complex system and its specification is complex. For the verification of 'the TCP protocol, [102] first developed a formal specification of TCP protocol and its extension. After that, he applied verification techniques to prove the correctness of the protocol.

Formal specification and verification techniques may be required together for different reasons: under all circumstances a formal specification serves as an unambiguous reference for the system's behaviour because of its formal nature. The reliability of a system can be improved when using verification to detect its errors; a process that requires detailed knowledge of both the proof system that is utilized and the axioms that are applied may be helpful through the development of a system [102].

2.4 Formal Verifications Techniques

The requirement of formal verification has become prevalent [160], especially in hardware verification [26], network protocol analysis [93, 96], and critical security system validation [160].

Formal verification has the advantage of providing a systematic method to explore protocol flaws [139]. It can be applied to designs described at many different levels of abstraction from the first level through implementation [97].

2.5 Motivation for Using Formal Verification Techniques

The formal verification of hardware and software systems has increased in popularity [35] using both model checkers and theorem provers [91, 24]. Model checkers generally are used on systems of manageable size, whilst theorem provers may eventually be used on huge systems [74].

Formal verification works by proving characteristics of the mathematical models of given systems. Interactive systems consist of a general model and an environment that interact with each other to give the result. Some systems do not need to terminate and in this case modelling their calculations as infinite sequences of states and determining their characteristics utilising temporal logic is necessary [130].

For interactive systems there are two main approaches to formal verification: model checking (algorithmic verification) and theorem proving (deductive verification). Model checking and theorem proving complement each other in their strengths and weaknesses. At this point, we could say that Alloy and Z3 have a relative strengths in complementing each other as Alloy may show a counterexample and Z3 may prove it.

However, model checking is automatic; theorem proving is not. Theorem proving can handle complex formalisms; model checking cannot. Due to propositional logic being decidable, model checking is fully automated. Any formula of a logic with only limited types can be translated into propositional logic b cut this causes an explosion in the size of the translated formula. Often this makes the propositional decision procedures take an unacceptable amount of time or space and as a result expressive system logic in propositional logic will not work. So, model checkers are not used as decision procedures. Instead, researchers have utilized theorem provers by dividing a problem into model checkable parts whose correctness results are then reconstituted in the prover. Or, researchers have utilized the decision procedures in theorem provers to support abstract models of a checkable size [7].

Alloy gives us the option to visualize the model and related counterexamples (if any) using either the standard visualization form or the tree structure of the model which would include counterexamples if they exist [12]. In our models, we have chosen to display the counterexamples that were found by Alloy using the standard visualization as it is easy for us to understand and trace the problem [7].

2.6 Model Checking of Protocols

Models of security are used to provide a formal statement of a system's integrity, availability, and confidentiality needs. Models of security offer a precise and brief means of formally explaining security policies and assuring their validity. Since a system must not only be protective, formal models of security offer system designers assurance that they are building a consistent system, and with a foundation for future explanations that the system as established attains its specifications [108].

For example, in [106] the wireless authentication protocol is validated and in [107] the security protocol is modelled and validated.

An enormous collection of model-checking approaches has been developed for analysing security protocols, such as [6, 20, 53, 110, 112, 148, 144]. The general security problem is undecidable and, even under the assumption of perfect cryptography, verification of such properties is undecidable if the size (scope) of the problem of a model unbounded [111].

The biggest concerns for security is attacks such as: man-in-the-middle attacks where an attacker is involved in two parallel executing sessions and passes messages between them; replay attacks, where messages recorded from previous sessions are played in subsequent ones; reflection attacks where transmitted information is sent back to the originator; and type flaw (confusion) attacks, where messages of different types are substituted into a protocol (e.g., replacing a name with a key) [37]. The attacks problem is undecidable if the number of scopes of modelling the specification of an attack is unbounded since undecidability results show that a modelled case is too complex to detect the flaws of a protocol against an attack [111].

Model checking has been utilized in hardware, communication, and protocol verification [88]; recently, the trend is to apply model checking to analyse specifications of software systems.

Model checking has the advantage that the counterexamples find visualizations of actual attacks on the protocol. This gives insight into protocol vulnerabilities, and how they can be fixed. When the system has a defect and its required properties are not theorems, a decision procedure (model checking) can introduce a counterexample which can be invaluable in following the source of such errors in the system execution.

Under a model checking approach, a system implementing a security protocol is represented as a transition system with limited but numerous states. After that, the model checker utilizes many effective state detection techniques to detect whether the system can be in a state concerning a security infringement.

A model checker has been used to check the ABP (Alternating Bit Protocol) which is used to transfer messages in one direction between a pair of protocol entities [160]. Pagination messages that the client sends to the receiver and the acknowledgement pages that the receiver sends to the client using either zero or one are very simple to describe in the model checker [160]. As [160] reported, when the pagination is not limited to zero or one, i.e. in the sliding window protocol, the page number, acknowledgement number, and size of the sliding window may be any number, and that make it difficult for it to be modelled by model checking. Thus it is necessary to use an abstract model to solve this problem. Model checking has been applied widely and successfully in the analysis of security protocols [104]. There are many model checkers that have been used to verify security protocols as seen below.

2.6.1 Model Checking Using Spin

Simple Promela Interpreter (SPIN) [75] is an open-source model checking tool and is especially appropriate for synchronous systems. The models in SPIN are specified utilizing the Promela language and are also executable. The model checker allows both the simulation and exhaustive analysis of the behaviours specified.

The SPIN model checker is used in [129] to check the security protocols for sensor networks by building an authenticated routing application, and using a two-party key agreement protocol. As a result, SPIN has the ability to achieve data integrity during data authentication, which is a stronger property in the protocol.

2.6.2 Model Checking Using Linear Temporal Logic (LTL)

The model checking LTL is a modal temporal logic with modalities pointing to time. In LTL, the user encodes formula according to the future of path formula. For instance, the correctness of a condition at the end depends on the correctness of the initial fact. It enables both branching time and quantifiers [59].

LTL requires a reduction in the number of states and transitions and number of the accepting states to detect counterexamples efficiently. Its connected components are weak and costly, the automaton is commonly small, and the saving over verification can be very high [99]. However, our models such as transmitting data over single and multichannel in WLANs requires the number of states to be increased for detecting a counterexample, and connecting between signatures to be strong, to achieve the strong relation which lead to effective instances.

The security protocol that has been established by [29] depends on an insecure channel. For example, getting a confidential security protocol begins from modelling the properties of that protocol to detect its security flaws. Writing formula that utilize LTL, enables for the specification of assumptions on basis and communication channels beside to complex security properties. That are commonly not handled via state-of-the-art protocol analysers to discover of a critical security flaw in Google's SAML-based SSO for Google Apps. Also to discover a new attack on a patched version of the Asokan, Shoup, and Waidner (ASW) contract signing protocol.

2.6.3 Model Checking Using Alloy

Alloy is a tool for modelling and analysing systems. Modelling in Alloy is structural, logical, and intuitive. The specification language is based on typed first-order logic and as such, expressive enough to enable us to capture complex behaviour and specify and analyse interesting security properties such as confidentiality, integrity, authenticity, and non-repudiation. If a system can perform these four interesting properties at all times, then we can ensure that the system is *secure*.

Alloy is a lightweight language. Its specification language is based on first order related logics. It is designed for software design. When using Alloy to support program construction, developers may begin from simple and small models of the systems and expand slowly. At every stage, the Alloy Analyser will routinely offer the developer an instant response. Alloy is very helpful in sensing errors and defective parts of a system, reporting them to the developers and thus accomplishing an improved understanding of end user needs [80].

The Alloy tool provides a software design framework for modelling of critical design properties just like checking them [131].

2.6.4 Alloy and Security

Alloy is used in network systems and web based applications to check for security holes [83]. In the context of security, Alloy and the Alloy Analyser can be used to model network protocols and identify security loopholes that will make the application vulnerable to attacks. Some trials [145, 42, 76] have been made in the past to automate reasoning about web applications and network systems focusing mainly on security issues such as the correctness of access control that affected the users. Alloy and the Alloy Analyser tool are now the language that can be used by application and web application software engineers to identify and report security problems [134, 54].

2.6.5 Alloy and Security Definitions

In network communication and web-based applications, security is the most important aspect. In this project, analysing the security of network systems and web applications using Alloy is framed by the following benefits and features that the user will receive.

- Confidentiality

Confidentiality is the perception that communication made in network systems and through web application can only be received by the intended person. There should be no other reader or listener when this information is passed through networks and other application networks. For cryptography oriented applications, this signifies that those not intended to read or listen to this communication that is, anybody who does not have the decryption key, cannot access the plain text communication message in the encrypted text [125].

The Alloy language has been used to model security requirements for secure communications [125], where predicates were specified for secure message confidentiality, integrity, authenticity and non-repudiation. The work was successful in designing a general, reusable model for communication security properties.

- Integrity

Once a message is protected, it cannot be modified by a process. Changing a message requires accessing the protected contents of a secured message to read it, modify the contents while they are unprotected, and then re-securing them using the same policy. Thus, it cannot be differentiated between the original and the modified messages. We require that attempts to change the original message without write permission destroy the integrity of the protected message, so that after reading the contents of the message, it is easy to determine the modification [125].

- Authenticity

Authentication is an approach which identifies what was done from where, what was said, and particularly, which processes produced a certain communication. To do this, we use several built-in practical methods to identify the areas that may leak communications in network systems and web-based applications, and report them.

The first is an idea like a personal identification number for an automated banking system. The ATM system will do a checking of the customer entered PIN code with what the system stores to determine whether the PIN is the same as what the system stores, and then the client is logged in to transact. Alloy can model that by connecting two possible states (pin, pin for before and after states respectively) of an ATM system. In our ATM model we define the relationship between two states: before, when then ATM is waiting for PIN, and after when the PIN is typed. In our model we suppose the PIN always valid in any ATM. In open or confidential key encoding system, the personal key is an undisclosed code known by the sender, and every open key owner that read a legitimate communication message may understand who sent it.

In this research, we make use of a method related to the open confidential key encoding technique, although it does not use encoding as a protection strategy, but rather uses the idea of authenticity to recognize who sent the communication. In Alloy and the Alloy Analyser, a collection of practices can have write authentication, so the definitions simply narrow the recognition of the communication to an entity within the group. In circumstances providing only a single procedure, an Alloy description offers precise authentication [125].

- Non Repudiation

Non repudiation means the incapability of a process to deny passing communication it receives or to refuse to receive a communication. Non repudiation is tricky to use in network systems and web-based applications because information about communication must be passed and received at the local sender and the receiver correspondingly. To articulate non repudiation in network systems and web-based applications we need to check the security status of the systems. In this context the systems will be checked to make sure that they are secure and safe to receive and send information across the networks.

For Alloy to be efficient and effective here it must know the sending and the receiving system internet protocol, file transfer protocol, and hypertext transfer protocol so that it may analyse the validity of data packets that are received and sent. In [125], the system does not permit any instance to be sent or received unless it is valid, secure, and authenticated [125].

2.6.6 The Importance of Alloy for Detecting Flaws Using Small Scope

The general problem of determining whether a model meets its specification is undecidable, so automation can only be obtained in return for some compromise. In our case, we want to detect as many errors as possible before developing and executing our protocol, so we have compromised completeness. The problem of state explosion with model checking [33] is one that [81] suggests requires a pragmatic approach to solve [28], by using the Alloy language in a small scope hypothesis, which conjectures that many flaws in models can be detected in small instances. Our work assumes the small scope hypothesis for detecting security flow tracing, making examples of instances easy to be examined and Alloy problems easy to be checked, as a limited small scope can be easy to compute [113]. With our current work, a limited scope was sufficient. The impact of limiting scopes to be small in our work to accelerate detecting flaws and generating a counterexample if one exists.

2.6.7 Further Alloy Applications

Alloy has recently been utilised in the context of security assessment, for example to model the Java Virtual Machine (JVM) security constraints [140], access control policies [152], or attacks in cryptographic protocols [101].

Alloy is characterized as easily defining the meta-model, as [25] found when analysing avionic architectures characterised as models in first-order logic to facilitate specification of the relational properties expression to be checked. Both [25] and [101] expressed the specific architecture of a case study in Alloy.

Moreover, [101] has chosen the Alloy modelling language and its related automated reasoning tool because their powerful support for sets proved worthy when they tried to model and simulate the PIN decimalisation attack on IBM's architecture.

Alloy has proved its success in checking designs of diverse applications, such as the Intentional Naming System for resource discovery in mobile networks [4], and avionics systems [43], as well as designs of cancer therapy machines [85].

The Alloy Analyser has become increasingly popular and is applicable in a variety of different applications. For example, in [119] a design visualising the possibilities of detecting a MitM attack while using a credit card online is investigated. Alloy has been used to model the efficient and Secure Card-based Payment System (ESCPS) with no counterexample found. In [116] Alloy is used to automate the specification and verification of an Aspect UML model used in Aspect oriented (A-O) programming. In [8] the Alloy analyser is used to explore design weaknesses in the early stages of software development in an e-commerce application that is vulnerable to MitM.

Alloy has also been used for the modelling and analysis of protocols in networks [61] and distributed systems [150], and to model and emulate partial and complete attacks [101].

2.7 Theorem Proving for Proving System Satisfiability

2.7.1 Introduction

Theorem proving is a technique that is used to support software development and can help with automating the process of validating large sized codes with optimal cost [154, 71]. To achieve verification of secure systems in security applications, proving whether a specification satisfies security models may be achieved by using a theorem prover for automated support [14, 154].

Theorem proving is a tool composed of strong combinations of inference steps. It is one of many methods which are used for verification of specifications of system models and to produce new properties of concerns [38]. Theorem proving is undecidable and one of the key approaches to formal verification. It is a technique wherein the system and its required properties are expressed as formulas using mathematical logic which is provided via formal system, which defines a set of both axioms and inference rules [141].

Under a theorem proving approach, the system and its properties are discussed via logical formula and the formal proof is found via proving theorems that state that the provided properties hold in the system.

A theorem prover has the advantage of checking a proof mechanically, or verifying that a series of formulas does not comply with a legal derivation in the proof system. To do this, it needs to check that every derivation in the series is either an axiom or follows from the prior ones via a legal application of the rules of inference. Moreover, it can help the user in the building a proof by performing many heuristics. In addition, it is able to remember the decision steps for the part to determine if the part is a theorem or not [139]. Also, if a goal needs to be verified, the theorem prover divides it into sub-goals, and each step of the prover proves each sub goal automatically [38].

There are many theorem provers in active use today. The popular ones include ACL2 [90], Coq [57], HOL [66], Isabelle [127], NuPrl [36], and PVS [121].

In our work, theorem proving has been chosen for analysing, checking, and proving properties of our system.

2.7.2 Difficulties and Advantages of Using Theorem Proving

Theorem proving is a very general approach and one of its advantages is that it is easily applied to a wide variety of systems, especially infinite-state and parametrised systems. However, one of its disadvantages is that a high degree of highly skilled work is required, even when proof tools are utilized. The difficulty with a theorem prover is that the user needs to be very familiar with technology and have good knowledge of the specification.

However,program verification techniques such as the B-method and theorem provers such as HOL have become widely accepted [142]. Recently, new versions of HOL theorem provers such as HOL-light and Isabelle HOL have been introduced [142]. These provers need numerous human interventions and wide, manual guidance while the proof is running. Faster provers are required, especially for massive programs with thousands of invariants with minimum human intervention within a minimum period of time [11].

Theorem proving has an advantage in giving the user control of the proofs and it is flexible in dealing with proofs. Another advantage is that if there is a model that the model checker cannot verify because of the size of the state space is too large, a theorem prover may be able to prove it without have to take into account the size. Regardless of the size of the state space, it is not a problem for a theorem prover because it does not need to use abstraction techniques which make verification easy to be implemented on the whole ready model. This makes using a theorem prover required for verifying any size of code with law cost [154, 71].

Another advantage of using the theorem prover besides reducing the cost of verifying codes, as [154] mentioned, is that it helps to develop numerous problem theories, by being used either in subsequent analysis or as guidelines, which appears in its ability to detect an ambiguous problem. [128] confirmed this property when he used Isabelle to prove and discover a problem with the security of cryptographic protocols in the massive theory that was built up.

Also, [128] added another benefit which is that using a theorem prover can save time and reduce the analysing and verification protocol procedure from weeks to minutes.

2.7.3 Theorem Proving and Security

There are many theorem provers that have been used to prove security protocols as seen below. Theorem provers are increasingly being used today in the mechanical verification of safety-critical properties of hardware and software designs [141]. To achieve verification of secure systems in security applications, proving whether a specification satisfies security models or not is required by using a theorem prover for automated support and evaluation of any system [14, 154].

2.7.4 Theorem Proving Using Isabelle and Coq

Isabelle and Coq are generic theorem provers. Isabelle is equipped with comparatively robust automation, while Coq depends on a theory called Calculus of Inductive Construction and enables one to extract certified programs from proofs.

[126] utilized Isabelle with higher order logic (HOL) to prove three basic security properties of a Transport Layer Security (TLS). This is a protocol that guarantees a secure connection over transport protocols such as TCP and under application protocols like HTTP and SMTP with HTTPS which support them by the secure connection version. The three basic properties are: (1) every client, after sending

its completed message, and receiving a matching completed message from its peer, records the session parameters to enable sending and receiving again. The peer's identity can be authenticated utilizing public key cryptography, (2) neither the eavesdroppers nor any authenticated connection can get a negotiated secret, and (3) any message communication modification that can be made by the attacker can be discovered by the connected parties.

In proving these three basic security properties of a Transport Layer Security (TLS) above, the use of Isabelle failed to detect whether the connection between parties is secure, especially in the prescience of passive eavesdroppers because the model does not support unauthenticated connections. After that [151] has translated the formal model of Isabelle into the Coq syntax and built the proof in the Coq system. Despite Isabelle and Coq theorem provers sharing many common principles, their utilization and their techniques differ dramatically, and consequently, the resulting proofs using Coq are more clearly built than those done in Isabelle.

2.7.5 Theorem Proving using Z3

Our approach uses the Z3 SMT solver, which is a high performance theorem prover developed at Microsoft Research and an automated satisfiability checker [22]. Z3 checks whether the set of formulas are satisfiable in the built-in theories of Satisfiability Modulo Theory (SMT) [147].

Z3 has several facilities of application programming interface to make it straight forward for components to map into Z3, but there are no user centric facilities or stand-alone editors for interacting with Z3 [16]. Here, we used Z3 directly online using the URL ”http://rise4fun.com/z3”

2.7.6 Security Protocols using Z3

According to [15], web application and networks are experiencing calamities. A carefully considered attempt is required if safety, and secure development, coding, and scrutiny methods are to be merged with the swift design and installation of significant safety and security disseminated web-based programs and networks. Z3 has been used to protect users, companies, and their information by proving safety and security. For instance, consider an online banking system installed in the server at the bank head office from which users access all the services they need. However, before the bank web application is installed, Z3 must be used to prove satisfiability and to verify security and safety before implementation.

[17] says studies on safe and protected web-based applications and secure networks systems and protocols are making good progress now when Z3 is being used to check any bugs or security pitfalls, and to make sure that no application will allow unauthorized persons to access confidential information. Studies on application authentication are also progressing quickly, and the outcome is a set of tools that can now locate errors in complex programs.

2.8 Existing Approaches Combining Model Checking and Theorem Proving

Using formal techniques raising two main questions about the connection between a system and its formal description: [98]

- Does the model description visualise the specific behaviour of the system? To answer this question, the model needs to be checked using a model checker methodology.
- How can we get an implementation that satisfies its specification? To answer this question, the implementation needs to be proved using theorem prover methodology.

We have described the two main approaches to formal verification, model checking, and theorem proving. Both of these approaches have different strengths and weaknesses, and there has been much work in trying to combine them to be able to verify larger and more complex examples faster, more conveniently, and with less user time required. Below we discuss briefly a few of the most well-known techniques that combine model checking and theorem proving.

There are several examples of combinations of theorem provers and model checkers [49, 95]. Usually, the theorem prover is utilized to divide the proof into different sub-goals to be small enough to be verified by a separate model checker. There is no actual combination so the translation between the languages of the theorem prover and the model checker is done manually. In our work we translate from model checking into theorem prover manually.

Usually, integrations of theorem provers with model checkers make the theorem prover able to call upon the model checker as a black-box decision procedure provided as an atomic proof rule [122, 137]. The prover converts expressions belonging to values over finite domains into propositional expressions. This enables utilisation of the result as a theorem (as in our framework with the reverse direction) but this does not extend easily to a fully expansive approach. It achieves better efficiency at the cost of complex integration and lower assurance of safety.

Theorem provers have been utilized to help with abstraction (used to reduce the size of the model) [34, 67, 146] for model checking. Model checking in the theorem prover is utilised to provide assumptions to refine an abstraction that appeared to be too complex and which adds too much non-determinism to the system which led to the prevention of pseudo counterexamples.

Traditional formal approaches like model checking and theorem proving have been used widely in the field of security protocols. Here the data communication is trusted and requires the user to decide on authentication aims in advance. Model checking and theorem proving are used to model the behaviour of a protocol and mathematically verify that the design and implementation satisfy requirements of the protocol successfully [52].

Model checking discovers a path that leads to a state where the properties are not hold and focuses on searching incorrect traces, while theorem proving uses theorems in order to prove the satisfiability of the protocol properties and focuses on proving the correctness of the protocol according to the properties.

In order to utilise theorem proving, we need to convert our model verification into a theorem prover which requires encoding both the design and the specification into the logic of a theorem prover.

In the Nimbus Toolset [38, 87], there is a translator to enable verification as well as a translator to the NuSMV model checker. In the Prototype Verification System (PVS) theorem prover, this translator is responsible for interpreting requirements specifications from the Requirements State Machine Language (RSML) specification language into the input language. This enables the user to verify interpreted requirements specifications easily via theorem proving and model checking techniques. Verifying the safety properties of systems using models specified in RSML using theorem proving has been done by [87] with both the PVS theorem prover and the NuSMV model checker to analyse the Flight Guidance System (FGS) model for Mode confusion [87].

[56] describe a case study in which the Alloy address book problem is analysed using the Yices SMT solver. In this case study the Alloy language is translated to Yices and analysed utilizing the bounded Yices SMT solver rather than the

bounded SAT solver. They concluded that whenever the Alloy Analyser returns a counterexample, Yices returns a valid counterexample as well, and when Alloy cannot find a counterexample in a finite scope, Yices does not find any in that scope either. Also, analysing problems using the SMT solver is faster than using the SAT solver. Our case studies have been translated from Alloy to the Z3 and analysed using the unbounded Z3 SMT solver rather than the bounded SAT solver and we conclude the same as[56]. However, the difference between our results and theirs is that we use the unbounded SMT solver while [56] used the bounded SMT solver.

In our case study, we try to create a balance between the expressiveness of theorem proving and performance as well as the efficiency and automation of model checking. Representing the requirements of a case study is the main problem in theorem proving formulation and is one of the main contributors towards success or failure in model checking.

2.9 Discussion

According to our study of the literature above, it is not clear whether model checking or theorem proving is better for establishing properties of communications protocols. We cannot determine which formal method is better unless we determine the goal of using them. If the the goal is finding mistakes and just checking not proofing, and the domains are restricted to be finite (using scope), then we need to use a decidable model checker because the first period for realising a scenario is modelling and seeing the result, and in this period lots of mistakes may occur.

In contrast, an SMT solver like Z3 is undecidable. Decidable means that a model checker like Alloy should provides the result even it takes very long time with finite states, while undecidable means that a theorem prover like Z3 may run forever and not provide the result or even time is out, and just provides "unknown". However, if we need to prove the correctness of the result that has been got from model checking for more confidence, we need a theorem prover.

Z3 supports unbounded verification and the SMT solver tries to find the counterexample (CE) for unlimited scope. When it gives the result as "there is no CE", it guarantees that the assertion is 100% valid because it covers all possible scopes, even if it may take a longer time to prove the validity of the assertion. However, Z3 does not guarantee a complete analysis: it may find a counterexample preceded by the keyword unknown, implying that the property may or may not be valid, or time out.

On the other hand, Alloy supports bounded verification and when it says there is no CE, then the assertion may be valid, i.e, the Alloy Analyser checks the existence of the CE based on a finite scope. In this case the assertion is valid and there is no CE within a finite scope which means the number of atoms for each type do not exceed the number of scopes and the maximum that the assertion can take is the number of scopes we specified. However, it may be invalid (not guaranteed) in the larger scope when we increase the number of scopes.

Also, Z3 proves the assertion, while Alloy just checks a model and does not prove analysing, and proof is stronger than modelling. However, Alloy could be stronger than Z3 in controlling the number of scopes to find the problem for the model if the goal is just finding a counterexample to limit the problem. For example, limiting the length of the key for encryption to guarantee it is not able to be broken.

Consequently, our work focuses on proposing a substantial experiment to evaluate both model checking and theorem proving in understanding and characterising MitM, and exploring solutions to MitM.

So, for developing a solution we need both model checking and theorem prover. In the case when a theorem prover fails to prove a theorem, it can be difficult to tell what's gone wrong: whether the theorem is invalid, or whether the proof strategy failed. On the other hand, if the model checker finds no counterexample, the assertion may still be invalid unless we increase to a large enough scope. In our case study, increasing scopes are required by extending the complexity of the proposed solution such as the communicated parties, number of MitMs, and the procedure of the solution.

We use Alloy to generate counterexamples which help in detecting flaws in the proposed solution (transmit data securely over multichannel) compared to the existed solution (transmit data securely over single channel).

We use Alloy to explore axioms which help in reducing the appearance of counterexamples. Moreover, we develop the properties to be applied in the predicates of the model based on the explored axioms.

The Alloy analyser is a refuter rather than a prover [79]. It can falsify a model but not verify it. It can prove that an assertion does not hold for all instances of a model by finding a counterexample, but it cannot prove that an assertion holds in all instances of a model. It can only prove that an assertion holds for all instances up to a bounded size. Consequently, we need to the Z3 SMT unbounded verification to prove that an assertion holds in all instances of a model with confidence in the correctness of the property.

It is difficult to compare the techniques of Alloy and Z3 because both of them work with different mechanisms. Alloy transfers the model into first order logic and then the Alloy Analyser translates the model with its constraints to propositional logic, while Z3 transfers the model into first order predicate logic. The main feature of the Alloy to Z3 SMT translation is that it guarantees equisatisfiability of the produced FOL proof obligation with respect to its Alloy counterpart. Consequently, the result of analysing the FOL proof obligation of an Alloy problem with an SMT solver, whether showing validity or providing a counterexample, is faithful. However, as we will show in the evaluation section, this is done at the expense of efficiency.

The next two chapters provide detailed information of the frameworks and tools used in this dissertation. We begin with Alloy followed by Z3, using an ATM system as an example in both frameworks.

Chapter 3

Foundations: Alloy

This chapter provides detailed information about the framework of Alloy, the Alloy modelling language and its supporter verification tool the Alloy Analyser. This can be used to automatically analyse properties of Alloy models. The chapter provides a foundation in using Alloy to model a system and the Alloy Analyser to check the validity of the properties up to the limited scope using an Automated Teller Machine (ATM) as an example.

3.1 Alloy Definition and Process

Alloy is a tool for modelling and analysing systems. It consists of two parts, the Alloy language and the Alloy Analyser. The Alloy language is a tiny modelling language used for the expression of the fundamental structure of a system, as well as restrictions and operations to determine how the system may change. The Alloy language consists of Boolean algebra, set theory, quantifiers, and first-order relational logic. In general, Alloy is a combination of logic, language, and analysis: *logic* because it depends on both first order logic and relational calculus, *language* because it has syntax for structuring specifications in the logic, and *analysis* because it uses bounded exhaustive search for counterexample to a required property utilising a SAT solver to analyse Boolean expression automatically [81].

We have chosen to use Alloy because it has the ability to make a fully automatic semantic analysis, which can provide checking of consequences and consistency, and simulated execution [86]. It also has the ability to generate sample transitions of an operation described implicitly, utilising negation and conjunction without sacrificing abstraction to achieve executability [84]. Moreover, it tries to find counterexamples, within a bounded scope, which violate the restrictions of the system [78]. Also, Alloy has an advantage for exploring design ideas gradually from a small model which is then developed to a larger scale; Alloy can analyse the model at any level.

The Alloy process passes through three steps. First one specifies the logic of theory which is a meta-model and constraints. Second one bounds the number of instances of each object in the meta-model to search and model (build). Third, the Alloy Analyser (AA) is used analyse the specifications to generate instances of the meta-model that satisfy the constraints based on the second step. AA provides a model for a given logic of theory after doing inclusive verification for all models that have up to a bounded number of instances [50].

3.2 Alloy Analyser

The Alloy Analyser (AA) is the original analysis tool for Alloy. It can be used to edit, build, and test specifications written in the Alloy language. The goal of the analysis is to check if the property holds in the system, within a given scope. It provides two main forms of analysis: [81]

The first analysis, called predicate-running provides a simulation of the specification. It checks the consistency of a model, by finding an instance that satisfies the model. It is applied to Alloy problems with a predicate providing results as a visualization which shows satisfying structures called instances that satisfy the predicate and the Alloy model if the model is consistent. No instances will be visualized if the model is inconsistent.

The second analysis, called assertion-checking refutes the specification's correctness. It checks the properties of the model. It is applied to Alloy problems with an assertion providing results as a visualization which shows a structure (called a counterexample (CE)) that falsifies the implication of the assertion from the Alloy model. Otherwise, there is no counterexample found.

The Alloy Analyser is a GUI application that includes three main parts: for editing and adjusting specifications, for displaying information about solutions that the analyser generated, and for providing information about the internal data structures used by the tool during analysis. It is able to represent solutions in a graphical format.

Bounding a small number of instances of each object in the meta-model is better than bounding a large number because it makes it easier, faster, and accurate for AA to detect a problem in the theory if there is any. Moreover, if the model is getting huge and the bounded instances is getting large, then a number of instances may be lost and not investigated [50].

The Alloy Analyser works as a compiler and starts its process by implementing its analysis. Its analysis lies in using a translator called "KodKod" as a model finder to translate the Alloy problem (verification queries) expressed in relational logic formula from FOL into a corresponding propositional logic formula (Boolean logic formula). It is then delivered with the scopes (that a user provided to bound the size of Alloy problem types) to the SAT solver to check its satisfiability by checking the *negation* of the checked property and find a solution by answering verification queries. After that, if a solution to the Boolean formula is found, it is translated back into an instance of the corresponding relational formula. This involves binding the formula's relational variables to constants. The problems that are solved do not exceed the scope that the user specified to bound the size of the domains, which has the advantage of making the problem finite [83]. Consequently, the Alloy Analyser has no ability to prove the correctness of an Alloy assertion.

Figure 3.1 shows the internal Alloy analysis process of checking an assertion.

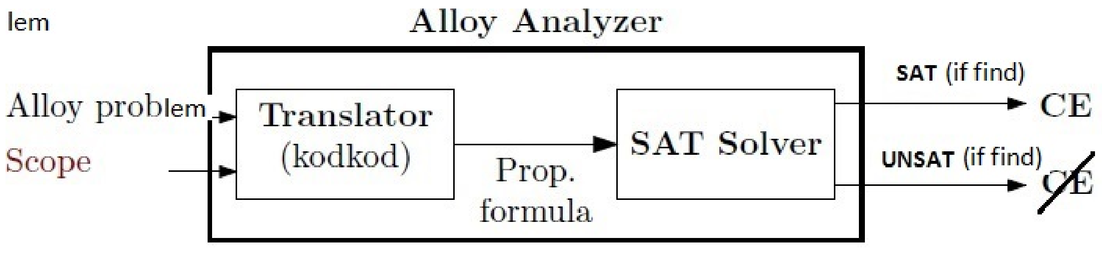


Figure 3.1: An Abstract View Of The Alloy Analysis Process of Checking An Assertion [62]

3.3 Motivation of Using Alloy

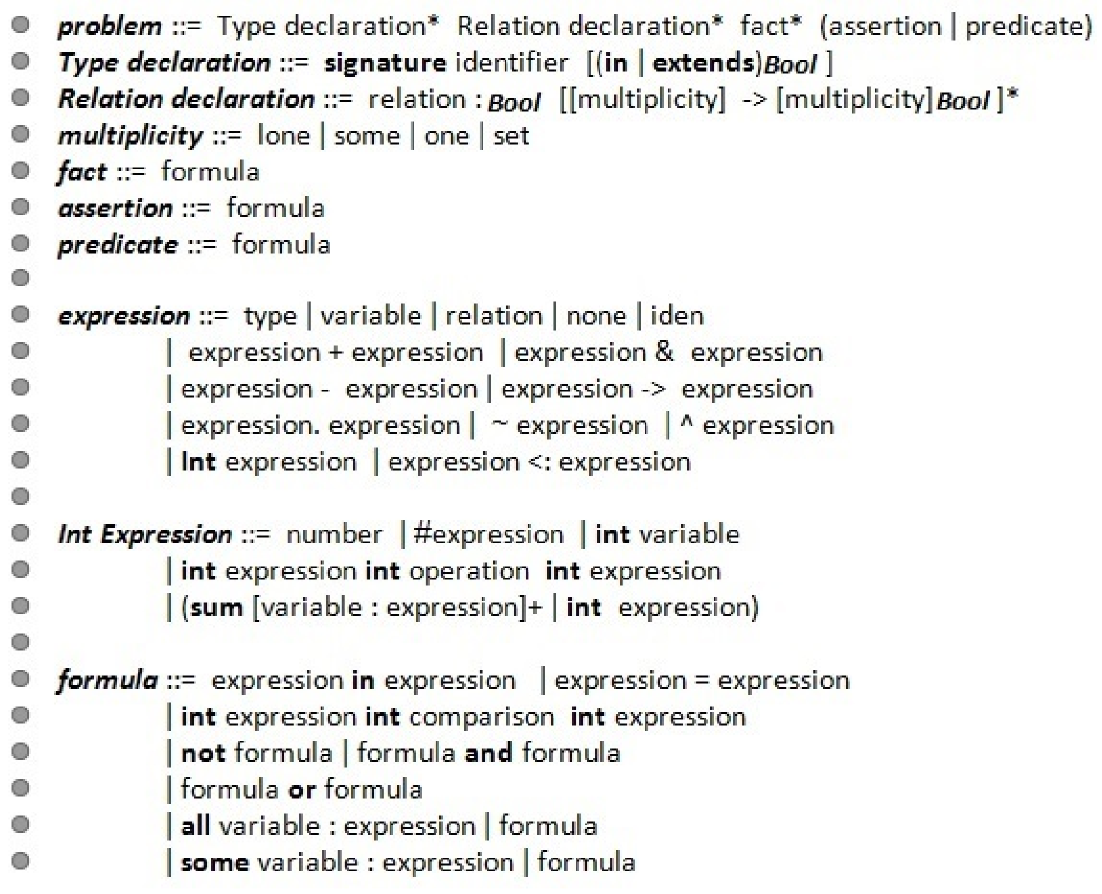
One of the motivations for the development of the Alloy language, according to [81], was the lack of tools with automated analysis capabilities. In Alloy, the proposed approach can be modelled in the early phases of the software development life cycle.

Another motivation for using Alloy is the use of fact to analyse and restrict Alloy models automatically, and also generate a counterexample which is considered as a solution [82].

While there are many model checker tools available at present, the Alloy specification language has been chosen for use primarily because of its simplicity to represent program language abstractions and its ability to explore their semantics with a well-integrated analysis tool. As [81] points out, according to his approach as "lightweight" formal methods, Alloy models can be created simply and tested early during the development process, and then extended gradually. In addition, [81] illustrates that the aim of Alloy was to get the benefits of traditional formal methods in a short time with less effort.

3.4 Alloy Problem

As seen in Figure 3.2, an Alloy problem is composed of a combination of type declarations, relation declarations, relational first order formulas marked as fact, and another formula (if required) marked as assertion to check or as predicate to run [81]. Each component will be explained later.



```
● problem ::= Type declaration*  Relation declaration*  fact*  (assertion | predicate)
● Type declaration ::= signature identifier  [(in | extends)Bool ]
● Relation declaration ::= relation : Bool  [[multiplicity]  -> [multiplicity]Bool ]*
● multiplicity ::= lone | some | one | set
● fact ::= formula
● assertion ::= formula
● predicate ::= formula
●
● expression ::= type | variable | relation | none | iden
●              | expression + expression  | expression &  expression
●              | expression -  expression | expression ->  expression
●              | expression. expression |  ~ expression  | ^ expression
●              | Int expression  | expression <: expression
●
● Int Expression ::= number  | #expression  | int variable
●              | int expression int operation  int expression
●              | (sum [variable : expression]+ | int  expression)
●
● formula ::= expression in expression   | expression = expression
●              | int expression int comparison  int expression
●              | not formula | formula and formula
●              | formula or formula
●              | all variable : expression | formula
●              | some variable : expression | formula
```

Figure 3.2: Abstract Syntax For The Core Alloy Logic [62]

3.4.1 Type Declarations and Relation Declarations

The type declarations introduce the global types (called signatures) which are a central part of Alloy's modelling specification. Signatures are the entities of systems which represent sets of atoms. Each atom (or scalar) is a unity and includes three main properties: it is indivisible (are not able to be split into smaller parts), immutable (their properties do not change), and uninterpreted (there are no inherent properties) [9].

Signatures are utilised to define the existence of a unary, binary, or ternary relation while fields are utilised to define other relations of those signatures.

Each signature declaration, such as signatures A and B below:

```
sig A{}
sig B{}
sig C{r:B m  -> n A}
```

declares a top-level type, here named A and B. The identifier A and B always refers to the set and does not include number of relation declarations (called fields of signatures).

The identifier C refers to a set including the number of relation declarations (called fields of signatures) which represent sets of n-arity tuples depending of the relation arity, for example *r: C->B ->A* declares a ternary relation named r $\subseteq$ C $\times$ B $\times$ A.

These may also be constraints on field values such as *n* and *m*. *n* and *m* restrict that for each c $\in$ C the binary relation c.r maps each tuple in B to n tuples of A, and each tuple in A to m tuples of B.

Signatures such as A and B, called *top-level signatures*, are not a subset of another signature and they will be implicitly disjoint unless they show the relation (*field*) between them such as relation r between A, and B below.

Each top-level signature limits a maximum set of three *atoms* by default.

```
sig A{}
sig B{r:A}
```

A signature can contain at least zero relation declarations, separated by commas. Each declaration indicates a (unary, binary [relates two atoms], or ternary [relates three atoms]) relation or maybe more than 3 relations between the set defined by the signature as seen in Table 3.1 below.

A relation is a set of ordered tuples (vectors of atoms); every tuple is an ordered sequence of atoms, and each tuple lists atoms (entities) that are related to each other. Each pair represent a tuple. For example: in Table 3.1 below, the binary relation represents 2 tuples (a0,b0),(a1,b1), the ternary relation represents 3 tuples (a0,b0,c0),(a0,b1,c1),(a1,b1,c2), while the 4-relation represents 4 tuples (a0,b0,c0,d0),(a0,b1,c1,d2),(a1,b2,c0,d3),(a2,b1,c1,d3). Moreover, each element in each tuple is called an atom.

Each relation has a size equal to the number of tuples per pair in this relation. The arity of a relation is the number of atoms in each tuple of the relation and it is greater than zero. Some signatures may extend or include other signatures and constraints using multiplicities as seen in the next section.

Signature and Relations	Related atoms	(Type/size) of relation
sig A	none	Unary (set of a's), f={(a0),(a1),(a2)}
sig A f: B	(a, b)	Binary (from A to B), f={(a0,b0),(a1,b1)} f $\subseteq A \times B$
sig A f: B ->C	(a, b, c)	Ternary (from A to B to C), f={(a0,b0,c0),(a0,b1,c1),(a1,b1,c2)} f $\subseteq A \times B \times C$
sig A f: B ->C ->D	(a, b, c, d)	4-relation (from A to B to C to D), f= {(a0,b0,c0,d0),(a0,b1,c1,d2), (a1,b2,c0,d3),(a2,b1,c1,d3)} f $\subseteq A \times B \times C \times D$

Table 3.1: Relations And Their Types

3.4.2 Signature Extension, Inclusion, and Abstraction

The signature declaration

$$sig\ B\ (in \mid extends)\ A\{...\}$$

declares a subtype of A named B. Signature **extension** and signature **inclusion** define new signatures **B** as subsets of a main signature (its parent) called sub signatures. This serves as sub-typing and is utilised to support pyramidal specification. The key word *extends* indicates disjoint subsets if more than one signature **extends** a top level signature, while the key word **in** indicates not disjoint subsets if there are more than one signature **in** a top level signature

The type signatures therefore form a type hierarchy whose structure is a forest: a collection of trees rooted in the top-level types, while signatures that do not extend another signature are said to be a type signature that is a top-level signature, and their type is a top-level type. Top-level signatures represent mutually disjoint sets, while sub-signatures of a signature are mutually disjoint. A subtype (subset) signature inherits the fields of the signature it extends, along with any fields that signature inherits.

Using the **extension** mechanism makes the model clearer and more modular by separating the main signature aspect of the model into separate paragraphs. The extended signature must be a top-level signature. A signature may not extend itself, directly or indirectly.

An abstract signature, marked *abstract*, is restricted to hold only those elements that belong to the signatures that extend it. If there are no extensions, then the marking **abstract** has no effect. An abstract signature represents a classification of elements that is refined further by more *concrete* signatures. If it has no extensions, the abstract keyword is likely an indication that the model is incomplete.

3.4.3 Multiplicity

Multiplicity constrains the sizes of sets. In a relation, multiplicity is how many atoms on one side are related with an atom on the other side. Multiplicities can be applied to the domain, range or both of a relation. Depending on where multiplicity keywords are placed they induce different constraints as seen in Table 3.2. Relation may be restricted by multiplicity key words *lone* which means only one or zero, *set*, which means any number, *one*, which means exactly one, and *some*, which means at least one. The default multiplicity keyword for unary relations is one and for multiple-arity relations is set.

Multiplicity	Constrains Meaning
r: A ->B some ->one C,	It constrains for each a: A the expression a.r to associate each tuple in B with exactly one tuple in C and each tuple in C with at least one tuple in B.
r: A ->B lone ->some C,	It constrains for each a: A the expression a.r to associate each tuple in B with at least one tuple in C and each tuple in C with at most one tuple in B.
r: X ->A lone ->B some ->set C,	It constrains for each x: X the expression x.r to associate each tuple in A with at least one tuple in B and each tuple in B with at most one tuple in A, and each tuple in B with any number of tuples in C and each tuple in C with at least one tuple in B.

Table 3.2: Multiplicity Constrains Meaning

3.4.4 General Example to Apply Type Declarations, Relation Declarations, Signature Extension, Inclusion, Abstraction, and Multiplicity

The example bellow models the relations between men and women as persons showing the difference between *in* and *extends*:

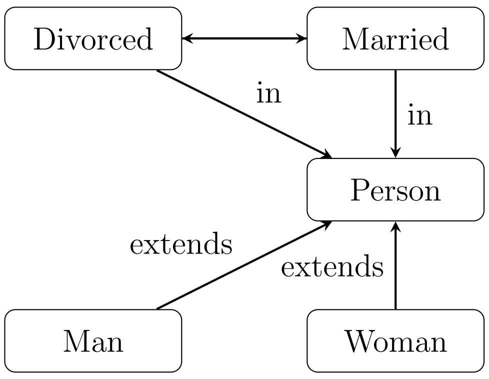
```
abstract sig Person
{
father: lone Man,
mother: lone Woman
}
sig Man extends Person
{
wife: lone Woman
}
sig Woman extends Person
{
husband: lone Man
}
sig Married in Person { }
sig Divorced in Person { }
```

In the example above, *Person* is a top-level signature that includes two fields, *father* and *mother*, to relate signatures *Man* and *Woman* with signature *Person.*

Each *Man* in *Person* could or could not be a father, and each *Woman* in *Person* could or could not be a mother, thus, the relation is restricted by multiplicity *lone* which means only one or zero.

Extends indicates that *Man* and *Woman* are subset of *Person* i.e, every *Man* is *Person* and every *Woman* is *Person* but *Man* cannot be *Woman* and vice versa. Therefore, *Man* and *Woman* are disjoint.

In the example above as seen in structure (1), *in* indicates that *Married* and *Divorced* are subset of *Person*, i,e: every *Married* is *Person* and every *Divorced* is *Person*, however *Married* can be *Divorced* and *Divorced* can be *Married.* Therefore, *Married* and *Divorced* are not disjoint



Flowchart (1): General Example for Type Declarations, Relation Declarations, Signature Extension, Inclusion, Abstraction, and Multiplicity

3.4.5 Fact, Predicate, and Assertion

Facts describe invariants and are always true.

Predicates in Alloy are declared utilizing the keyword *pred.* Predicates define and control operations as formulas and describe state changes. Predicates make

Alloy produce instances that satisfy with its restriction on each state. Predicates are composed of at least one constraint and are used to represent operations.

The command *run* is used to ask Alloy to search for instances that satisfy a predicate. It is applied to the signatures and relations and always gives a result of "consistent" or "may be inconsistent" depending on whether a model was found. The difference between a fact and a predicate is that a fact always holds whereas a predicate only holds when called an appropriate arguments.

Assertions in Alloy are declared utilising the keyword *assert*. Assertions are statements that must be true about the system. Assertions serve as checks to guarantee that the system is behaving correctly. Assertions are utilized to check specifications. An assertions is a formula whose validity is checked, taking into account the facts in the model. From assertion, the Alloy Analyser is asked to search for possible (finite) counterexamples based on the constraints assumed in the specification. The *check* command is given to the Alloy Analyser to search for counterexamples of an assertion. It enables the expression of properties that are anticipated to hold as the result of determined facts [62].

3.4.6 Expressions

Alloy expressions represent the fundamental buildings blocks of Alloy formula; they always evaluate to relations. There are two kinds of relational expressions, basic and complex. Basic Alloy expressions are constant relations; this includes all declared signatures and relations as well as the built-in constants: *none* for the unary empty set. Complex Alloy expressions are generated from basic expressions using Alloy's relational operators such as r + s (union), r++s (override), r & s (intersection), and r - s (difference) for same arity relations r and s. Also, r ->s is used for Cartesian product and r.s for relational join of arbitrary relations r and s as seen in Table 3.2.

Alloy provides integer expressions. An integer expression is different from a relational expression. An integer value is not considered as an atom; however, utilizing an integer in a relational expression, Alloy provides for every integer value x, Int contains exactly one atom that identifies that value [60]. It indicates rudimentary integers. The type Int represents the set of all atoms carrying rudimentary integers. The expression **Int x** denotes the atom carrying the integer denoted by the integer expression x, whereas **int y** denotes the integer value of the atom represented by the variable y. Integer expressions are obtained from an infinite set Z of numbers (. . . , -1, 0, 1, . . .), and are combined using standard arithmetic operators (+ , -).

3.4.7 Formulas

Fundamental Alloy formulas are formed from Alloy expressions utilizing the subset operator **in**, the equality operator =, the integer comparison operators less than < and greater than >, and integer equality =. Fundamental formulas can be merged using logical connectivities including conjunction (**and** or &&), disjunction (**or** or |), implication (**implies** or =>), and negation (**not** or ¬).

Complex Alloy formulas are created utilizing quantifiers. Quantified Alloy formulas take the form Q x: exp || F where Q is one of the Alloy quantifier: *all, some, no, one, lone* , x is a variable (usually) occurring in F and bounded by the Alloy expression exp.

However the expression b may not begin with a multiplicity keyword. This called a first order quantification, so x points to a single element of b. However, every Alloy expressions is considered relational, meaning that x is a singleton subset of b. The meanings of quantifiers are as follow: [60]

all a : A | of holds when Formula holds for every a in A, some a : A | of holds when Formula holds for at least one a in A, no a : A | of holds when there is no a in A such that Formula holds, lone a : A | of holds when Formula holds at most one a in A, one a : A | of holds when Formula holds for exactly one a in A.

However, if there is a formula of the form *r in e*, this says r is a subset of e. So, e can be prefixed with multiplicity keyword to restrict the value of r. The meaning of restrictions are as follow: [60]

r in (set e) does not induce any restriction and holds when r is a subset of e, r in (one e) holds when r is a singleton subset of e, r in (lone e) holds when r is a subset of e that contains at most one element, r in (some e) holds when r is none empty subset of e.

3.5 Counterexample, Scopes and Inconsistency

The counterexample explains (using visualization) why a model has not satisfied a specification. By following the counterexample we can determine the source of the error in the model and then correct it and repeat the process again as shown in [32, 123].

A scope is specified as a positive integer number to bound the domain (the number of atoms in each instance of each model element) in an instance of the system that the SAT solver analyses [78].

Checking Properties for the Alloy Analyzer are expressed as either predicates or assertions. For properties that are required should be expressed as predicates, Alloy tries to find an instance that satisfies the property. For properties that are required to be expressed as assertions, the Alloy Analyzer tries to find a counterexample that violates the property. The Alloy Analyzer finds the example or counterexample within a scope that is limited by the number of instances of each entity in the system [78].

A model checker either asserts that the properties holds or otherwise reports that they are violated, providing a counterexample which is a run that violate the property. Such a run can give valuable feedback and points to design errors.

3.6 Instances

Any instance of an Alloy model is regarded as an assignment of sets and relations to the signatures and fields [60]. The instance can be acceptable if its assignment corresponds with the declarations of the fields and signatures. Instances that satisfy the predicate and the model will be visualized if the model is consistent. No instances will be visualized if the model is inconsistent [81].

3.7 An Automated Teller Machine (ATM) Example

The motivation for choosing the ATM system and specifically the withdrawal transaction is that it is an example of privacy, integrity, and security in a system, and any mistake in its functioning may cause a disruption of economic balance [94].

3.7.1 ATM System Description

An ATM system [10] is an electronic machine which allows customers to access their bank accounts in order to perform certain transactions, such as withdrawing money, making a balance enquiry, and transferring money securely. These services depend on the accuracy of the ATM process. Thus, visualization of these transactions is necessary.

In this section, we will describe, analyse, determine the properties and requirements of ATM system, and model ATM using Alloy. Consider the interaction between a bank customer and an ATM cash machine. The ATM main function is providing limited amounts of money through withdrawal transactions to correctly authenticated users. Users use a card and PIN as authenticated tools [21]. However, the ATM may provide extra functionality such as balance enquiry and money transfer that are not included in our model. We only focus on withdrawal transactions.

An ATM system has modelled, analysed and visualized both balance enquiry and withdrawal transactions by [21]. The difference between our model and the system modelled by [21] is that their model depends on examining the behavioural properties of the ATM, to visualize the interaction between a customer and an ATM user interface. This is to solve a common user error with an interactive system, when users forgot their cash card after withdrawing cash.

Our model examines the behavioural properties of an ATM to visualize the interaction between a customer and the ATM processes during two sequential operations:

- The first operation is similar to [21]; the user inserts a card and types a PIN, and then the ATM reads the card to continue with the withdrawal.
- The second operation, occurs when the user requests an amount of money to withdraw. The ATM checks whether the requested amount is available, and if so, reduces the account balance and dispenses the money. If the amount

requested is not available, the card is returned and the balance remains the same.

3.7.2 ATM System Analysis

The withdrawal transaction requires a sequence of six statuses and operations. Its process passes through five procedures. The ATM status begins with, *WaitingCard* for a (bank card) to be inserted, followed by the required operation which is *EnterCard*.

The second status is *WaitingPin*, followed by the required operation for this status, *TypePin*. During these statuses the balance remains the same.

The third status is *WaitingMoney* which requires the desired amount of money to be typed, followed by the required operation for this status, *RequestCash*.

The next status is either the fourth or the fifth status. The fourth, *WaitingReceiveCashAndCard*, is achieved if the result of the previous operation, *RequestCash*, is less than or equal to the balance of the account. The equivalent operation for this status, *ReceiveCashAndCard*, follows, reducing the balance by the requested amount.

If the main property of *RequestCash* is not achieved, meaning the *RequestCash* operation is greater than the balance, the fifth status, *WaitingReceiveCard*, is activated and followed by its equivalent operation, *ReceivedCard*. The balance of the account remains the same.

The final status is *Update*, which returns the system to its first status, *WaitingCard*. ATM model has system properties and system requirements which will be detailed next.

3.7.3 ATM Model Properties and Requirements

In the ATM system above there are system requirements and system properties. The ATM system achieves its function correctly when the system properties satisfy the system requirements.

The model of the ATM system can be enhanced by adding properties (axioms), which are written as logical formulas. In Alloy, properties or constraints are defined as facts and assertion. The **first property** is that all ATM cards with corresponding PINs should be identified in any ATM machine; the **second property** is that the balance and PIN of cards that interact with the ATM should not be less than zero (positive); and the **third property** is that the required amount of money should be less than or equal to the available balance and greater than zero for the used card.

The system requirements are inserting the card, typing the corresponding PIN, and finally requesting the available amount of money.

3.7.4 An Alloy Specification of An ATM System

In Appendices (A.1, A.2, A.3, A.4) and Line (2) in Appendix (A.6), we give a sample model in Alloy of an ATM system.

- **Signatures, Abstract, and Extension**

In our model as seen in Appendix (A.1), the type hierarchy consists of the four top-level types *Operations* for customer operations (Line 2); *ATM_Status* for statues changes in ATM system (Line 4); *Card* for identifier in ATM system (Line

6); and *ATM* for ATM system itself contains some fields for showing relations with other signatures (Line 7). All top-level signatures are mutually disjoint. For example: disj(Card, ATM).

Types (signatures) are declared using the *sig* keyword and represent sets of elements. The top-level types *Operations* (Line 2) and *ATM_Status* (Line 4) are the basic types and labelled with the keyword *abstract*. For top-level type *Operations*, the keyword *abstract* constrains that every element in the *Operations* type is an element in (belongs to) one of its extensions *EnterCard*, *TypePin*, *RequistCash*, *ReceiveCashAndCard*, and *ReceiveCard* (Line 3) for five customer operations: inserting card, typing PIN, requesting the amount of money to withdraw, receiving the requested cash and the inserted card if the ATM system property is achieved, or receiving the card if it is not. Thus, the *abstract* may be expressed equivalently as formula below:

$$\forall\ o : Atom \mid (o \in Operations) \ => (o \in EnterCard) \ \vee \ (o \in TypePin) \ \vee$$

$$(o \in RequistCash) \ \vee \ (o \in ReceiveCashAndCard) \ \vee \ (o \in ReceiveCard)$$

.

For top-level type *ATM_Status*, the keyword *abstract* constrains that every element in the *ATM_Status* type is an element in one of its extensions *WaitingCard*, *WaitingPin*, *WaitingMoney*, *WaitingReceiveCashAndCard*, *WaitingReceiveCard*, and *Update* (Line 5) for six ATM statuses: waiting for entering card, waiting for typing PIN, waiting for selecting the amount of money, waiting for receiving card and money, waiting for just receiving card, and update if the ATM system property is achieved, then the balance will be updated. Thus, the *abstract* may be expressed equivalently as formula below:

$$\forall\, o : Atom \mid (o \in ATM\_Status) \ => (o \in WaitingCard) \vee (o \in WaitingPin) \vee$$

$$in(o, WaitingMoney)\ (o \in WaitingReceiveCashAndCard)$$

$$\vee\ (o \in WaitingReceiveCard)\ \vee\ (o \in Update)$$

Extension types *EnterCard*, *TypePin*, *RequistCash*, *ReceiveCashAndCard*, *ReceiveCard*, *WaitingCard*, *WaitingPin*, *WaitingMoney*, *WaitingReceiveCashAndCard*, *WaitingReceiveCard*, and *Update* are represented as singletons using the keyword *one* which means there are no more than two *Operations* or two *ATM_Status* occur together, otherwise the model will be inconsistent. Singleton sets can be and are often used in Alloy specifications as aliases of their unique element. The *extensions* may be expressed equivalently as seen below:

$$(EnterCard\ \subseteq\ Operations),\ (TypePin\ \subseteq\ Operations),$$

$$(RequistCash\ \subseteq\ Operations), (ReceiveCashAndCard\ \subseteq\ Operations),$$

$$(ReceiveCard\ \subseteq\ Operations),\ (WaitingCard\ \subseteq\ ATM\_Status),$$

$$(WaitingPin\ \subseteq\ ATM\_Status),\ (WaitingMoney\ \subseteq\ ATM\_Status),$$

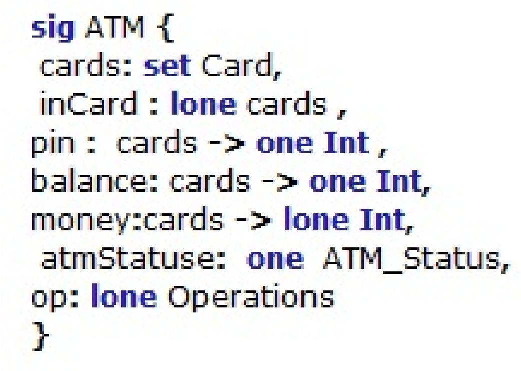
$$(WaitingReceiveCashAndCard\ \subseteq\ ATM\_Status),$$

$$(WaitingReceiveCard\ \subseteq\ ATM\_Status),\ (Update\ \subseteq\ ATM\_Status)$$

The top-level type *ATM* shows the relation declarations between all defined types and subtypes (Lines from 1 to 7). These relations will be described in detail in the next section.

- **Relation Declarations, and Multiplicities**

  In our model as seen in Appendix (A.2), relations are declared as the following signature fields.



```
sig ATM {
 cards: set Card,
 inCard : lone cards ,
pin :  cards -> one Int ,
balance: cards -> one Int,
money:cards -> lone Int,
 atmStatuse:  one  ATM_Status,
op: lone Operations
}
```

  - The field *cards* represents all the *Cards* associated with the ATM, and declares a binary relation of type ATM ->Card which maps each element of *ATM* to a *set* of elements of *Card*.

    The binary relation *cards* ⊆ ATM × Card. Lines (7 in Appendix A.1, and 1 in Appendix A.2) declare a relation *cards* with domain *ATM* and range *Card*. The declaration of *cards* contains the multiplicity annotation *set* which makes *cards* a function: a binary relation that associates every *ATM* with a set of *Card*. For every element atm in *ATM*, the keyword *set* before *Card* constrains the term atm.cards to be a *set*. The declaration of

$$cards\ in\ ATM - >\textbf{set}\ Cards$$

    can thus be expressed as:

$$\forall\ atm : ATM|\ atm.cards\ \subseteq\ (Cards)$$

    We can express the multiplicity constraints using formula as:

$$all\ atm : ATM|\ \textbf{set}\ atm.cards$$

The multiplicity keyword *set* makes the declaration hold when the relation *cards* is a function from *ATM* to *Card*.

– The field *inCard* represents the inserted cards in the *ATM*, and declares a binary relation of type ATM ->Card which maps each element of *ATM* to *lone* element of *ATM.cards*.

  The binary relation *inCard* $\subseteq$ ATM $\times$ Cards. Lines (7 in Appendix A.1, and 2 in Appendix A.2) declare a relation *inCard* with domain *ATM* and range *Card*. The declaration of *inCard* contains the multiplicity annotation *lone* which makes *inCard* a partial function: a binary relation that associates every *ATM* with at most one *ATM.cards*. For every element atm in *ATM*, the keyword *lone* before *atm.cards* constrains the term atm.inCard to be *lone*. The declaration

$$inCard\ in\ ATM - >\textbf{lone}\ cards$$

  can thus be expressed by:

$$\forall\ atm : ATM|\ atm.inCard\ \subseteq\ (atm.cards)$$

  We can express the multiplicity constraints using formula:

$$all\ atm : ATM|\ lone\ atm.inCard$$

  The multiplicity keyword *lone* makes the declaration hold when the relation *inCard* is a partial function from *ATM* to *atm.cards*. When a field of the same signature appears in another field's declaration, it is interpreted in the context of that signature. Field *cards* of the same signature *ATM* appears in another field's declaration *inCard*. Field *inCard* is added to capture that *lone* atm.cards might be *inCard*. This declaration makes every element atm in *ATM*, atm.inCard *subset* of at most one atm.cards.

– The field *pin* represents the *pin* numbers for the cards, and declares a ternary relation of type ATM ->Card ->Int which maps each element of *ATM* to an element in *atm.cards*; each element in *atm.cards* to *one* integer *Int*; each *Int* to an element in *atm.cards*; each element in *atm.cards* to an element in *ATM*.

The ternary relation *pin* ⊆ ATM × Cards × Int. Lines (7 in Appendix A.1, and 3 in Appendix A.2) declare a relation *pin* with domain *atm.cards.(atm.pin)* and range *Int*. The declaration of *pin* contains the multiplicity annotation *one* which makes *pin* a total function: a ternary relation that associates every every atm : ATM, and every c : Card included in atm.cards, the pair (atm, c) must be mapped to *one integer*. For every element atm in *ATM* and for every element c in *Card*, the keyword *one* before *Int* constrains the term atm.cards.(atm.pin) to be *one*. The declaration of

$$pin\ in\ ATM -> cards -> int$$

can thus be expressed by:

$$\forall\ atm : ATM|\ atm.pin\ \subseteq\ atm.cards\ \&\&$$

$$\forall\ atm : ATM,\ atm2 : atm.cards,\ pn : int | atm2.(atm.pin)\ \subseteq\ (\textbf{one}\ atm.pn)$$

We can express the multiplicity constraints using formula:

$$all\ atm : ATM|\ atm.pin\ \&\&\ all\ atm : ATM,\ atm2 : atm.cards | one\ atm2.(atm.pin)$$

The multiplicity keyword *one* makes the declaration hold when the relation *pin* is a total function from *atm.cards.(atm.pin)* to *Int*. Field *pin* is added to capture that *one* atm cards has one *pin* which is exactly one *Int*. This declaration makes every element atm in *ATM*, atm.pin a subset of atm.cards because field *cards* of the same signature *ATM* appears in another field's declaration *pin*.

– The field *balance* represents the *balance* value for the cards, and declares a ternary relation of type ATM ->Card ->Int which maps each element of *ATM* to an element in *atm.cards*; each element in *atm.cards* to *one* integer *Int*; each *Int* to an element in *atm.cards*; each element in *atm.cards* to an element in *ATM*.

The ternary relation *balance* ⊆ ATM × Cards × Int. Lines (7 in Appendix A.1, and 4 in Appendix A.2) declare a relation *balance* with domain *atm.cards.(atm.balance)* and range *Int*. The declaration of *balance* contains the multiplicity annotation *one* which makes *balance* a total function: a ternary relation that associates every every atm : ATM, and every c : Card included in atm.cards, the pair (atm, c) must be mapped to *one integer*. For every element atm in *ATM* and for every element c in *Card*, the keyword *one* before *Int* constrains the term atm.cards.(atm.balance) to be *one*. The declaration of

$$balance\ in\ ATM -> cards -> int$$

can thus be expressed by:

$$\forall\ atm : ATM|\ atm.balance\ \subseteq\ atm.cards\ \&\&$$

$$\forall\ atm : ATM,\ atm2 : atm.cards,\ m : int|atm2.(atm.balance)\ \subseteq\ (\mathbf{one}\ atm.m)$$

We can express the multiplicity constraints using formula:

$$all\ atm : ATM|\ atm.balance\ \&\&\ all\ atm : ATM,\ atm2 : atm.cards|$$

$$one\ atm2.(atm.balance)$$

The multiplicity keyword *one* makes the declaration hold when the relation *balance* is a total function from *atm.cards.(atm.balance)* to *Int*. Field *balance* is added to capture that *one* atm cards has one *balance* which is exactly one *Int*. This declaration makes every element atm in *ATM*, atm.balance a subset of atm.cards because field *cards* of the same signature *ATM* appears in another field's declaration *balance*.

– The field *money* represents the *money* value that is required to be withdrawn, and declares a ternary relation of type ATM ->Card ->Int which maps each element of *ATM* to an element in *atm.cards*; each element in *atm.cards* to *lone* integer *Int*; each *Int* to an element in *atm.cards*; each element in *atm.cards* to an element in *ATM*.

  The ternary relation *money* $\subseteq$ ATM $\times$ Cards $\times$ Int. Lines (7 in Appendix A.1, and 5 in Appendix A.2) declare a relation *money* with domain *atm.cards.(atm.money)* and range *Int*. The declaration of *balance* contains the multiplicity annotation *lone* which makes *balance* a partial function: a ternary relation that associates every every atm : ATM, and every c : Card included in atm.cards, the pair (atm, c) must be mapped to *lone* (at most one) *integer*. For every element atm in *ATM* and for every element c in *Card*, the keyword *lone* before *Int* constraints the term atm.cards.(atm.money) to be *lone*. The declaration of

  $$money\ in\ ATM -> cards -> int$$

  can thus be expressed by:

  $$\forall\ atm : ATM|\ atm.money\ \subseteq\ atm.cards\ \&\&$$

  $$\forall\, atm : ATM,\ atm2 : atm.cards,\ m : int|atm2.(atm.money)\ \subseteq\ (\textbf{lone}\, atm.m)$$

  We can express the multiplicity constrains using formula:

  $$all\ atm : ATM|\ atm.money\ \&\&\ all\ atm : ATM,\ atm2 : atm.cards|$$

  $$lone\ atm2.(atm.money)$$

  The multiplicity keyword *lone* makes the declaration hold when the relation *money* is a partial function from *atm.cards.(atm.money)* to *Int*. Field *money* is added to capture that *lone* atm cards has lone *balance* which is at most one *Int*. This declaration makes every element atm in *ATM*, atm.money a subset of atm.cards because field *cards* of the same signature *ATM* appears in another field's declaration *money*.

- The field *atmStatuse* represents all the statuses in the ATM, and declares a binary relation of type ATM ->ATM_Status which maps each element of *ATM* to *one* element of *ATM_Status*.

  The binary relation *atmStatuse* $\subseteq$ ATM $\times$ ATM_Status. Lines (7 in Appendix A.1, and 6 in Appendix A.2) declare a relation *atmStatuse* with domain *ATM* and range *ATM_Status*. The declaration of *atmStatuse* contains the multiplicity annotation *one* which makes *atmStatuse* a total function: a binary relation that associates every *ATM* with exactly *one* of *ATM_Status*. For every element atm in *ATM*, the keyword *one* before *ATM_Status* constrains the term atm.atmStatuse to be *one*. The declaration of

  $$atmStatuse\ in\ ATM -> ATM\_Status$$

  can thus be expressed by:

  $$\forall\ atm : ATM |\ atm.atmStatuse\ \subseteq\ (\textbf{one}\ ATM\_Status)$$

  We can express the multiplicity constrains using formula:

  $$all\ atm : ATM |\ \textbf{one}\ atm.atmStatuse$$

  The multiplicity keyword *one* makes the declaration hold when the relation *atmStatuse* a total function from *ATM* to *ATM_Status*.

- The field *op* represents all the operations that take place in the ATM by the customer, and declares a binary relation of type ATM ->Operations which maps each element of *ATM* to *lone* element of *ATM_Status*.

  The binary relation *op* $\subseteq$ ATM $\times$ Operations. Lines (7 in Appendix A.1, and 7 in Appendix A.2) declare a relation *op* with domain *ATM* and range *Operations*. The declaration of *op* contains the multiplicity annotation *lone* which makes *op* a partial function: a binary relation that associates every *ATM* with at most *one* of *ATM_Status*. For every element atm in *ATM*, the keyword *lone* before *Operations* constrains the term atm.op to be *lone*. The declaration of

$$op\ in\ ATM -> Operations$$

can thus be expressed by:

$$\forall\ atm : ATM|\ atm.op\ \subseteq\ (\textbf{lone}\ Operations)$$

We can express the multiplicity constraints using formula:

$$all\ atm : ATM|\ \textbf{lone}\ atm.op$$

The multiplicity keyword *lone* makes the declaration hold when the relation *op* is a partial function from *ATM* to *Operations*.

Figure 3.3 shows the corresponding structure of the Alloy ATM system specification. It shows that all ATM statuses extends ATM_Status, and all operations extends Operations. Also, it illustrates the relations between entities and the multiplicities that restrict each relation. As seen, the relation between ATM and Operations is zero to one. The relation between Cards and ATM is zero to many. The relation between cards in ATM and pin, balance is one to one. The relation between cards in ATM and money is zero to one. The relation between cards in ATM and atm status is one to one. The relation between inserted Card and cards in ATM is zero to one.

In addition to the above implicit constraints, the specification introduces two explicit general constraints (properties) as facts written as logical formulas.

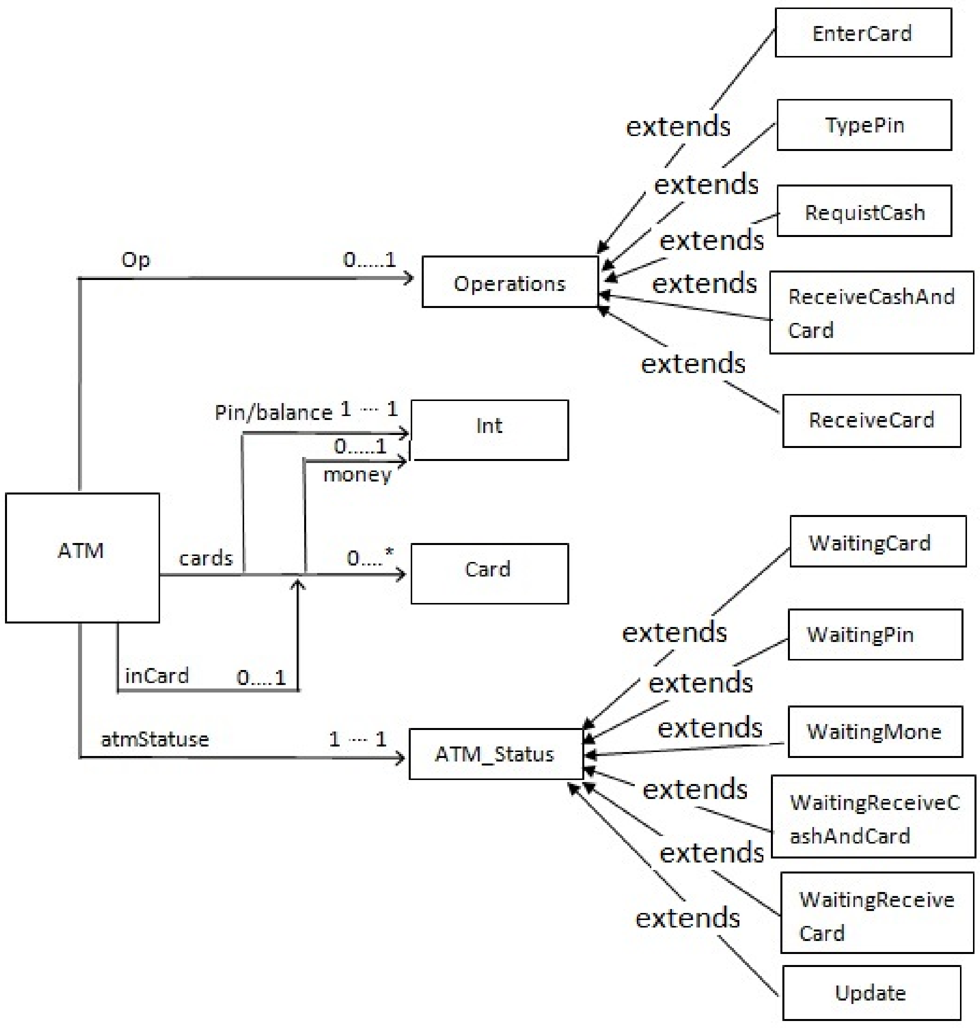


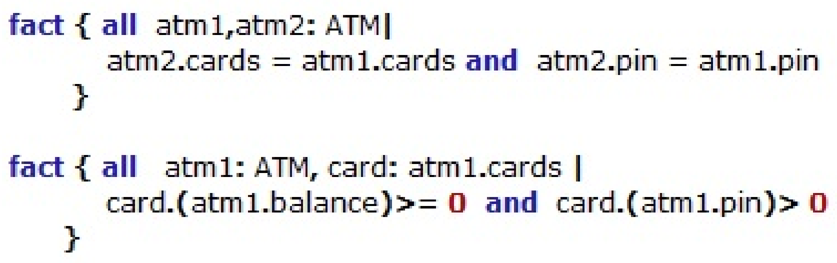
Figure 3.3: Structure Class Diagram of The Alloy ATM System Specification

- **Fact**

As seen in Appendix (A.3), the facts (Lines 1 and 2) represent constraints that are written as logical formulas and are assumed to hold. The first fact constrains that all ATM cards with its corresponding PIN should be identified in any ATM machine (atm1 and atm2) (Line 1). The second fact constrains that the balance and PIN of cards that interact with the ATM should not be less than zero (positive) (Line 2). Thus, the first and the second facts should hold, i.e always true.

- **Predicates**

In Alloy, operations are specified using predicates. A predicate is a logical formula with declaration parameters. A predicate describes a set of statuses and

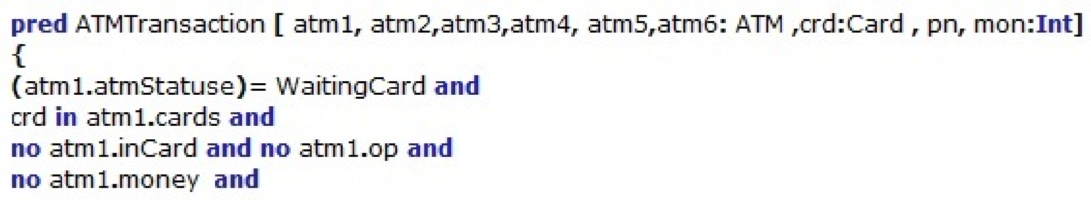


```
fact { all  atm1,atm2: ATM|
         atm2.cards = atm1.cards and  atm2.pin = atm1.pin
     }

fact { all   atm1: ATM, card: atm1.cards |
         card.(atm1.balance)>= 0  and  card.(atm1.pin)> 0
     }
```

transitions, by using constraints among signatures and their fields. Behavioural properties of the protocol can be expressed in terms of logical predicates which can be checked by the Alloy Analyser. In this formal specification, consistency of different statuses of protocol can be checked and generate instances for a given specification that satisfy this predicate with its restrictions on each status.

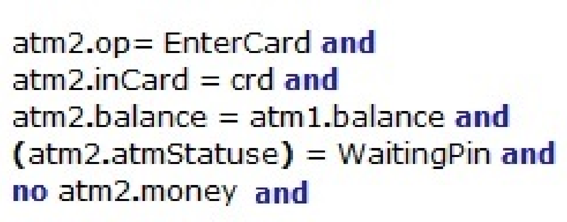
In our model as seen in Appendix (A.4), predicate *ATMTransaction* controls six statuses atm1, atm2, atm3, atm4, atm5, and atm6, crd for Card, pn,mon for Int (Line 1). First of all, we need to illustrate the main point which is, each *atmStatuse* expresses the previous status that takes place before the next operation takes place. Every *atmStatus* is constrained to be *one*. Furthermore, Every *Operations* constrained to be *lone*.

The first status (**atm1**), represents the status to be *WaitingCard* for the first *operation* (Line 2). This status also defines a *Card* element (crd) to be subset of the set of *cards* that belonged to *ATM* (Line 3). Moreover, in the first status, there is no card in use (entered), no operation, and no withdrawing money take place yet (Line 4).



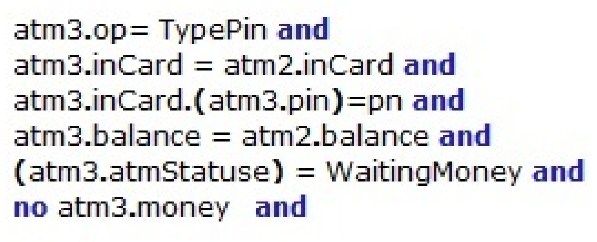
```
pred ATMTransaction [ atm1, atm2,atm3,atm4, atm5,atm6: ATM ,crd:Card , pn, mon:Int]
{
(atm1.atmStatuse)= WaitingCard and
crd in atm1.cards and
no atm1.inCard and no atm1.op and
no atm1.money  and
```

The second status (**atm2**), is responsible for inserting a card. The first operation that takes place is *EnterCard* (Line 5). In this status the inserted card is equal to the already defined *Card* element (crd) which is a subset of the set of *cards* that belonged to *ATM* (Line 6). The ATM is now in the *WaitingPin* status for the next *operation* and no withdrawing money takes place yet (Line 8). The balance has not been changed yet (Line 7).



```
atm2.op= EnterCard and
atm2.inCard = crd and
atm2.balance = atm1.balance and
(atm2.atmStatuse) = WaitingPin and
no atm2.money  and
```

The third status (**atm3**), is responsible for typing the pin for the inserted card *inCard*. The operation that takes place is *TypePin* (Line 9). In this status the inserted card in the current status should be equal to the inserted card in the previous status (Line 10) and typing the PIN for the inserted card is integer (*pn*)(Line 11). The ATM is now in the *WaitingMoney* status for the next *operation* and no withdrawing money takes place yet (Line 13). The balance has not been changed yet (Line 12).



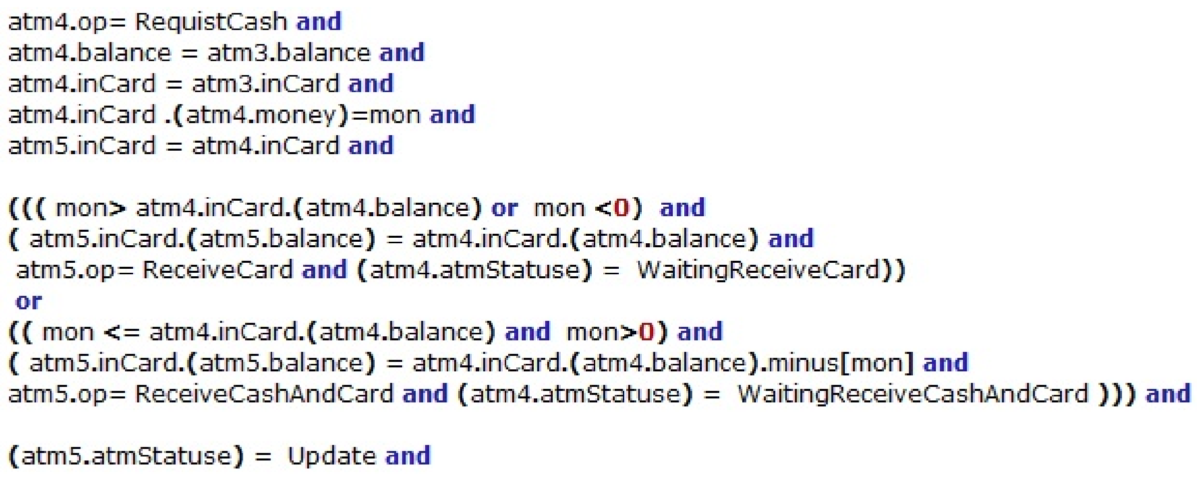
```
atm3.op= TypePin and
atm3.inCard = atm2.inCard and
atm3.inCard.(atm3.pin)=pn and
atm3.balance = atm2.balance and
(atm3.atmStatuse) = WaitingMoney and
no atm3.money   and
```

The fourth status (**atm4**), is responsible for requiring money for the inserted card. The operation that takes place is *RequistCash* (Line 14). In this status the inserted card in the current status should be equal to the inserted card in the previous status and there is withdrawing money (mon) as integer takes place in this status (Line 16). The balance has not been changed yet (Line 15). However, this status shows the procedure of requesting money, which passes to one of two processes.

In the first process, if the money is greater than the balance of the inserted card *inCard* or the money is less than zero (Line 18), then the balance has not been changed and remains the same (Line 19), allowing progression to the *operation ReceiveCard* in the next status (**atm5**), and the ATM is now in the *WaitingReceiveCard* status (Line 20).

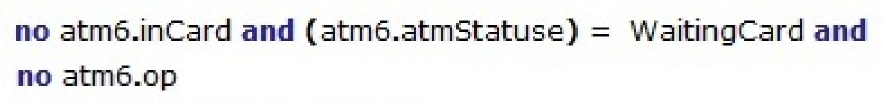
However, in the second process, if the money is less than or equal to the balance of the inserted card *inCard* and the money is greater than zero (Line 22), then the balance has been changed and decreased by the requested amount of money (mon) to withdraw (Line 23), allowing to the *operation ReceiveCashAndCard* in the next status (**atm5**) (Line 24) and the ATM is now in the *WaitingReceiveCashAndCard* status (Line 25).

The fifth status (**atm5**), is responsible for updating for the new ATM procedure. The operation that takes place is either *ReceiveCard* (Line 20) if the first process occurred, or *ReceiveCashAndCard* (Line 24) if the other (second) process occurred. It took place after the fourth status, with either *WaitingReceiveCashAndCard* or *WaitingReceiveCard*. The ATM is now in the *Update* status applied the current *operation* (Line 26).



```
atm4.op= RequistCash and
atm4.balance = atm3.balance and
atm4.inCard = atm3.inCard and
atm4.inCard .(atm4.money)=mon and
atm5.inCard = atm4.inCard and

((( mon> atm4.inCard.(atm4.balance) or  mon <0)  and
( atm5.inCard.(atm5.balance) = atm4.inCard.(atm4.balance) and
 atm5.op= ReceiveCard and (atm4.atmStatuse) =  WaitingReceiveCard))
 or
(( mon <= atm4.inCard.(atm4.balance) and  mon>0) and
( atm5.inCard.(atm5.balance) = atm4.inCard.(atm4.balance).minus[mon] and
atm5.op= ReceiveCashAndCard and (atm4.atmStatuse) =  WaitingReceiveCashAndCard ))) and

(atm5.atmStatuse) =  Update and
```

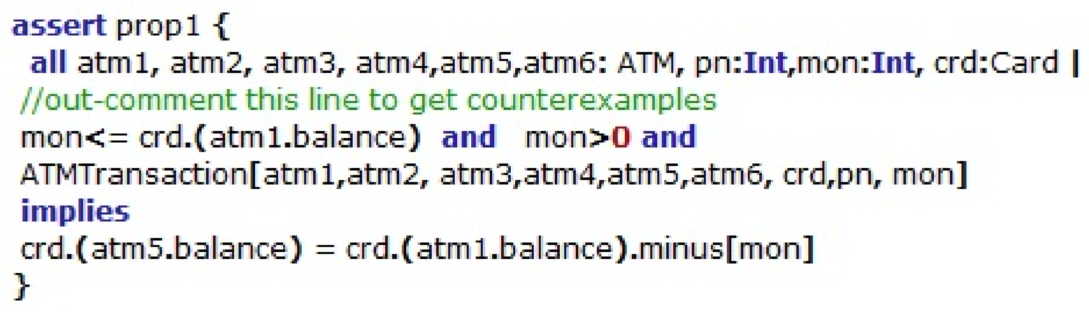
The sixth (last) status (**atm6**), is responsible for waiting for the new ATM procedure. There is no card in use (entered) and no operation takes place. The status returns to the first status, *WaitingCard* preparing for the first *operation* (Line 27). The last status is same as the first one.



```
no atm6.inCard and (atm6.atmStatuse) =  WaitingCard and
no atm6.op
```

- Assertion

In our model as seen in Appendix (A.5), assertions are the intended properties and are used to check specifications. As seen in the assertion , if *ATMTransaction* holds for *ATM* atm1,atm2,atm3,atm4,atm5,atm6, *Card* crd, *Int* pn,mon (Line 1), and the required amount of money is less than or equal to the main balance in the first status (atm1) for the inserted card (crd) and the money is greater than zero (Line 2), that implies that the main balance in the first status is updated and changed in the fifth status and decreased by the required money (Line 3).



```
assert prop1 {
 all atm1, atm2, atm3, atm4,atm5,atm6: ATM, pn:Int,mon:Int, crd:Card |
 //out-comment this line to get counterexamples
 mon<= crd.(atm1.balance)  and   mon>0 and
 ATMTransaction[atm1,atm2, atm3,atm4,atm5,atm6, crd,pn, mon]
 implies
 crd.(atm5.balance) = crd.(atm1.balance).minus[mon]
}
```

3.8 Results

In this section, we present the results we achieved from modelling, analysing, and checking the three properties of the ATM system using Alloy Analyser, and bounded SAT solver. The **first property** is all ATM cards with corresponding PINs should be identified in any ATM machine. The **second property** is the balance and PIN of cards that interact with the ATM should not be less than zero (positive). The **third property** is the required amount of money should be less than or equal to the available balance, and greater than zero for the used card.

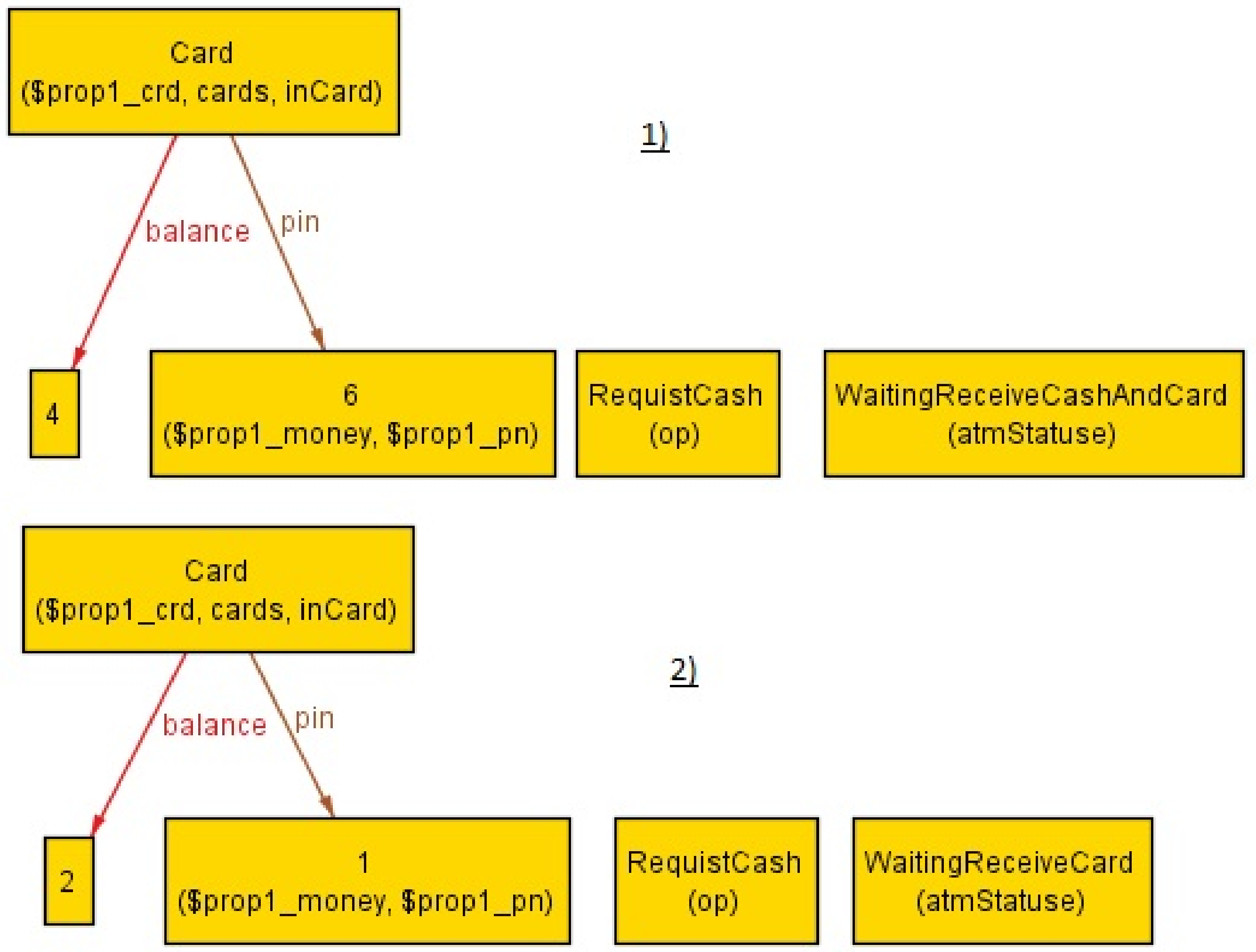
In Appendix (A), part (A.6), we used two commands: **check** (Line 1) and **run** (Line 2). **check** checks the satisfiability of the model, while **run** checks the consistency of the model.

Before checking the satisfiability of the model, we need first to check its consistency. If the model is inconsistent, the analyser cannot work efficiently for detecting a counterexample.

The version of Alloy Analyser (4.1.10) that we have used works with many state-of-the-art solvers such as BirkMin [65], MiniSat [55], ZChaff [115], and SAT4J which is the only SAT solver we tried. The **run** command runs the SAT4J solver. The command asks the analyser to search for instances to visualize them. These instances assign sets and relations with their sizes is limited to be 1 atom for each signature except ATM 6 atoms. The visualised instances for the ATM model are acceptable. They correspond to the declarations of the fields and signatures and satisfies the predicate and the Alloy model together which means the model is consistent.

The **check** command searches for a counterexample showing an execution path that caused an error if one exists. The command looks for an instance that violates the assertion. This analysis is implemented with respect to the bounded scope of 1 atom for each signature except ATM 6 atoms. Only a finite number of elements for each type is taken into account. Therefore, the absence of an instance does not include checking satisfiability.

The Alloy Analyser spent 0.233s to find a counterexample in the limited scope. That means the assertion *prop1* in ATM model does not hold because the third property (the required amount of money should be less than or equal to the available balance, and greater than zero for the used card) has not been restricted. The detected counterexample is provided to the user by interpreting the SAT valuation as a solution to the original problem, as seen in Figure 3.4.



Card
($prop1_crd, cards, inCard)
1)
balance
pin
4
6
($prop1_money, $prop1_pn)
RequistCash
(op)
WaitingReceiveCashAndCard
(atmStatuse)
Card
($prop1_crd, cards, inCard)
2)
balance
pin
2
1
($prop1_money, $prop1_pn)
RequistCash
(op)
WaitingReceiveCard
(atmStatuse)

Figure 3.4: Two Instances of The Generating Two Counterexamples: (1) Shows ATM Provides The Amount of Money When The Amount of Money Is Greater Than The Balance, and (2) Shows The ATM Does Not Provide The Amount of Money When The Amount of Money Is Less Than The Balance.

In Figure 3.4, (1) and (2), the fourth status show two counterexamples explained as a visualization. They conflict with the assumption of the third property (the required amount of money should be less than or equal to the available balance, and greater than zero for the used card).

The first counterexample in (1) shows that: in the fourth status when the card is in the ATM machine and the ATM operation is *RequestCash*, we see that the requested amount of money is "6" while the available balance is "4". We noticed that the requested amount of money is greater than the available balance. So, we assumed that the ATM status is *WaitingReceiveCard*. However, we did not see what we assumed. We saw that an error took place and the ATM status is *WaitingReceiveCashAndCard* which means the ATM provides the required amount of money even it is greater than the available balance. Consequently, the visualised instance reflects the assumption and as a result the **security** is broken.

The second counterexample (2) shows that: in the fourth status when the card is in the ATM machine and the ATM operation is *RequistCash*, we see that the requested amount of money is "1" while the available balance is "2". We noticed that the requested amount of money is less than the available balance. So, we assumed that the ATM status is *WaitingReceiveCashAndCard*. However, we did not see what we assumed. We saw that an error took place and the ATM status is *WaitingReceiveCard* which means ATM does not provide the required amount of money even it is less than the available balance. Consequently, the visualised instance reflects the assumption and as a result the **availability** is broken.

So, based on Figure 3.4 the counterexamples mean the third property did not hold. Therefore, we need to restrict the assertion to achieve the third property. The restriction allows the ATM to provide the amount of money if the amount of money is less than or equal to the balance and the amount of money is greater than "0" as seen in Appendix (A) part (A.5) line (2). Otherwise, no amount of money is provided and the card is rejected.

After adding the third property this analysis is implemented with respect to a bounded scope of 1 atom for each signature except ATM 6 atoms. Alloy Analyser spent 0.187s checking the three properties to generate counterexamples with respect the finite scope.

The results is that no counterexample is found and it is acceptable. That means, the model has satisfied the assertion. The model allows us to visualise the model states with the status changes as seen in Figures 3.5, 3.6, 3.7 indicated by $ before the variables name in the predicate. The figures conclude that the properties that have been analysed hold in the model within the provided scopes and also when the scope is increased.

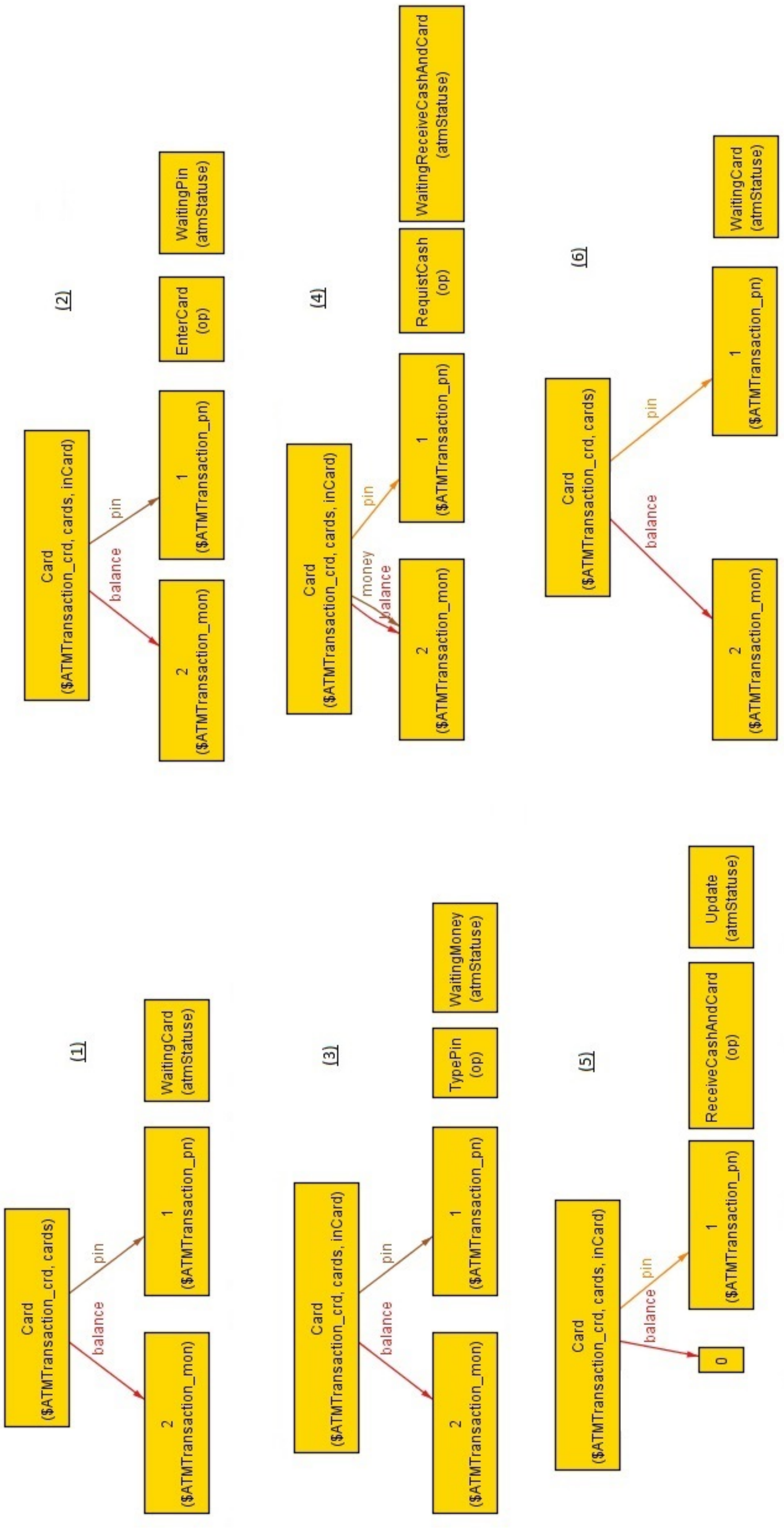


Figure 3.5: An Instance of The The ATM Model (Amount = Balance)

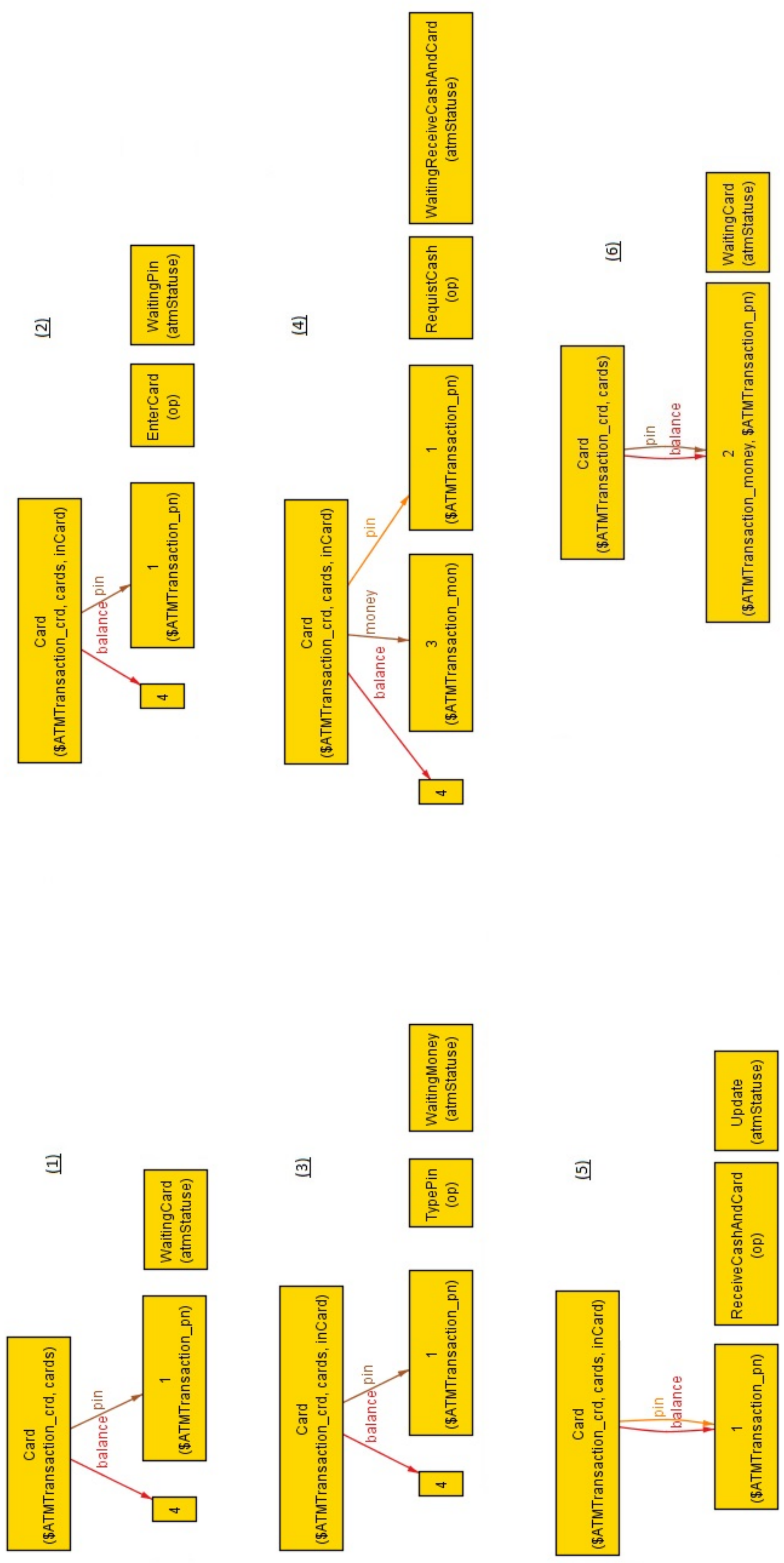


Figure 3.6: An Instance of The ATM Model (Amount <Balance)

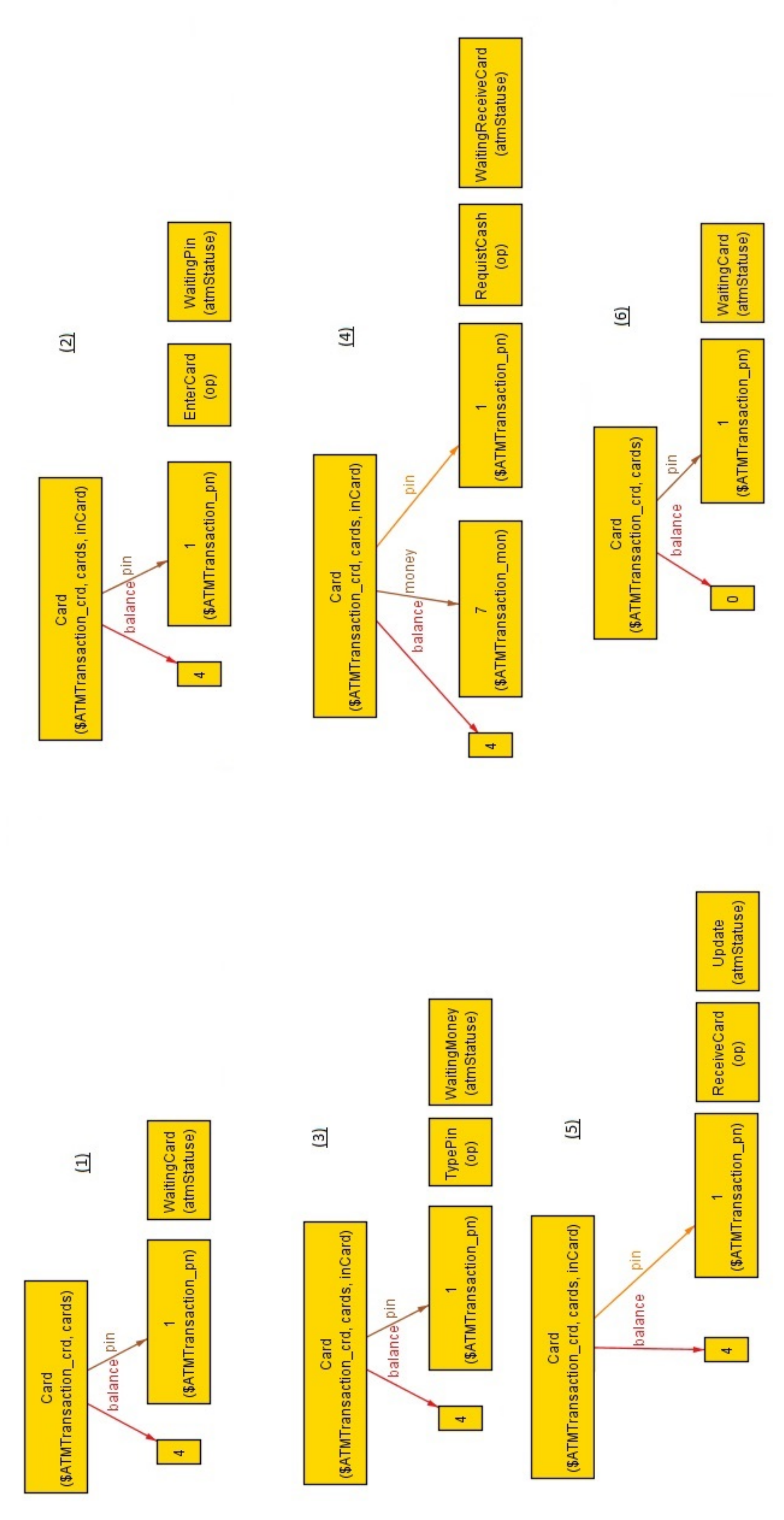


Figure 3.7: An Instance of The ATM Model (Amount >Balance)

Figures 3.5, 3.6, 3.7 achieved and illustrate the ATM properties. They show numbers (1 ,2, 3, 4, 5, 6) which mean sequential statuses. Each number indicates the ATM status which includes its status and operation. The status number is shown to the right of each diagram. Figures 3.5, 3.6, 3.7 show the ATM properties when the required amount of money is equal to balance, less than the balance, and greater than the balance respectively. They share the first status (1), second status (2), third status (3), and sixth status (6). However they differ in the fourth status (4), and fifth status (5).

In the first status (1) the ATM status is *WaitingCard* and there is no operation yet. We see that there is no card in ATM yet, the balance is "2" in Figure 3.5 but "4" in Figures 3.6, 3.7. We restrict the PIN number to be always true to hold the second property.

The next status is (2). In this status we see that the operation is *EnterCard* and the card is in the ATM (means the transaction is going on). The ATM status is *WaitingPin.* The balance and PIN number remain the same.

The next status is (3). In this status we see that the operation is *TypePin* and the card is in the ATM. The ATM status is *WaitingMoney.* The balance and PIN number remain the same.

The next status is (4). In this status we see that the operation is *RequestCash* and the card is in the ATM. In Figure 3.5, the ATM status is *WaitingReceive-CashAndCard.* We see the required amount of money equals the balance "2"2. In Figure 3.6, the ATM status is *WaitingReceiveCashAndCard.* We see the required amount of money "3" is less than the balance "4". In Figure 3.7, the ATM status is *WaitingReceiveCard.* We see the required amount of money "7" is greater than the balance 4. The balance and PIN number remain the same.

The next status is (5). In this status we see that the operation is *Receive-CashAndCard* in Figures 3.5, 3.6, while the operation is *ReceiveCard* in Figure 3.7 and the card is in the ATM. The ATM status is *Update*. We see the balance is updated, changed and decreased by the required amount of money to become "0" in Figure 3.5 and "1" in Figure 3.6, but it remains the same in Figure 3.7. The PIN number remains the same.

The last status is (6). In this status we see that there is no operation and the card is out the ATM. The ATM status is *WaitingCard*: back to first status waiting for a new customer.

We will discuss these results at the end of the next chapter, after looking at the foundation of Z3.

Chapter 4

Foundations: Z3 SMT Solver

This chapter provides detailed information and the foundation of the framework of the Z3 SMT solver, again using an ATM system as an example. Because the Alloy Analyser cannot prove the validity of an assertion [64], we need to use the Z3 SMT solver's unbounded verification to prove the properties correct with confidence.

Prove the properties correct using Z3 required following as same stages that have been used in Alloy, except for using scopes as Z3 is unbounded. These stages are: determining the properties to be achieved in the system; determining the main entities that interact in the system; determining and constraining how the entities are related to each other; predicate how a system will behave and describe how the state changes; restrict facts that hold for the properties of the system; and build a formula as an assertion to check the specification of the model.

Z3 has no syntax for defining signatures, build relations, or multiplicity keywords the same as Alloy. Z3 uses the SMT2 language which is a standard constraints language.

4.1 SMT Solver

SMT stands for "`SAT Modulo Theories`". SAT is "Boolean satisfiability". SAT is based on Boolean formulas: Boolean variables combined with AND ($\wedge$), OR ($\vee$) and NOT ($\neg$). SMT is a popular extension to SAT. With the extension, the SMT solver adds theories to the basic logic SAT. This makes encoding many problems much easier with SMT than with just SAT, and it can use high-level knowledge about the constraints to implement particular theories more efficiently. The language of SMT solvers is first-order which is undecidable and implies that the unbounded Z3 SMT solver is undecidable [1].

4.1.1 Z3

Z3 is a novel and efficient SMT Solver available free from Microsoft Research. Z3 is a low level tool. It can be utilized for verifying logical formulas' satisfiability with quantifiers. The main goal for Z3 is to check the satisfiability of a provided formula i.e, is there a model for the negation of the provided formula or not. It proves the correctness of the negation of the formula of the property [40].

SMT benchmark consists of sorts and functions declarations, and a set of SMT formulas which are the assertions given by the user [63]. The commands that are used in Z3 are *declare-const* which declares a constant of a given types; *declare-fun* which declares a function; *assert* which adds a formula into the Z3 internal stack; and *check-sat* which asks the Z3 SMT solver to *check* if the negation of the conjunction of the provided assertions is satisfiable or not.

The Z3 SMT solver returns one of three types of results: *UNSAT*, *SAT*, and *UNKNOWN* [63]. If the command outputs *UNSAT*, the *negation* of the property has been proven correct and the set of formulas in the Z3 stack is satisfiable which means there is an interpretation (for the user declared constants and functions) that makes all asserted formulas true [63]. If the command outputs *SAT*, a valid counterexample has been found and *get-model* can be used to retrieve an interpretation that makes all formulas on the Z3 internal stack true. If the command outputs *unknown*, the property may or may not be valid so here Z3 does not guarantee a complete analysis [63].

The formulas in the Z3 SMT solver are formulated in typed first-order logic with equality and type system the same as in Alloy. It has no explicit sub-typing like Alloy. The satisfiability of the formulas in the Z3 SMT solver is considered with respect to a set of logical background theories, which commonly constrain the interpretation of symbols utilized in the formula [114]. Moreover, to create sorts like Bool, Int, and Real, the language enables the declaration of novel uninterpreted sorts and functions [138].

The Alloy language is undecidable because it uses first-order logic and first-order logic is undecidable. Thus it is necessary to do some adaptations. In order to make it decidable, Alloy utilizes the first-order logic with a finite scope n. This finite scope allows a satisfying instance in limited scope with no more than n atoms of each type. Limiting scope size when creating a model is essential to avoid exponential explosion. However, because the analysis is restricted in scope, there is no guarantee that a counterexample would not be found with a larger scope. While the decidability for a finite scope is achieved, the complete analysis is lost [135]. In this case, using Z3 SMT allows quantifiers over free sorts, and thus is undecidable. However, Z3 does not guarantee a complete analysis: it may output a counterexample preceded by the keyword unknown, which implies the property may or may not be valid, or time out.

4.1.2 The SMT Language

We translate the specification of the ATM from Alloy into Z3 FOL such that if there is a counterexample in Alloy in finite scope, there is supposed to be a counterexample in Z3 in infinite scopes and vice versa. The formulas that we are going to generate utilize the quantified theories of free sorts, linear integer arithmetic, and uninterpreted functions and constant.

4.1.2.1 Declarations

The logic implicit in SMT enables two kinds of declaration: to declare new sorts (types) utilizing the *declare-sort* command, and to declare functions using *declare-fun*. Functions are considered as the basic building blocks of SMT formulas.

The expression *(declare-sort X 0)* declares a novel simple uninterpreted top-level sort named X. The expression

$$(declare - fun\ f\ (X1\ X2)\ X3)$$

declares the novel uninterpreted total function

$$F:\ X1\ \times\ X2\ ->X3$$

The command

$$(declare - fun\ f\ (X_1,\ ....,\ X_{n-1})\ X_n)$$

declares

$$f:\ X_1\ \times\ \ldots\ \times X_{n-1}\ ->X_n$$

All functions are *total* and so they are defined for all elements of their domain. Constants are functions but without arguments, i.e. a constant v of type X is declared as

$$(declare - fun\ v\ ()\ X)$$

4.1.2.2 Assertions and Quantifiers

The command *(assert f)* is used to assert a formula f in the logical context. Basic formulas are function applications and easy to be combined utilizing the boolean operators such as *and*, *or*, *not*, and *=>(implies)*.

Universal quantifiers are indicated by

$$(forall\ (a_1\ A_1)\ \ldots\ (a_n\ A_n)\ f)$$

while *existential* quantifiers are indicated by

$$(exists\ (a_1\ A_1)\ \ldots\ (a_n\ A_n)\ f)$$

4.2 An Automated Teller Machine (ATM) Example in Z3

In this section, we translate the specification of the ATM from Alloy into a satisfiability-equivalent SMT problem using Z3 SMT logic (FOL). This would be solved by an SMT solver such that if there is a counterexample in Alloy in finite scopes, it supposed to be a counterexample in Z3 in infinite scopes and vice versa. Our translation is manual. We choose this method as it is easy and we can trust it as correct [63, 62] proved this methodology in general and we simplified it to be easier to learn and use.

4.2.1 Type and Subtype Declarations

In Appendix (B) we give full details of the Z3 models with annotations to show the equivalent Alloy.

As seen in Appendix (B), the hierarchical type system is translated implicitly. However, because the SMT language does not support subtypes, we use uninterpreted membership functions to enforce type hierarchy declarations. Consequently, top-level types are translated to the uninterpreted sorts, while the top-level (super-type) of a type is translated to uninterpreted membership function *isType* to indicate which elements of the super-type belongs to the type. It is not necessarily to declare the membership functions of top-level types, but we declared them to determine the semantic of the subtype.

As seen in Appendix (B.1), top-level types: *Operations*; *ATM_Status*; *Card*; and *ATM* are declared as uninterpreted sorts (Lines from 1 to 4). The membership functions in Appendix (B.2): *isEnterCard*; *isTypePin*; *isRequistCash*; *isReceive-CashAndCard*; *isReceiveCard*; *isWaitingCard*; *isWaitingPin*; *isWaitingMoney*; *isWaitingReceiveCashAndCard*; *isWaitingReceiveCard*; and *isUpdate* (Lines from 8 to 18) are declared to specify the semantics of subtypes: *EnterCard*; *Type-Pin*; *RequistCash*; *ReceiveCashAndCard*; *ReceiveCard*; *WaitingCard*; *WaitingPin*; *WaitingMoney*; *WaitingReceiveCashAndCard*; *WaitingReceiveCard*; and *Update.* i.e all membership functions are disjoint subsets of the declared sort *Operations* (Lines from 8 to 12), and all membership functions are disjoint subsets of the declared sort *ATM_Status* (Lines from 13 to 18).

4.2.2 Properties Of The Sub-signatures

As seen in Appendix (B.5), we adjust the return types of the "oneOf " functions/constants by specifying each return value to be of type *Operation* (Lines 1-5) and *ATM_Status* (Lines 6-11). For example: Line (1) calls function *oneOf_EnterCard* in Line (2) in Appendix (B.3) to return *one* Operation type *isEnterCard* in Line (8) Appendix (B.2) which already specified as a return type in Line (1) Appendix (B.5).

Sub-signatures (Lines from 2 to 6) Appendix (B.3) declare functions of the property *some* for each sub signatures *EnterCard*, *TypePin*, *RequistCash*, *ReceiveCashAndCard*, and *ReceiveCard* of the super signature *Operations* because *Operations* has at least one element of the sub signatures. The functions restrict the super signature to have at least *one* element in each operation. Sub-signatures (Lines 8-13) Appendix (B.3) declare functions of the property *some* for each sub signatures *WaitingCard*, *WaitingPin*, *WaitingMoney*, *WaitingReceiveCashAndCard*, *WaitingReceiveCard*, and *Update* of the super signature *ATM_Status* because *ATM_Status* has exactly one element of the sub signatures. The functions restrict that the super signature *ATM_Status* and *Operations* have at least *one* element in each ATM_Status. In (Lines from 2 to 6) and (Lines from 9 to 14) Appendix (B.6) we need to assert the *lone* property of the previous sub signatures because *Operations* and *ATM_Status* have at most one element of the sub signatures. The assertion (Lines from 2 to 6) and (Lines from 9 to 14) Appendix (B.6) is expressed in the formulas AA and BB below respectively :

Formula AA:

$\forall$ *o1, o2: operation.(o1* $\in$ *isEnterCard* $\wedge$ *o2* $\in$ *isEnterCard) =>o1 = o2*
$\forall$ *o1, o2: operation.(o1* $\in$ *isTypePin* $\wedge$ *o2* $\in$ *isTypePin) =>o1 = o2*
$\forall$ *o1, o2: operation.(o1* $\in$ *isRequistCash* $\wedge$ *o2* $\in$ *isRequistCash) =>o1 = o2*
$\forall$ *o1, o2: operation.(o1* $\in$ *isReceiveCashAndCard* $\wedge$ *o2* $\in$ *isReceiveCashAndCard) =>o1 = o2*
$\forall$ *o1, o2: operation.(o1* $\in$ *isReceiveCard* $\wedge$ *o2* $\in$ *isReceiveCard) =>o1 = o2*

The formula AA specifies constraints that for each *operation*, there is at most only one corresponding *operation*: if there exist two *operations* belonging to *isEnterCard* for example, then these two *operations* should be equal. i.e we restrict the characteristics of the multiplicity *lone operation* of type *isEnterCard* for each status to avoid the inconsistency.

Formula BB:

$\forall$ *a1, a2: ATM_Status.(a1* $\in$ *isWaitingCard* $\wedge$ *a2* $\in$ *isWaitingCard) =>a1 = a2*
$\forall$ *a1, a2: ATM_Status.(a1* $\in$ *isWaitingPin* $\wedge$ *a2* $\in$ *isWaitingPin) =>a1 = a2*
$\forall$ *a1, a2: ATM_Status.(a1* $\in$ *isWaitingMoney* $\wedge$ *a2* $\in$ *isWaitingMoney) =>a1 = a2*
$\forall$ *a1, a2: ATM_Status.(a1* $\in$ *isWaitingReceiveCashAndCard* $\wedge$ *a2* $\in$*)*
isWaitingReceiveCashAndCard =>a1 = a2
$\forall$ *a1, a2: ATM_Status.(a1* $\in$ *isWaitingReceiveCard* $\wedge$ *a2* $\in$ *isWaitingReceiveCard)*
=>a1 = a2
$\forall$ *a1, a2: ATM_Status.(a1* $\in$ *isUpdate* $\wedge$ *a2* $\in$ *isUpdate) =>a1 = a2*

The formula BB specifies constraints that for each *ATM_Status*, there is at most only one corresponding *ATM_Status*: if there exist two *ATM_Statuses* belonging to *isWaitingCard* for example, then these two *ATM_Status* should be equal. i.e we restrict the characteristics of the multiplicity *lone ATM_Status* of type *isWaitingCard* for each status to avoid the inconsistency.

4.2.3 Abstraction

As seen in Appendix (B.7), abstract types are the union of their subtypes. Thus abstract types constrain every element of type to belong to one of its extending subtypes. Lines (1 and 15) are expressed in the formulas CC and DD below respectively:

Formula CC:

$\forall$ *o:operation.*$\neg$(*o* $\in$ *isEnterCard* $\wedge$ *o* $\in$ *isTypePin* $\wedge$
o $\in$ *isRequistCash* $\wedge$ *o* $\in$ *isReceiveCashAndCard* $\wedge$
o $\in$ *isReceiveCard*) $\wedge$
$\neg$(*o* $\in$ *isTypePin* $\wedge$ *o* $\in$ *isRequistCash* $\wedge$
o $\in$ *isReceiveCashAndCard* $\wedge$ *o* $\in$ *isReceiveCard* $\wedge$
o $\in$ *isReceiveCard*) $\wedge$
$\neg$(*o* $\in$ *isRequistCash* $\wedge$ *o* $\in$ *isReceiveCashAndCard* $\wedge$
o $\in$ *isReceiveCard* $\wedge$ *o* $\in$ *isReceiveCashAndCard*)

The formula CC specifies constraints that for each *operation* there is only one corresponding *operation*, and this *operation* is either *isEnterCard*; *isTypePin*; *isRequistCash*; *isReceiveCashAndCard*; or *isReceiveCard* to avoid inconsistency. No two *operations* occur at any one time.

Formula DD:

$\forall$ *a:ATM_Status.*$\neg$(*a* $\in$ *isWaitingCard* $\wedge$ *a* $\in$ *isWaitingPin* $\wedge$
a $\in$ *isWaitingMoney* $\wedge$ *a* $\in$ *isWaitingReceiveCashAndCard* $\wedge$
a $\in$ *isWaitingReceiveCard* $\wedge$ *a* $\in$ *isWaitingCard* $\wedge$ *a* $\in$ *isUpdate*) $\wedge$
$\neg$(*a* $\in$ *isWaitingPin* $\wedge$ *a* $\in$ *isWaitingMone* $\wedge$
a $\in$ *isWaitingReceiveCashAndCard* $\wedge$ *a* $\in$ *isWaitingReceiveCard* $\wedge$
a $\in$ *isUpdate*) $\wedge$
$\neg$(*a* $\in$ *isWaitingMoney* $\wedge$ *a* $\in$ *isWaitingReceiveCard* $\wedge$
a $\in$ *isWaitingReceiveCashAndCard* $\wedge$ *a* $\in$ *isUpdate*) $\wedge$
$\neg$(*a* $\in$ *isWaitingReceiveCard* $\wedge$ *a* $\in$ *isWaitingReceiveCashAndCard*) $\wedge$
$\neg$(*a* $\in$ *isWaitingReceiveCashAndCard* $\wedge$ *a* $\in$ *isUpdate*)

The formula DD specifies constraints that for each *ATM_Status* there is only one corresponding *ATM_Status*, and this is either *isWaitingCard*; *isWaitingPin*; *isWaitingMoney*; *isWaitingReceiveCard*; *isWaitingReceiveCashAndCard*; or *isUpdate* to avoid inconsistency. No two *ATM_Statuses* occur at any one time.

4.2.4 Extension

As seen in Appendix (B.7), the extends types are mutually disjoint. Lines (4 to 13) and Lines (18 to 32) are expressed in the formulas EE and FF below respectively :

Formula EE:

$\forall$ *o:operation.(o* $\in$ *isEnterCard* $\vee$ *o* $\in$ *isTypePin* $\vee$
o $\in$ *isRequistCash* $\vee$ *o* $\in$ *isReceiveCashAndCard* $\vee$
o $\in$ *isReceiveCard)*
$\forall$ *o:operation.(o* $\in$ *isTypePin* $\vee$ *o* $\in$ *isRequistCash* $\vee$
o $\in$ *isReceiveCashAndCard* $\vee$ *o* $\in$ *isReceiveCard)*
$\forall$ *o:operation.(o* $\in$ *isRequistCash* $\vee$ *o* $\in$ *isReceiveCashAndCard* $\vee$
o $\in$ *isReceiveCard)*
$\forall$ *o:operation.(o* $\in$ *isReceiveCashAndCard* $\vee$ *o* $\in$ *isReceiveCard)*

The formula EE specifies constraints that for all *operation* , the operation (o) does not belong to more than one of *isEnterCard*; *isTypePin*; *isRequistCash*; *isReceiveCashAndCard*; and *isReceiveCard*.

Formula FF:
$\forall$ *a:ATM_Status.(a* $\in$ *isWaitingCard* $\vee$ *a* $\in$ *isWaitingPin* $\vee$
a $\in$ *isWaitingMoney* $\vee$ *a* $\in$ *isWaitingReceiveCashAndCard* $\vee$
a $\in$ *isWaitingReceiveCard* $\vee$ *a* $\in$ *isUpdate)*
$\forall$ *a:ATM_Status.(a* $\in$ *isWaitingPin* $\vee$ *a* $\in$ *isWaitingMoney* $\vee$
a $\in$ *isWaitingReceiveCashAndCard* $\vee$ *a* $\in$ *isWaitingReceiveCard* $\vee$
a $\in$ *isUpdate)*
$\forall$ *a:ATM_Status.(a* $\in$ *isWaitingMoney* $\vee$ *a* $\in$ *isWaitingReceiveCard* $\vee$
a $\in$ *isWaitingReceiveCashAndCard* $\vee$ *a* $\in$ *isUpdate)*
$\forall$ *a:ATM_Status.(a* $\in$ *isWaitingReceiveCard* $\vee$ *a* $\in$ *isWaitingReceiveCashAndCard* $\vee$
a $\in$ *isUpdate)*

The formula FF specifies constraints that for all *ATM_Status*, the ATM_Status (a) does not belong to more than one of *isWaitingCard*; *isWaitingPin*; *isWaitingMoney*; *isWaitingReceiveCard*; *isWaitingReceiveCashAndCard*; and *isUpdate*.

4.2.5 Facts

Alloy facts are assumed to be true. They represents the first and the second ATM system property. As seen in Appendix (B.9), Line (1) declares quantifiers to restrict the first fact. Line (2) declares quantifiers to restrict the second fact.

The first fact in (Line 1) as seen below is expressed in formula GG below:

First fact:

```
 (forall ((atm1 ATM)(atm2 ATM))(and(forall ((c1 Card))(=>
(cards atm2 c1)(cards atm1 c1)))
(forall ((c2 Card))(=>(cards atm1 c2)(cards atm2 c2)))
(forall ((c3 Card)(i Int))(=>(pin atm2 c3 i)(pin atm1 c3 i)))
(forall ((c4 Card)(i1 Int))(=>(pin atm1 c4 i1)(pin atm2 c4 i1)))))
```

Formula GG:

$\forall$ *atm1,atm2:ATM,c1:Card.(atm2,c1)* $\in$ *cards =>(atm1,c1)* $\in$ *cards* $\wedge$
$\forall$ *c2:Card.(atm1,c2)* $\in$ *cards =>(atm2,c2)* $\in$ *cards* $\wedge$
$\forall$ *c3:Card, i:Int.(atm2,c3,i)* $\in$ *pin =>(atm1,c3,i)* $\in$ *pin* $\wedge$
$\forall$ *c4:Card, i1:Int.(atm1,c4,i1)* $\in$ *pin =>(atm2,c4,i1)* $\in$ *pin*

The formula GG specifies constraints that: if different atoms of *ATM* belong to the same *Card*, then they have the same pin

The second fact in (Line 2) as seen below is expressed in formula HH below:

The formula HH specifies constraints that: in all atoms in *ATM* belong to a *card* have a balance greater than or equal to 0 and a positive pin number.

Second fact:

*(forall ((atm1 ATM)(card Card))(=>(cards atm1 card)(and
(forall ((i Int))(=>(balance atm1 card i)(>= i 0)))
(forall ((i1 Int))(=>(pin atm1 card i1)(>i1 0)))))))*

Formula HH:

$\forall$ *atm1:ATM,card:Card.(atm1,card)* $\in$ *cards =>*
$\forall$ *i:Int.(atm1,card,i)* $\in$ *balance =>(i >= 0)* $\wedge$
$\forall$ *i1:Int.(atm1,card,i1)* $\in$ *pin =>(i1 >0)*

4.2.6 Relations Declaration

Relations are translated to Boolean-valued SMT2 functions, uninterpreted, membership functions. As seen in Appendix (B.2) these functions are declared over top-level types because only top-level types are declared as sorts. Because all SMT functions are total function, relations are specified utilizing three parts: function name, received sorts, and returned value of Boolean type. The Boolean type includes two kinds of value, true for the tuples that are involved in the declared relation, or false for all others that are not involved.

Lines (1-7) Appendix (B.2) declare Boolean-valued SMT2 functions of relations. These relations are: *cards* and *inCard* which are declared as a Boolean-valued function over top-level types *ATM* and *Card* (Lines 1, and 2); *pin*, *balance*, and *money* which are declared as a Boolean-valued function over top-level types *ATM*, *Card*, and *Int* (Lines 3,4,and 5); *atmStatuse* and *op* which are declared as a Boolean-valued function over top-level types *ATM*, *ATM_Status*, and *ATM*, *Operations* respectively (Lines 6, and 7).

Lines (1-6) Appendix (B.8) declare constraint guarantee that each relation is defined for its specific types considering the multiplicity keywords constraints.

The first relation *cards: set Card* is not required to be translated to a formula to show its constraints as the *set* keyword constrains and allows any number of elements. Thus, its defined Boolean-valued function (Line 1) Appendix (B.2) is equivalent to its meaning.

Line (1) shows the second relation below and as seen in Appendix (B.8) is expressed in formula II below:

inCard : lone cards

(forall ((this ATM))(and (forall ((c1 Card))(=>(inCard this c1)
(cards this c1)))
(forall ((c3 Card)(c2 Card))(=>(and(inCard this c2)(inCard this c3))
(= c2 c3)))))

Formula II:

$\forall$ *atm:ATM,c1:Card.(atm,c1)* $\in$ *inCard =>(atm,c1)* $\in$ *cards* $\wedge$
$\forall$ *c2,c3:Card.(atm,c2)* $\in$ *inCard* $\wedge$ *(atm,c3)* $\in$ *inCard =>(c2 = c3)*

The formula II specifies constrains that: for all set of atoms *atm* in *ATM* and *c1* in *Card* if the atoms belong to *inCard* then they also belong to *cards* as well, and for all *c2* and *c3* in Card such that *atm* and *c2* belong to *inCard* and *atm* and *c3* belong to *inCard* then *c2* equals *c3* because the maximum number of (*inCard*) in used is one.

Line (2) shows the third relation below and as seen in Appendix (B.8) is expressed in formula JJ below:

pin : cards ->one Int

(forall ((this ATM))(and(forall ((c1 Card)(i Int))(=>(pin this c1 i)(cards this c1)))
(forall ((a1 Card))(=>(cards this a1)(and(exists ((i1 Int))(pin this a1 i1))
(forall ((i3 Int)(i2 Int))(=>(and(pin this a1 i2)(pin this a1 i3))(= i2 i3))))))))

Formula JJ:

$\forall$ *atm:ATM, c1:Card,i:Int.(atm,c1,i)* $\in$ *pin =>(atm,c1)*$\in$ *cards* $\wedge$
$\forall$ *a1:Card.(atm, a1)* $\in$ *cards =>*$\exists$ *i1:Int.(atm,a1,i1)* $\in$ *pin* $\wedge$
$\forall$ *i2, i3:Int.(atm, a1, i2)* $\in$ *pin* $\wedge$ *(atm, a1, i3)* $\in$ *pin =>(i2=i3)*

The formula JJ specifies constrains that: for all set of atoms *atm* in *ATM*, *c1* in *Card* and *i* in *Int* such that the atoms belong to the *pin* then they belong to *cards* as well, and for all set of atoms *a1* in *Card* such that the atoms *atm* and *a1* belong to *cards* then if there exist integer (i1) such that *atm*, *a1*, and *i1* belong to *pin*, and for all *i2* and *i3* in integer such that *atm*, *a1*, and *i2* belong to *pin* and *atm*, *a1*, and *i3* belong to *pin* then *i2* equals *i3* because the the number of (*pin* is exactly one.

Line (3) shows the fourth relation below and as seen in Appendix (B.8) is expressed in formula KK below:

balance : cards ->one Int

(forall ((this ATM))(and(forall ((c1 Card)(i Int))(=>(balance this c1 i)(cards this c1)))
(forall ((a1 Card))(=>(cards this a1)(and(exists ((i1 Int))(balance this a1 i1))
(forall ((i3 Int)(i2 Int))(=>(and(balance this a1 i2)(pin this a1 i3))
(= i2 i3))))))))

Formula KK:

$\forall$ *atm:ATM,c1:Card,i:Int.(atm,c1,i)* $\in$ *balance =>(atm,c1)* $\in$ *cards* $\wedge$
$\forall$ *a1:Card.(atm, a1)* $\in$ *cards =>*$\exists$ *i1:Int.(atm,a1,i1)* $\in$ *balance* $\wedge$
$\forall$ *i2, i3:Int.(atm, a1, i2)* $\in$ *balance* $\wedge$ *(atm, a1, i3)* $\in$ *balance =>(i2=i3)*

The formula KK specifies constrains that: for all set of atoms *atm* in *ATM*, *c1* in *Card* and *i* in *Int* such that the atoms belong to the *balance* then they belong to *cards* as well, and for all set of atoms *a1* in *Card* such that the atoms *atm* and *a1* belong to *cards* then if there exist integer (i1) such that *atm*, *a1*, and *i1* belong to *balance*, and for all *i2* and *i3* in integer such that *atm*, *a1*, and *i2* belong to *balance* and *atm*, *a1*, and *i3* belong to *balance* then *i2* equals *i3* because the the number of *balance* is exactly one.

Line (4) shows the fifth relation below and as seen in Appendix (B.8) is expressed in formula LL below:

money : cards ->lone Int

(forall ((atm ATM))(and(forall ((c1 Card)(i Int))(=>(money atm c1 i)(cards atm c1)))
(forall ((a1 Card)(i3 Int)(i2 Int))(=>(and(money atm a1 i2)(money atm a1 i3))
(= i2 i3)))))

Formula LL:

$\forall$ *atm:ATM,c1:Card,i:Int.(atm,c1,i)* $\in$ *money =>(atm,c1)* $\in$ *cards* $\wedge$
$\forall$ *a1:Card, i2, i3:Int.(atm, a1, i2)* $\in$ *money* $\wedge$ *(atm, a1, i3)* $\in$ *money =>(i2=i3)*

The formula LL specifies constrains that: for all set of atoms *atm* in *ATM*, *c1* in *Card* and *i* in *Int* such that the atoms belong to the *money* then they belong to *cards* as well, and for all set of atoms *a1* in *Card*, *i2* and *i3* in integer such that *atm*, *a1*, and *i2* belong to *money* and *atm*, *a1*, and *i3* belong to *money* then *i2* equals *i3* because the the number of *money* is at most one.

Line (5) shows the sixth relation below and as seen in Appendix (B.8) is expressed in formula MM below:

atmStatuse: one ATM_Status

(forall ((this ATM))(and(exists ((a1 ATM_Status))(atmStatuse this a1))
(forall ((a3 ATM_Status)(a2 ATM_Status))(=>(and
(atmStatuse this a2)(atmStatuse this a3))(= a2 a3)))))

Formula MM:

$\forall$ *atm:ATM* $\exists$ *a1:ATM_Status.(atm,a1)* $\in$ *atmStatus* $\wedge$
$\forall$ *a2,a3:ATM_Status.(atm, a2)* $\in$ *atmStatus* $\wedge$ *(atm, a3)* $\in$ *atmStatus*
=>(a2=a3)

The formula MM specifies constrains that: for all set of atoms *atm* in *ATM*, if there exists one atom *a1* in *ATM_Status* such that, the atoms *atm* and *a1* belong to the *atmStatus*, and for all *a2* and *a3* in *ATM_Status* such that *atm* and *a2* belong to *atmStatus* and *atm* and *a3* belong to *atmStatus* then *a2* equals *a3* because the the number of *ATM_Status* is exactly one.

Line (6) shows the seventh relation below and as seen in Appendix (B.8) is expressed in formula NN below:

op: lone Operations

(forall ((this ATM)(o2 Operations)(o1 Operations)
(=>(and(op this o1)(op this o2))(= o1 o2)))

Formula NN:

$\forall$ *atm:ATM, o1,o2:Operations.(atm,o1)*$\in$ *op* $\wedge$ *(atm,o2)*$\in$ *op*
=>(o1=o2)

The formula NN specifies constrains that: for all set of atoms *atm* in *ATM* and *o1* and *o2* in *Operations* such that the atoms *atm* and *o1* belong to the *op* and *atm* and *o2* belong to *op* as well then *o1* equals *o2*, because the the maximum number of *Operations* used is one.

4.2.7 Predicates

As seen in Appendix (B.10), the translation focuses on "inlining " of the predicate *ATMTransaction*. Inlining means without explicit declaration the *ATM* passes through 6 statuses. We illustrates each status individually.

Lines (5-8) show the predicate below and in Appendix (B.10) for the first status is expressed in formula OO below:

(atm1.atmStatuse)= WaitingCard and
crd in atm1.cards and no atm1.inCard and
no atm1.op and no atm1.money and

(forall ((a1 ATM_Status))(=>(atmStatuse atm1 a1)(isWaitingCard a1)))
(forall ((w ATM_Status))(=>(isWaitingCard w)(atmStatuse atm1 w)))
(cards atm1 crd)
(forall ((c2 Card))(not(inCard atm1 c2)))
(forall ((o Operations))(not (op atm1 o)))
(forall ((m Int)(c9 Card))(not(money atm1 c9 m)))

Formula OO:

$\forall$ *atm1:ATM,a1:ATM_Status.(atm1,a1)* $\in$ *atmStatus =>(a1)* $\in$ *isWaitinCard*
$\forall$ *atm1:ATM,w:ATM_Status.(w)* $\in$ *isWaitinCard =>(atm1,w)* $\in$ *atmStatus* $\wedge$
$\forall$ *atm1:ATM,crd:Card.(atm1, crd)* $\in$ *cards* $\wedge$
$\forall$ *atm1:ATM,c2:Card.(atm1,c2)* $\notin$ *inCard* $\wedge$
$\forall$ *atm1:ATM,o:Operations.(atm1,o)* $\notin$ *op* $\wedge$
$\forall$ *atm1:ATM,m:Int,C9:Card.(atm1,C9,m)* $\notin$ *money*

The formula OO specifies constrains that: for all atoms *atm1* in *ATM* and *a1* in *ATM_Status* such that the atoms *atm1* and *a1* belong to *atmStatus* then the atom *a1* belongs to *isWaitinCard*. For all atoms *atm1* in *ATM* and *w* in *ATM_Status* such that the atom *w* belongs to *isWaitinCard* then the atoms *atm1* and *w* belong to *atmStatus*. For all atoms *atm1* in *ATM* and *crd* in *Card* such that the atoms *atm1* and *crd* belong to *cards*. For all atoms *atm1* in *ATM* and *c2* in *Card* such that the atoms *atm1* and *c2* do not belong to *inCard*. For all atoms *atm1* in *ATM* and *o* in *Operations* such that the atoms *atm1* and *o* do not belong to *op*. For all atoms (atm1) in *ATM*, *m* in *Int*, and *C9* in *Card* such that the atoms *atm1*, *C9*, and *m* do not belong to *money*.

Lines (9-13) show the predicate below and in Appendix (B.10) for the second status is expressed in formula PP below:

atm2.op= EnterCard and atm2.inCard = crd and
atm2.balance = atm1.balance and
(atm2.atmStatuse) = WaitingPin and
no atm2.money and

```
(forall ((o1 Operations))(=>(op atm2 o1)(isEnterCard o1)))
(forall ((e Operations))(=>(isEnterCard e)(op atm2 e)))
(forall ((c3 Card))(=>(inCard atm2 c3)(= crd c3)))
(inCard atm2 crd)
(forall ((c5 Card)(i1 Int))(=>(balance atm2 c5 i1)(balance atm1 c5 i1)))
(forall ((c6 Card)(i2 Int))(=>(balance atm1 c6 i2)(balance atm2 c6 i2)))
(forall ((a15 ATM_Status))(=>(atmStatuse atm2 a15)(isWaitingPin a15)))
(forall ((w1 ATM_Status))(=>(isWaitingPin w1)(atmStatuse atm2 w1)))
(forall ((m Int)(c9 Card))(not(money atm2 c9 m))))))
```

Formula PP:

$\forall$ *atm2:ATM,o1:Operations.(atm2,o1)* $\in$ *op =>(o1)* $\in$ *isEnterCard* $\wedge$
$\forall$ *atm2:ATM,e:Operations.(e)* $\in$ *isEnterCard =>(atm2,e)* $\in$ *op* $\wedge$
$\forall$ *atm2:ATM,c3:Card.(atm2,c3)* $\in$ *inCard =>(crd=c3)* $\wedge$
$\forall$ *atm2:ATM,crd:Card.(atm2, crd)* $\in$ *inCard* $\wedge$
$\forall$ *atm1,atm2:ATM,c5:Card,i1:Int.(atm2,c5,i1)* $\in$ *balance =>(atm1,c5,i1)* $\in$ *balance* $\wedge$
$\forall$ *atm1,atm2:ATM,c6:Card,i2:Int.(atm1,c6,i2)* $\in$ *balance =>(atm2,c6,i2)* $\in$ *balance* $\wedge$
$\forall$ *atm2:ATM,a15:ATM_Status.(atm2,a15)* $\in$ *atmStatus =>(a15)* $\in$ *isWaitinPin* $\wedge$
$\forall$ *atm2:ATM,w1:ATM_Status.(w1)* $\in$ *isWaitinPin =>(atm2,w1)* $\in$ *atmStatus* $\wedge$
$\forall$ *atm2:ATM,m:Int,C9:Card.(atm2,C9,m)* $\notin$ *money*

The formula PP specifies constraints that: for all atoms *atm2* in *ATM* and *o1* in *Operations* such that the atoms *atm2* and *o1* belong to *op* then the atom *o1* belongs to *isEnterCard*. For all atoms *atm2* in *ATM* and *e* in *Operations* such that the atom *e* belongs to *isEnterCard* then the atoms *atm2* and *e* belong to *op*. For all atoms *atm2* in *ATM* and *c3* in *Cards* such that the atoms *atm2* and *c3* belong to *inCard* then *crd* is in used: *crd* equals *c3*. For all atoms *atm2* in *ATM* and *crd* in *Card* such that the atoms *atm2* and *crd* belong to *inCard*.

For all atoms *atm1* and *atm2* in *ATM*, *c5* in *Card*, and *i1* in *Int* such that the atoms *atm2*, *c5*, and *i1* belong to *balance* then the atoms *atm1*, *c5*, and *i1* in the first status belong to *balance* as well. For all atoms *atm1* and *atm2* in *ATM*, *c6* in *Card*, and *i2* in *Int* such that the atoms *atm1*, *c6*, and *i2* belong to *balance* then the atoms *atm2*, *c6*, and *i2* in the second status belong to *balance* as well.

For all atoms *atm2* in *ATM* and *a15* in *ATM_Status* such that the atoms *atm2* and *a15* belong to *atmStatus* then the atoms a15 belongs to *isWaitinPin*. For all atoms *atm2* in *ATM* and *w* in *ATM_Status* such that the atom *w1* belongs to *isWaitinPin* then the atoms *atm2* and *w1* belongs to *atmStatus*.

For all atoms *atm2* in *ATM*, *m* in *Int*, *C9* in *Card* such that, the atoms *atm2*, *C9*, and *m* do not belong to *money*.

Lines (14-19) show the predicate below and in Appendix (B.10) for the third status is expressed in formula QQ below:

atm3.op= TypePin and
atm3.inCard = atm2.inCard and
atm3.inCard.(atm3.pin)=pn and
atm3.balance = atm2.balance and
(atm3.atmStatuse) = WaitingMoney and
no atm3.money and

(forall ((o2 Operations))(=>(op atm3 o2)(isTypePin o2)))
(forall ((t Operations))(=>(isTypePin t)(op atm3 t)))
(forall ((c7 Card))(=>(inCard atm3 c7)(inCard atm2 c7)))
(forall ((c8 Card))(=>(inCard atm2 c8)(inCard atm3 c8)))
(forall ((i3 Int))(=>(exists ((c9 Card))(and(inCard atm3 c9)(pin atm3 c9 i3)))
(= pn i3)))
(exists ((c10 Card))(and (inCard atm3 c10)(pin atm3 c10 pn)))
(forall ((c11 Card)(i5 Int))(=>(balance atm3 c11 i5)(balance atm2 c11 i5)))
(forall ((c12 Card)(i6 Int))(=>(balance atm2 c12 i6)(balance atm3 c12 i6)))
(forall ((a32 ATM_Status))(=>(atmStatuse atm3 a32)(isWaitingMoney a32)))
(forall ((w2 ATM_Status))(=>(isWaitingMoney w2)(atmStatuse atm3 w2)))
(forall ((m Int)(c9 Card))(not(money atm3 c9 m)))

Formula QQ:

$\forall$ *atm3:ATM,o2:Operations.(atm3,o2)* $\in$ *op =>(o2)* $\in$ *isTypingPin* $\wedge$
$\forall$ *atm3:ATM,t:Operations.(t)* $\in$ *isTypingPin =>(atm3,t)* $\in$ *op* $\wedge$
$\forall$ *atm2,atm3:ATM,c7:Card.(atm3,c7)* $\in$ *inCard =>(atm2, c7)* $\in$ *inCard* $\wedge$
$\forall$ *atm2,atm3:ATM,c8:Card.(atm2,c8)* $\in$ *inCard =>(atm3, c8)* $\in$ *inCard* $\wedge$
$\forall$ *atm3:ATM,pn,i3:Int* $\exists c9 : Card.(atm3, c9) \in$ *inCard* $\wedge$
(atm3,c9,i3) $\in$ *pin =>(pn = i3)*
$\exists$ *atm3:ATM,c10:Card,pn:Int.(atm3,c10)* $\in$ *inCard* $\wedge$ *(atm3, c10, pn)* $\in$ *pin* $\wedge$
$\forall$ *atm2,atm3:ATM,c11:Card,i5:Int.(atm3,c11,i5)* $\in$ *balance =>(atm2,c11,i5)* $\in$ *balance* $\wedge$
$\forall$ *atm2,atm3:ATM,c12:Card,i6:Int.(atm2,c12,i6)* $\in$ *balance =>(atm3,c12,i6)* $\in$ *balance*$\wedge$
$\forall$ *atm3:ATM,a32:ATM_Status.(atm3,a32)* $\in$ *atmStatus =>(a32)* $\in$ *isWaitinMoney* $\wedge$
$\forall$ *atm3:ATM,w2:ATM_Status.(w2)* $\in$ *isWaitinMoney =>(atm3,w2)* $\in$ *atmStatus* $\wedge$
$\forall$ *atm3:ATM,m:Int,C9:Card.(atm3,C9,m)* $\notin$ *money*

The formula QQ specifies constraints that: for all atoms *atm3* in *ATM* and *o2* in *Operations* such that the atoms *o2* and *atm3* belong to *op* then the only *o2* atom belongs to *isTypingPin* in the third status *atm3*. For all atoms *atm3* in *ATM* and *t* in *Operations* such that the atom *t* belongs to *isTypingPin* then the atoms *atm3* and *t* belong to *op*. For all atoms *atm2* and *atm3* in *ATM* and *c7* in *Card* such that, the atoms *atm3* and *c7* belong to *inCard* then the atoms *atm2* and *c7* belong to *inCard*; For all atoms *atm2* and *atm3* in *ATM* and *c8* in *Card* such that, the atoms *atm2* and *c8* belong to *inCard* then the atoms *atm3* and *c8* belong to *inCard* which means, the used card in the second status is the same card in third status. For all atoms *atm3* in *ATM* and *pn*; *i1* in *Int* if there exists atom c9 in *Card* such that the existed *Card c9* in the third status *atm3* belong to *inCard* and the atoms *atm3*, *c9*, and *i3* belong to pin then *pn* equals *i3*. So, if there exists a card *c10* in *Card*, *atm3* in *ATM*, and *pn* in *Int* such that *atm3* and *c10* belong to *inCard* and *atm3*, *c10*, and *pn* belong to *pin*. For all atoms *atm2* and *atm3* in *ATM*, *c11* in *Card*, and *i5* in *Int* such that the atoms *atm3*, *c11*, and *i5* belong to *balance* then the atoms *atm2*, *c11*, and *i5* with same card and same amount belong to balance as well (the balance still remains the same in the third status). For all atoms *atm2* and *atm3* in *ATM*, *c12* in *Card*, and *i6* in *Int* such that the atoms *atm2*, *c12*, and *i6* belong to *balance* then the atoms *atm3*, *c12*, and *i6* belong to balance. For all atoms *atm3* in *ATM* and *a32* in *ATM_Status* such that the atoms *atm3* and *a32* belong to *atmStatus* then the only *a32* atom belongs to *isWaitinMoney*. For all atoms *atm3* in *ATM* and *w2* in *ATM_Status* such that the atom *w2* belongs to *isWaitinMoney* then the atoms *atm3* and *w2* belong to *atmStatus*. For all atoms *atm3* in *ATM*, *m* in *Int*, and *C9* in *Card* such that the atoms *atm3*, *C9*, and *m* do not belong to *money*.

Lines (20-23) show the predicate below and in Appendix (B.10) for the fourth status is expressed in formula RR below:

atm4.op = RequistCash and
atm4.balance = atm3.balance and
atm4.inCard = atm3.inCard and
atm4.inCard.(atm4.money)= mon and

(forall ((o3 Operations))(=>(op atm4 o3)(isRequistCash o3)))
(forall ((r Operations))(=>(isRequistCash r)(op atm4 r)))
(forall ((c13 Card)(i7 Int))(=>(balance atm4 c13 i7)(balance atm3 c13 i7)))
(forall ((c14 Card)(i8 Int))(=>(balance atm3 c14 i8)(balance atm4 c14 i8)))
(forall ((c15 Card))(=>(inCard atm4 c15)(inCard atm3 c15)))
(forall ((c16 Card))(=>(inCard atm3 c16)(inCard atm4 c16)))
(forall ((i3 Int))(=>(forall ((c9 Card))(and(inCard atm4 c9)(money atm4 c9 i3)))
(= mon i3)))
(forall ((c10 Card))(and(inCard atm4 c10)(money atm4 c10 mon)))

Formula RR:

$\forall$ *atm4:ATM,o3:Operations.(atm4,o3)* $\in$ *op =>(o3)* $\in$ *isRequestCash* $\wedge$
$\forall$ *atm4:ATM,r:Operations.(r)* $\in$ *isRequestCash =>(atm4,r)* $\in$ *op* $\wedge$
$\forall$ *atm3,atm4:ATM,c13:Card,i7:Int.(atm4,c13,i7)* $\in$ *balance =>* $\wedge$
(atm3,c13,i7) $\in$ *balance* $\wedge$
$\forall$ *atm3,atm4:ATM,c14:Card,i8:Int.(atm3,c14,i8)* $\in$ *balance =>* $\wedge$
(atm4,c14,i8) $\in$ *balance* $\wedge$
$\forall$ *atm3,atm4:ATM,c15:Card.(atm4,c15)* $\in$ *inCard =>(atm3, c15)* $\in$ *inCard* $\wedge$
$\forall$ *atm3,atm4:ATM,c16:Card.(atm3,c16)* $\in$ *inCard =>(atm4, c16)* $\in$ *inCard* $\wedge$
$\forall$ *atm4:ATM,mon,i3:Int,c9:Card.(atm4,c9)* $\in$ *inCard* $\wedge$ $\wedge$
(atm4,c9,i3) $\in$ *money =>(mon = i3)* $\wedge$
$\forall$ *c10:Card.(atm4,c10)* $\in$ *inCard* $\wedge$ *(atm4,c10,mon)* $\in$ *money*

The formula RR specifics constraints that: for all atoms *atm4* in *ATM* and *o3* in *Operations* such that the atoms *atm4* and *o3* belong to *op* then the only *o3* atom belongs to *isRequestCash* in the fourth status *atm4*. For all atoms *atm4* in *ATM* and *r* in *Operations* such that the atom *r* belongs to *isRequestCash* then the atoms *atm4* and *r* belong to *op*. For all atoms *atm3* and *atm4* in *ATM*, *c13* in *Card*, and i7 in *Int* such that the atoms *atm4*, *c13*,and *i7* belong to *balance* then the atoms *atm3*, *c13*, and *i7* belong to *balance* (the balance still remains the same in the fourth status). For all atoms *atm3* and *atm4* in *ATM*, *c14* in *Card*, and i8 in *Int* such that the atoms *atm3*, *c14*,and *i8* belong to *balance* then the atoms *atm4*, *c14*, and *i8* belong to *balance*. For all atoms *atm3* and *atm4* in *ATM* and *c15* in *Card* such that the atoms *atm4* and *c15* belong to *inCard* then the atoms *atm3* and *c15* belong to *inCard* as well which means, the used card in the fourth status is the same card in third status. For all atoms *atm3* and *atm4* in *ATM* and *c16* in *Card* such that the atoms *atm3* and *c16* belong to *inCard* then the atoms *atm4* and *c16* belong to *inCard* as well. For all atoms *atm4* in *ATM*, *mon*; *i1* in *Int*, and *c9* in *Card* such that the *Card c9* in the fourth status *atm4* belong to *inCard* and the atoms *atm4*, *c9*, and *i3* belong to *money* then *mon* equals *i3*. For all c10:Card, then atm4, and c10 belong to *inCard*; and atm4, c10,and mon belong to *money*.

Lines (24-27) show the predicate below and in Appendix (B.10) for the fifth status is expressed in formula SS below:

atm5.inCard = atm4.inCard and
((mon <= atm4.inCard.(atm4.balance) and mon >0) and
(atm5.inCard.(atm5.balance) = atm4.inCard.(atm4.balance).minus[mon]
and atm5.op= ReceiveCashAndCard and
(atm4.atmStatuse) = WaitingReceiveCashAndCard)))and
or(((mon >atm4.inCard.(atm4.balance) or mon <0) and
(atm5.inCard.(atm5.balance) = atm4.inCard.(atm4.balance) and
atm5.op= ReceiveCard and (atm4.atmStatuse) = WaitingReceiveCard))
(atm5.atmStatuse) = Update and

```
 (forall ((c17 Card))(=>(inCard atm5 c17)(inCard atm4 c17)))
(forall ((c18 Card))(=>(inCard atm4 c18)(inCard atm5 c18)))
(or (and(forall ((i9 Int))(=>(exists ((c19 Card))(and
 (inCard atm4 c19)(balance atm4 c19 i9)))(<= mon i9)))(<0 mon)
(forall ((i10 Int)(i11 Int))(=(and(exists ((c20 Card))
(and(inCard atm5 c20)(balance atm5 c20 i10)))
(exists ((c21 Card))(and(inCard atm4 c21)(balance atm4 c21 i11))))
 (= i10 (- i11 mon))))
(forall ((o4 Operations))(=>(op atm5 o4)(isReceiveCashAndCard o4)))
(forall ((r1 Operations))(=>(isReceiveCashAndCard r1) (op atm5 r1)))
(forall ((a57 ATM_Status))(=>(atmStatuse atm4 a57)(isWaitingReceiveCashAndCard a57)))
(forall ((w3 ATM_Status))(=>(isWaitingReceiveCashAndCard w3)(atmStatuse atm4 w3)))))
(and(or(forall ((i12 Int))(=>(exists ((c22 Card))(and(inCard atm4 c22)
(balance atm4 c22 i12)))(<i12 mon)))(<mon 0))
(forall ((i13 Int))(=>(exists ((c23 Card))(and(inCard atm5 c23)(balance atm5 c23 i13)))
(exists ((c24 Card))(and(inCard atm4 c24)(balance atm4 c24 i13)))))
(forall ((i14 Int))(=>(exists ((c25 Card))(and(inCard atm4 c25)(balance atm4 c25 i14)))
(exists ((c26 Card))(and(inCard atm5 c26)(balance atm5 c26 i14)))))
(forall ((o5 Operations))(=>(op atm5 o5)(isReceiveCard o5)))
(forall ((r2 Operations))(=>(isReceiveCard r2)(op atm5 r2)))
(forall ((a72 ATM_Status))(=>(atmStatuse atm4 a72)(isWaitingReceiveCard a72)))
(forall ((w4 ATM_Status>isWaitingReceiveCard w4)(atmStatuse atm4 w4)))))
(forall ((a75 ATM_Status))(=>(atmStatuse atm5 a75)(isUpdate a75)))
(forall ((u ATM_Status))(=>(isUpdate u)(atmStatuse atm5 u)))
```

Formula SS:

∀ atm5,atm4:ATM,c17:Card.(atm5,c17) ∈ inCard =>(atm4,c17) ∈ inCard ∧
∀ atm5,atm4:ATM,c18:Card.(atm4,c18) ∈ inCard =>(atm5,c18) ∈ inCard ∧
∀ atm5,atm4:ATM,i9,mon:Int ∃ c19:Card.(atm4,c19) ∈ inCard ∧
(atm4,c19,i9) ∈ balance =>mon <= i9 ∧ mon >0 =>
∀ atm5,atm4:ATM,i10,i11,mon:Int (∃ c20:Card.(atm5,c20) ∈ inCard ∧
(atm5,c20,i10) ∈ balance ∧ ∃ c21:Card.(atm4,c21) ∈ inCard ∧
(atm4,c21,i11) ∈ balance) = ((i10 = i11 - mon))
∀ atm5:ATM,o4:Operations.(atm5,o4) ∈ op =>(o4) ∈ isReceiveCashAndCard ∧
∀ atm5:ATM,r1:Operations.(r1) ∈ isReceiveCashAndCard =>(atm5,r1) ∈ op ∧
∀ atm4:ATM,a57:ATM_Status.(atm4,a57) ∈ atmStatuse =>
(a57) ∈ isWaitingReceiveCashAndCard
∀ atm4:ATM,w3:ATM_Status.(w3) ∈ isWaitingReceiveCashAndCard =>
(atm4,w3) ∈ atmStatuse ∨**))**
∀ atm4:ATM,i12,mon:Int ∃ c22:Card.(atm4,c22) ∈ inCard ∧
(atm4,c22,i12) ∈ balance =>mon >= i12 ∨ mon <0 ∧ **(**
∀ atm4,atm5:ATM,i13:Int (∃ c23:Card.(atm5,c23) ∈ inCard ∧
(atm5,c23,i13) ∈ balance =>∃ c24:Card.(atm4,c24) ∈ inCard ∧
(atm4,c24,i13) ∈ balance) ∧
∀ atm4,atm5:ATM,i14:Int (∃ c25:Card.(atm4,c25) ∈ inCard ∧
(atm4,c25,i14) ∈ balance =>∃ c26:Card.(atm5,c26) ∈ inCard ∧
(atm5,c26,i14) ∈ balance) ∧
∀ atm5:ATM,o5:Operations.(atm5,o5) ∈ op =>(o5) ∈ isReceiveCard ∧
∀ atm5:ATM,r2:Operations.(r2) ∈ isReceiveCard =>(atm5,r3) ∈ op ∧
∀ atm4:ATM,a72:ATM_Status.(atm4,a72) ∈ atmStatuse =>
(a72) ∈ isWaitingReceiveCard ∧
∀ atm4:ATM,w4:ATM_Status.(w4) ∈ isWaitingReceiveCard =>
(atm4,w4) ∈ atmStatuse **))))**
∀ atm5:ATM,a75:ATM_Status.(atm5,a75) ∈ atmStatuse =>
(a75) ∈ isUpdate ∧
∀ a54:ATM,u:ATM_Status.(u) ∈ isUpdate =>
(atm5,u) ∈ atmStatuse

The formula SS specifics constraints that: for all atoms *atm5* and *atm4* in *ATM* and *c17* in *Card* such that the atom *atm5* and *c17* belong to (inCard) then atoms *atm4* and *c17* belong to *inCard*. For all atoms *atm5* and *atm4* in *ATM* and *c18* in *Card* such that the atom *atm5* and *c18* belong to (inCard) then atoms *atm5* and *c18* belong to *inCard*. For all atoms *atm5* and *atm4* in *ATM*, *i9*; *mon* in *Int* if there exists an atom *c19* in *Card* such that the atoms *atm5* and *c19* belong to *inCard* and the atoms *atm5*, *c19*, and *i9* belong to *balance* then *mon* should be less than or equals to the balance *i9* and greater than zero, if this formula true, then for all atoms *atm5* and *atm4* in *ATM* and *i10*; *i1*; *mon* in *Int* if there exists an atom *c20* in *Card* such that the atoms *atm5* and *c20* belong to *inCard* and the atoms *atm5*, *c20*, and *i10* belong to *balance* and if there exists an atom *c21* in *Card* such that the atoms *atm4* and *c21* belong to *inCard* and the atoms *atm4*, *c21*, and *i11* belong to balance equals to the balance after subtracted by the required amount of *mon*; and in this case: for all atoms *atm5* in *ATM* and *o4* in *Operations* such that the atoms *atm5* and *o4* belong to (op) then atom *o4* belong to *isReceiveCashAndCard*. For all atoms *atm5* in *ATM* and *r* in *Operations* such that the atom *r* belongs to (isReceiveCashAndCard) then atoms *atm5* and *r1* belong to *op*. For all atoms *atm4* in *ATM* and *a57* in *ATM_Status* such that the atoms *atm4* and *a57* belong to (atmStatuse) then atom *a57* belong to *isWaitingReceiveCashAndCard*. For all atoms *atm4* in *ATM* and *w3* in *ATM_Status* such that the atoms *w3* belongs to (isWaitingReceiveCashAndCard) then atom *atm4* and *w3* belong to *atmStatuse*.

For all atoms *atm4* in *ATM*, *i12*; *mon* in *Int* if there exists an atom *c22* in *Card* such that the atoms *atm4* and *c22* belong to *inCard* and the atoms *atm4*, *c22*, and *i12* belong to *balance* then *mon* should be greater than the balance *i12* or less than zero, if this formula true, then for all atoms *atm5* and *atm4* in *ATM* and *i13* in *Int* if there exists an atom *c23* in *Card* such that the atoms *atm5* and *c23* belong to *inCard* and the atoms *atm5*, *c23*, and *i13* belong to *balance* then if there exists an atom *c24* in *Card* such that the atoms *atm4* and *c24* belong to *inCard* and the atoms *atm4*, *c24*, and *i13* belong to balance and; for all atoms *atm4* and *atm5* in *ATM*, *i14* in *Int* if there exists an atom *c25* in *Card* such that the atoms *atm4* and *c25* belong to *inCard* and the atoms *atm4*, *c25*, and *i14* belong to *balance* then if there exists an atom *c26* in *Card* such that the atoms *atm5* and *c26* belong to *inCard* and the atoms *atm5*, *c26*, and *i14* belong to *balance*. For all atoms *atm5* in *ATM* and *o* in *Operations* such that the atoms *atm5* and *o5* belong to *op* then the atom *o5* belongs to *isReceiveCard*. For all atoms *atm5* in *ATM* and *r2* in *Operations* such that the atom *r2* belongs to *isReceiveCard* then the atoms *atm5* and *r2* belong to *op*. For all atoms *atm4* in *ATM*, *a72* in *ATM_Status* such that the atoms *atm4* and *a72* belong to *atmStatuse* then the atom *a72* belongs to *isWaitingReceiveCard*. For all atoms *atm4* in *ATM* and *w4* in *ATM_Status* such that the atom *w4* belongs to *isWaitingReceiveCard* then the atoms *atm4* and *w4* belong to *atmStatuse*. For all atoms *atm5* in *ATM* and *a72* in *ATM_Status* such that the atoms *atm5* and *a72* belong to *atmStatuse* then the atom *a75* belongs to *isUpdate*. For all atoms *atm5* in *ATM* and *u* in *ATM_Status* such that the atom *u* belongs to *isUpdate* then the atoms *atm5* and *u* belong to *atmStatuse*.

Lines (28) for the predicate below and in Appendix (B.10) for the sixth status is expressed in formula TT below:

**no atm6.inCard and (atm6.atmStatuse) = WaitingCard and
no atm6.op no atm6.money**

(forall ((c27 Card))(not(inCard atm6 c27)))
(forall ((a79 ATM_Status))(=>(atmStatuse atm6 a79)(isWaitingCard a79)))
(forall ((w5 ATM_Status))(=>(isWaitingCard w5)(atmStatuse atm6 w5)))
(forall ((o6 Operations))(not(op atm6 o6))))

Formula TT:

$\forall$ *atm6:ATM, c27:Card. (atm6,c27)* $\notin$ *inCard*
$\forall$ *atm6:ATM, a79 ATM_Status.(atm6,a79)* $\in$ *atmStatuse =>(a79)* $\in$ *isWaitingCard*
$\forall$ *atm6:ATM, w5 ATM_Status.(w5)* $\in$ *isWaitingCard =>(atm6,w5)* $\in$ *atmStatuse*
$\forall$ *atm6:ATM, o6:Operations.(atm6,o6)* $\notin$ *op* **)**

The formula TT specifics constraints that: for all atoms *atm6* in *ATM* and *c27* in *Card* such that the atoms *atm6* and *c27* do not belong to *inCard*. For all atoms *atm6* in *ATM* and *a79* in *ATM_Status* such that, the atoms *atm6* and *a79* belong to *atmStatuse* then the atom *a79* belongs to *isWaitingCard*. For all atoms *atm6* in *ATM* and *w5* in *ATM_Status* such that, the atom *w5* belongs to *isWaitingCard* then the atoms *atm6* and *w5* belong to *atmStatuse*. For all atoms *atm6* in *ATM* and *o6* in *Operations* such that, the atoms *atm6* and *o6* do not belong to *op*.

4.2.8 Assertion

Assertions are intended to be checked. As seen in Appendix (B.10), we negate an assertion so that any instance found by the SMT solver will be a counterexample to the assertion. If the SMT solver does not find an instances, the assertion is proven correct.

Line (29) for the ATM system requirement and Lines (4) in Appendix (B.10) are expressed in formula UU below:

mon <= crd.(atm1.balance) and mon >0 and
ATMTransaction implies
crd.(atm5.balance) = crd.(atm1.balance).minus[mon]

(forall ((i Int))(=>(balance atm1 crd i)(<= mon i)))(<0 mon)
and ATMTransaction =>
(forall ((i15 Int)(i16 Int))(=>(and(balance atm5 crd i15)
(balance atm1 crd i16))
(= i15 (- i16 mon))))

Formula UU:

$\forall$ *atm1,atm2,atm3,atm4,atm5,atm6:ATM,pn,mon:Int,crd:Card*
$\forall$ *i:Int.(atm1,crd,i)* $\in$ *balance =>(mon <=i)* $\wedge$
(mon >0) $\wedge$ **(** *ATMTransaction =>*
$\forall$ *atm1,atm5:ATM,crd:Card, i15,i16,mon:Int.(atm5,crd,i15)* $\in$ *balance*
$\wedge$ *(atm1,crd,i16)* $\in$ *balance =>i15= (i16 - mon)***)**

The formula UU specifies constraints that: for all atoms *atm1*, *atm2*, *atm3*, *atm4*, *atm5*, and *atm6* in *ATM*, *crd* in *Card*, and atoms *pn*; *mon* in *Int* such that, the atoms *atm1*, *crd*, and *i* belong to *balance* then the required *mon* should be less than or equal to the balance *i* and greater than zero to achieve the third ATM property. Otherwise, SMT solver finds an instance as a counterexample; and the predicate *ATMTransaction* then for all atoms (atm1,atm5) in *ATM*, *crd* in *Card*, and *i15*; *i16*; *mon* in *Int* such that the atoms *atm5*, *crd*, and *i15* belong to *balance* and the atoms *atm1*, *crd*, *i16* belong to *balance* then the updated balance *i15* equals (the exists balance *i16* - required money *mon*).

4.3 Results

We used the *check-sat* command as seen in Appendix (C), part (C.11) (Line 1) to ask the SMT solver to *check* whether the *negation* of the conjunction of the provided assertions is unsatisfiable or not.

In this section, we present the results we achieved from modelling, analysing, and checking the three properties of ATM system, using the Alloy Analyser, and bounded SAT solver. The **first property** is that all ATM cards with corresponding PINs should be identified in any ATM machine. The **second property** is that the balance and PIN of cards that interact with the ATM should not be less than zero (positive). The **third property** is that the required amount of money should be less than or equal to the available balance, and greater than zero for the used card.

The SMT solver spent 0.022s to try to find a model that satisfied the negation of the set of formula, but it did not find one.

However, omitting the logical formulas as seen in Line (4) Appendix (C.10) which represented the third property of the ATM model, the SMT solver provided the *SAT* result in 0.039s. By using the get-model command as seen in Appendix (C.11) (Line 2), the SMT solver could find a model. So that the instance (model) found by the SMT solver is a counterexample to the assertion.

4.4 Comparison Between SAT and SMT results, and SAT and SMT Tools

In this section we compare the results we achieved in modelling the ATM system and checking the satisfiability of their properties using the SAT and SMT solvers. Also, we compare the SAT and SMT solvers as tools. The goal of Alloy which is analysing and checking a model is different from the goal of Z3 which is proving the satisfiability of the model. We compare the following criteria: decidability, time spent to generate a counterexample, limitation and the accuracy in generating a counterexample, how is the counterexample look like, the abilities for Alloy and Z3, and the techniques for both solvers.

- Decidability

To compare the decidability, the Alloy Analyser restricts both operations *simulation* (for checking consistency of the model) and *checking* (for generating a counterexample) to a finite scope. A scope gives a finite bound on the sizes of the domains in the specification. Alloy translates its problem into a bounded SAT problem. A bounded SAT problem is passed to the SAT solver to solve the satisfiability problem. Thus, the SAT solver is decidable. On the other hand, Z3 translates its problem into unbounded SMT problem. Unbounded SMT problem is passed to the SMT solver. Thus, the SMT solver is undecidable in general.

In addition, Alloy uses an engine as a SAT solver and a SAT solver does not identify an unbounded problem. Consequently, the Alloy Analyser must translates its problem to a propositional logic. It needs therefore scopes to do this translation as propositional logic is decidable. On the other hand, the Z3 SMT solver takes input formula as first-order predicate logic and FOL is undecidable. Verification within a finite scope is decidable, thus we used Alloy for limiting our analysis to a finite scope in the specification that bounds the sizes of the types as the Alloy Analyser is decidable. In our ATM example, we could bound the problem to be small by restricting the scope of search to be 1 atom for all entities except the ATM to be 6 atoms according to the number of statuses. However, in Z3 we could not limit the sorts and the search for the counterexamples was infinite.

- Time

The Alloy analysis time is the summation of the time spent on generating CNF, and in the SAT solver as reported by the Alloy Analyzer 4.1.10 running the SAT4J solver. In Z3 time is what the SMT solver reports in proving the satisfiability of the formula. In comparison the time (in second) is measured on an Intel Core2Quad, 2.7GHz, 4GB memory. The SAT solver in the Alloy Analyser spent 0.233s to check the satisfiability for the properties of ATM system. On the other hand, the Z3 SMT solver spent 0.039s to generate the counterexample to check the satisfiability for the same properties.

- Limitation and Confidence

In finite scopes, the Alloy analysis is noticeably faster and reduces the size of the model to accelerate and making more confident finding a counterexample. For example, in analysing the ATM system, the Alloy Analyser does not find a counterexample in the first and second property when the scope was 1,2,3, or 4. However, when we increased the scope we could see a counterexample. In this case we know that there is a problem as seen in Figure 3.6. However, when the Alloy Analyser does not find a counterexample in the third property when the scope was increased and limited to be 1 atom for each signature but 6 atoms for ATM, we feel more confident about the specification in this limitation.

To increase our confidence, we increased the number of scopes with the same result. However, if there is a counterexample in the larger scope, we do not know what is the maximum number of scopes we need to try. In fact, the Analyser has been developed especially to do the process quickly in a small scope analysis. In the study done by [77] the experimental results demonstrates that specification with a default scope of three can be analysed well in a minute.

In contrast, Z3 SMT is an unbounded solver. When Alloy gives a counterexample within finite scope and Z3 gives a counterexample in infinite scope, there is no problem in this case and the comparison is equivalent. However, in contrast when Alloy says there is no counterexample within a finite scope and Z3 says there is no counterexample in an infinite scope, Alloy may find a counterexample in a larger scope. That happened when analysing the first and second property when the scope was from 1 to 4, and the SAT solver says there is no counterexample. However, the Z3 SMT solver guarantees that there is a counterexample and says the result is SAT. Consequently, the comparison is not equivalent in this case because if we did not increase the number of scopes, we may think that the property is correct. As a result, a mistake will take place. Therefore, the Z3 SMT solver is stronger in guaranteeing generating counterexample than the the SAT solver because the Z3 SMT solver proves that the formula is valid in general, while Alloy does not show that the formula is valid in general.

So, the use of a finite scope makes the analysis decidable but also incomplete. If an instance satisfying the formula cannot be found within a finite scope, that does not imply that the model is unsatisfiable. An instance may be found if the scope is increased. Also, the lack of a counterexample for an assertion does not imply that the asserted property holds in a larger scope.

Although a SAT solver has an advantage in minimizing the problem by bounding the number of scopes, we see that the number of variables and the verification time in SAT solver have increased with respect to increasing the number of scopes. That may lead to making the SMT solver faster and detect counterexample earlier.

- Counterexample

To compare the counterexamples, when Alloy Analyser generates a counterexample in analysing the ATM system, it was based on a finite number of domains, while in Z3 was based on infinite scope. Both Alloy and Z3 provided us with the result as a model. However, Alloy has an internal representation of a model which is logical as a mapping between values and variables. Because of the difficulty of reading the logical internal representation, Alloy gave us the ability to visualize the mappings between variables and values. As a result it helps us looking at counterexample. On the othe hand, Z3 presents the counterexample as a formula. It was difficult for us to detect the flows in the Z3 counterexample compared to Alloy because Alloy presents the detected counterexample using graphic capabilities.

- Abilities

To compare the abilities of Alloy and Z3, Alloy is a model finder and not a prover. It *shows* why there is a counterexample. Thus it cannot prove theorems without additional help such as the Z3 theorem prover. On the other hand, Z3 can *prove* why there is a counterexample. However, the Alloy model finder is still useful if the goal does not need to waste time trying to prove a theorem as a model finder can generate a counterexample for us automatically.

- Quantifiers

In our model both the Alloy Analyser and the SMT solver use quantifiers. However, the difference between them lies in the *analysis* not in the *language* itself.

All in Alloy means binding free variables for a specific domain. The specific domain of bounded scope just in the Analysis not in the language because *All* is for all variables in a type and type is bounded. On the other hand, ∀ in Z3 means the specific domain is not bounded in either the Analysis nor in the language. Thus, ∀ in Alloy has limited power in analysis and Z3 has absolute power in analysis. *Some* in Alloy and ∃ in Z3 are the same in the language and in the analysis.

So, we conclude that Alloy is better than Z3 in providing the counterexample in visualization saving time and effort for non experts to understand the problem. Z3 cannot do that. Z3 is better than Alloy in searching for the counterexample in general and proving why it is exists, while Alloy searches for the counterexample in a limited scope and just models the counterexample with no proof as why it is exist. Searching for a counterexample in a limited scope may lead to missing an instance that may cause a counterexample in the lager scope.

Z3 is faster than Alloy in providing the results even though it is unbounded and that may be due to the technique of Alloy which takes a longer process when the Alloy Analyser is using SAT4J solver. It provides a model after doing inclusive verification for all models until a bounded number of instances for a *prop1* assertion.

If the SAT4J solver found a model which means verification fails, Alloy adds the resulting counterexample as a constraint to guide the search for the next solution, and starts again. When verification succeeds, the solution is represented as a higher-order quantification, and can be returned to the user. As the scope increase, more time spent to analyse the model.

In addition, the technique of the SAT solver in Alloy and the Z3 SMT solver is to negate the assertion, and look for a model if there exist a counterexample. The Alloy Analyser uses the negation of the assertion automatically, while in Z3 we had to write that explicitly as seen in Appendix (C.4) line (2). If we forgot to write the negation, there will be a mistake in the result and thus there is a defect in the comparison.

Alloy uses relations, while Z3 uses functions. Relations are constrained by multiplicities and any mistakes in determining the multiplicity key word may lead to an inconsistent model which leads to making Alloy a sensitive language.

Chapter 5

Problem Specification and Case Study, and Multichannel Security Protocol Modelling and Analysis

In this chapter we model and analyse our solution protocol for transmitting data securely in WLANs using multichannel protocol. Our protocol is separated into two levels, the first for sending a mix of letters over a channel, and the second for sending indices over a different channel.

This chapter presents how the protocol works to detect security flaws for data transmission over a multichannel in a wireless network, depending on analysing the message into two unreadable messages (letters and indices) and sending each message over a separate channel, taking into account the changing MAC address for each channel.

5.1 Proposed Solution for The Case Study

The model depends on an analysis and combination technique for data transmission between the two hosts over a multichannel. Testing the model includes checking the security of the data transmission over a multichannel in a secure and in an insecure scope, which is in the presence of a MitM. It then provides a report with the results of whether the protocol achieved security for data transmission (unreadable).

We assume that there is just one MitM between two hosts that are transmitting an email message over two wireless channels taking into account changing MAC address for each channel using software like "Technitium MAC Address Changer".

When the MitM needs to intercept a communication between two parties, the MAC address is the key to accessing the data inside the packets transferred between them. So, when the MAC address is already known the MitM can hack the packets that are sent to or received from the specific host directly, or the MitM runs his tools to list all of the hosts' MAC addresses around the area that are sending and receiving data wirelessly [69].

The reason for analysing the message in two parts and submitting each part over a channel is to mislead the MitM. Once one channel has been intercepted the MitM will not get the message because the intercepted packet either includes indices or letters and getting a readable message requires both parts, as seen in Figure 5.2.

Changing the MAC address is part of our security strategy that intends to mislead MitM because each sender's wireless card (Network Interface Card (NIC)) has a unique MAC address provided with every packet to denote the sender's source even with a different wireless connection. NIC is used to connect a computer to an Ethernet network [117].

A MAC address is represented by six groups of two hexadecimal digits (0-9, A-F). Each group of two hexadecimal numbers is separated by the hyphen (-) or by the colon (:). For example: 02-43-40-99-79-Az, or 02:43:40:99:79:Az.

When the MitM attempts to hack a connection and intercept its packets, he is focused on displaying all the MAC addresses. Once he finds two MAC addresses that have the same hexadecimal values, he will recognise that these two packets have been sent from one sender. Matching the two packets will then provide readable data. So, changing the MAC address is essential in our case study.

The message that needs to be sent is analysed at the client side into a changeable array of letters and is submitted over one channel with indices submitted over another channel. This takes into account the changing MAC addresses to make it appear that this data was issued from different clients.

A changeable array of letters means that, the array contains 26 unordered alphabetical letters. The order of these letters is changeable randomly in each sending message.

Each index points to the location of the correct letter of the message, and by combining and matching each index with its index on the server side the receiver will get the message. The changeable array of letters means that each time the sender sends an email the arrangement of letters will be different, as they are chosen randomly.

The model consists of two systems:

- For the sender to analyse (decompose) the message into an array of letters and indices.

- For the receiver to receive the letters and indices and match the letters with the indices to get the original message.

The first channel contains the original MAC addresses. The second channel will contain the fake MAC addresses. Because the original MAC address will be used in the first connection to submit indices and the fake MAC address will be generated and used in the second connection to submit letters, the server's system recognizes which channel contains the original MAC addresses because the reply message should be sent to the correct sender known using the original MAC address.

In more detail regarding changing the MAC addresses, the system passes through two levels of connections while sending data using the following steps:

- Level 1: Sending indices
  - Connecting to the Internet via wireless A using MAC1 address.
  - Sending indices.
  - Disconnecting from the Internet.
- Level 2: Sending letters
  - Changing MAC1 address.
  - Generating new address, MAC2 address.
  - Connecting to the Internet via wireless B using MAC2 address.
  - Sending an array of mix of letters.
  - Disconnecting from the Internet.
  - Retrieving old MAC1 address.

5.2 Example in Both Single Channel and Multichannel

This section, shows a worked example for different protocols in transmitting data over single channel followed by multichannel, and the success in achieving security.

5.2.1 Single Channel

Transmitting data over a single channel means that there are two hosts - a client and a server who would like to transmit a message over a channel using their computers. Each host has a unique MAC address. A client connects to an ISP. When a message is transmitted from the client's computer, the MAC address of the client's computer is identified as the source. Suppose a sender needs to submit message "computer" to the server. In the existence of a MitM, due to transmitting the message over a single channel, and by finding out the MAC address of the sender, it is easy for the MitM to intercept the channel that has already been opened for the known MAC address. As a result, the message that was transmitted is vulnerable to eavesdropping as seen in Figure 5.1.

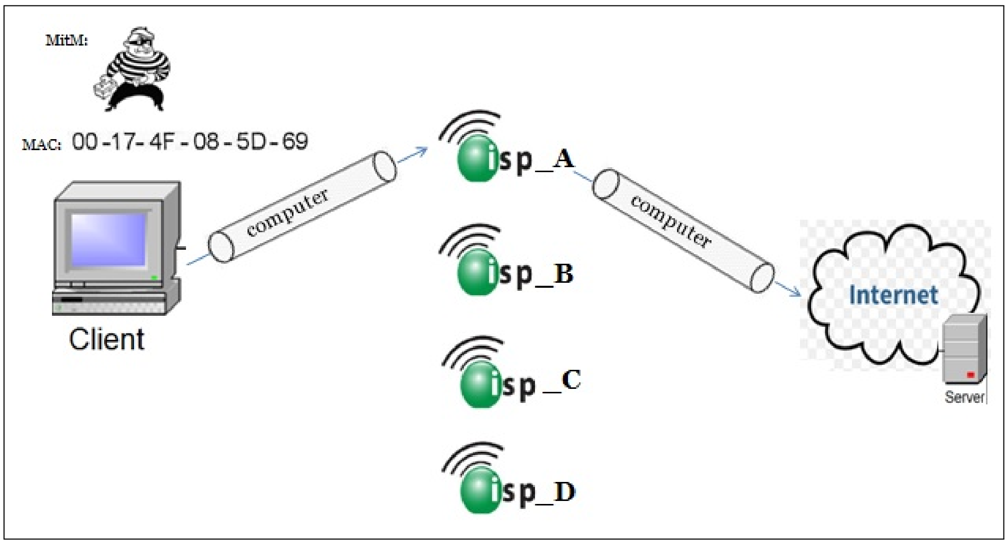


Figure 5.1: MitM Intercepts Data Transmission Between Two Parties Over Single Channel

5.2.2 Multichannel

For transmitting data over a multichannel each host has two different MAC addresses: one is original and the other is fake. The original MAC address will be used in the first connection to submit indices, while the fake MAC address will be generated and used in the second connection to submit letters. The client connects to two different ISPs. Suppose a sender needs to submit message "computer" to the server. In the existence of one MitM, due to transmitting the message over multichannel, it is not easy for the MitM to intercept both channels simultaneously because when one channel is connected, the other is not connected. Also it is not easy for the MitM to recognize that both channels belong to the same source as both of them identified by different MAC address. However, by finding out a MAC address of the sender and intercepting a channel that has already been opened for the known MAC address meaning that the message that was transmitted is now invulnerable to be eavesdropped as seen in Figure 5.2. The reason for that is, the intercepted channel either contains a message of 26 random letters or message of numbers which means the attackers got an unreadable message.

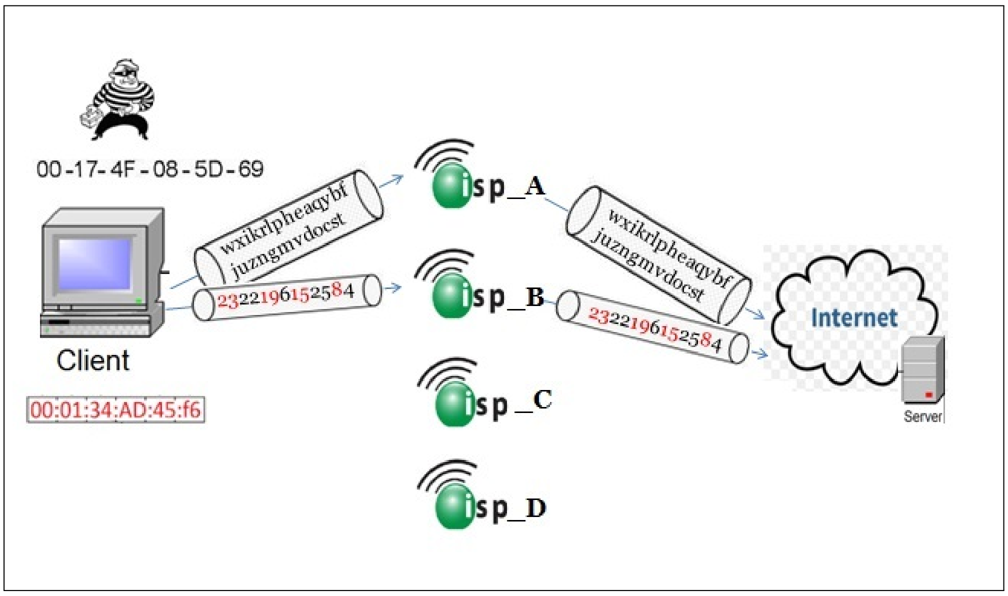


Figure 5.2: MitM Intercepts Data Transmission Between Two Parties Over Multichannel

- **Example for send a message over multichannel**

Each sender and receiver has a system. The sender's system works on sending a message following its steps below, while receiver's system works on receiving the message following its steps below. When a sender needs to submit the message "computer", their system does that in four steps. The first step is when a sender connects to a wireles network choosing an Internet Service Provider (ISP_A) using the original MAC address, the sender then opens the system dialogue (application form) as seen in Figure 5.3 to start sending their message to the receiver. The application form includes approved accounts (from email address /to email address); the email's password; the subject of the message; a MAC address box to type the generated fake MAC address; an adaptor to select an ISP; the body of the message to type a message; a *sendIndexs* icon which is responsible for submitting the indices file; a *ReConnect* icon which is responsible to re-connect to the second ISP as the system will disconnect from the first ISP directly after pressing *sendIndexs* icon; a *sendLetters* icon which is responsible for submitting the letters file; and *Exit* icon which is responsible to exit from the system.

In the second step, after the client's account is approved (client's email and password are authenticated). The sender's system will work in three parts. The first part is generating an array containing 26 unordered alphabetical letters such as *wxikrlpheaqybfjuzngmvdocst*. Then it opens an external file for example *letters.txt* to write each letter in the array in a line in the file.

The second part is finding the index of each letter of the message "computer" that has been written in the body of the message. Finding the index of each letter according to the array of letters to get an array of numbers containing the indices such as *2322190615250804*. After that the sender's system opens an external file for example *indices.txt* to write each number in the array in a line in the file. Here the index '23' points to letter 'c', the index '22' points to letter 'o', the index '19' points to letter 'm', the index '06' points to letter 'p', the index '15' points to letter 'u', the index '25' points to letter 't', the index '08' points to letter 'e', and the index '04' points to letter 'r'. Then decomposing the list into pairs gives:

23 ->c; 22 ->o; 19 ->m; 06 ->p; 15 ->u; 25 ->t; 08 ->e; 04 ->r

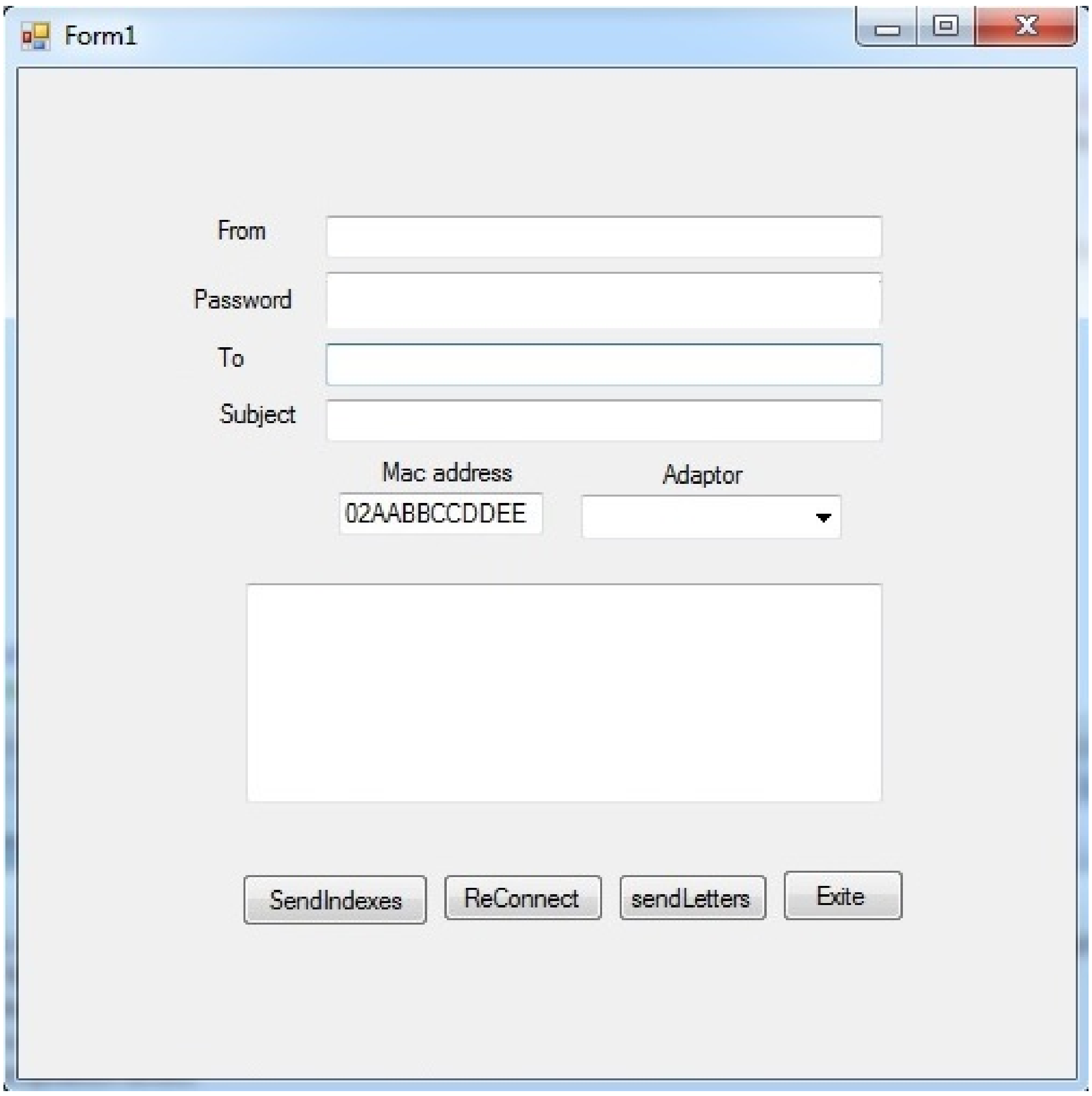


Figure 5.3: Application Form For Sending A Message

The third part is when sender's and receiver's account details and client email's password are stored in a different external file to be used in the second connection. The external file is *account.txt* file. These three files are automatically generated in the same directory of the protocol system.

After that, the sender's system directly opens the file that stores indices, reads the file, and replaces the body of the message with the content of the file in the same format as lines. After that the message of indices is transmitted over the first channel which is addressed by the original MAC address such as (00-17-4F-

08-5D-69) and connected to the first ISP such as ISP_A. After that the system will disconnect from the ISP_A, changing the original MAC address to a fake one such as (00-01-34-AD-45-F6), and reconnect to a different Internet Service Provider (ISP_*B*) from the adapter.

In the third step, when the client clicks on *sendLetters* icon the sender's system directly opens the file that stores letters, reads the file, and replaces the body of the message with the content of the file in the same format as lines. Moreover, the sender's system extracts the approved accounts (from email address /to email address); and client email's password from the *account.txt* file. After that the message of letters is transmitted over the second channel which is addressed by the fake MAC address such as (00-01-34-AD-45-F6) and connected to the second ISP such as ISP_B. At the end, the MAC address will be changed back to the original MAC address.

Finally, when the receiver's system receives these two channels which include letters and indices, in the first step the system writes two messages line by line in different external files. In the second step it matches each index with its location in the array of letter to get the message "computer". In the last step presents the readable message in their mailbox to be read at the other end.

5.3 Conclusion

In summary, we will investigate whether our multichannel protocol may provide more security compared with a single channel because it depends on dividing the message into two unreadable messages where each message is submitted over a different channel. We change the MAC address in the second submission and if one of the channels is intercepted, the MitM may not read the message.

Chapter 6

Multichannel Security Protocol Modelling Using Alloy

6.1 Overall Framework

This chapter presents the Alloy framework for modelling and checking the validity of properties of the provided protocols in two major analyses. The first major analysis is called predicate-running. It is applied to Alloy problems with a predicate and results in a visualization. If satisfying structures (called instances) exist, then the model is consistent; otherwise the model is inconsistent. This is required before checking the assertion.

The second major analysis form is called assertion-checking. It is applied to Alloy problems with an Alloy specification taking into account the properties that have been constrained as facts. It results in either generating a counterexample if the properties do not satisfy the requirements, or otherwise no counterexample has been generated in a limited scope.

To model properties of our protocols in these two major analyses using Alloy, we follow the same methodology of making the ATM model in chapter 3.

6.2 Approach

Our framework provides two kinds of protocols in different aspects. The first protocol is for data transmission over a single channel. The second protocol is for data transmission over a multichannel. Both protocols will be modelled in secure and in insecure scopes with the presence of MitM.

6.3 Motivation

The goal of building a model for such protocols as a case study is to describe the properties of the aspects of security protocols (but not the entire system), constrain the properties to exclude holes which lead to malfunction, and check if the properties satisfy the system requirement or not. We describe the procedure of a protocol that has been used to transmit data securely, add some constraints about how data transmitted securely should behave, and then check to see if any status of each aspect gets to its destination securely or, has been intercepted by a MitM to get a readable message. The modelling tool of Alloy in this case would then either say "this property always holds for the scope of X" or "this property does not always hold in the scope of Y, and here is a counterexample".

6.4 Protocol of Transmitting Data Over Single Channel Case Study

In this section we represent the first case study protocol in data transmission over a single-channel without the presence of MitM, followed by data transmission over a single-channel in the presence of MitM. This enables us to study how Alloy can be aligned with the model as it begins with a simple model and gradually introduce complexities. In addition, we also study how Alloy could model a secure and insecure protocol providing instances that satisfies the constraints if the model is consistent and then providing a counterexample if one exists. Consequently, in the case of providing a counterexample, we can understand the holes in the security of the protocol once system properties do not satisfy the system requirements in limited scope, and fix these holes before implementing our decided protocol.

6.4.1 First Protocol: Describing Data Transmitting Over Single Channel WLANs in Secure / Insecure Scope Model

The first protocol is modelled in aspects of secure and insecure scope. The first aspect is very simple and requires two hosts to be communicating regularly using one ISP and identified using their unique MAC address. Also there is no MitM be intercept data transmitting over a single channel between the two parties. As a result, time is not delayed which leads to data that has been received having not been read or modified. Thus, data that has received by the server equals data that has sent by the client.

The second aspect is developed from the first aspect by inserting a MitM between the two parties. As a result, time is delayed which leads to data that has been received possibly having being read or modified. Thus, data that has been received by a server may not equals to data that has sent by client.

6.4.1.1 Protocol Model Properties and Requirements

For the first aspect: **the first property** is that there is no MitM; **the second property** is opening one channel to exchange data between two hosts (client, server); **the third property** is the time for sending and receiving data over the opened channel has not been delayed which implies that received data by the server equals sent data by the client; **the fourth property** is that MAC addresses have to be unique. The second aspect has the same properties as first aspect except **the first property** which requires that there is a MitM, The protocol requirements for both aspects are that equalities of sending and receiving time implies that the data that has been received equals the data that has been sent. Thus, the submitted message has neither been modified nor eavesdropped.

Modelling this protocol passes through one status when two parties are connecting to their selected ISP. Thus opening a single channel between them to exchange data between them illustrates that sending data takes place at the sending time, while receiving data takes place over the same channel at the receiving time.

6.4.1.2 An Alloy Specification of the First Protocol

Abstractly seen in Appendix (C), an Alloy model M describes the structures of the problem i.e., signatures, Appendix (C.1), and fields declarations which shows the relation of the signatures including their implicit constraints, Appendix (C.2) , a set F of facts which restrict the properties of the model, Appendix (C.3), a predicate which shows how the model behaves, Appendix (C.4), and an assertion A describing some properties about the problem, Appendix (C.5). An Alloy problem is correct if any Alloy instance that conforms with the model M and satisfies the facts in F, satisfies the assertion A as well.

Modelling the protocol begins by determining and identifying the Alloy specification: the main objects (signatures), relations, and predicates for the interaction protocol as seen in Appendices (C.1, C.2, C.4).

- Signatures

In our model in Appendix (C.1), signatures in Alloy represent a sets of *atoms*, each *signature* represents a set of *atoms* with a maximum number of three and has no *relation* signs as (->). For example, object *Time* represents a set of *atoms* which are $\{time_0, time_1, time_2\}$ by default (Line 1).

The *signatures* in (Lines 1,2,3,4,5,7,8) are called *top-level signatures* because they do not extend other signatures and they will be implicitly disjoint until they show the relation (*field*) between them. Each *top-level signature* has a maximum set of three *atoms* by default with no relation between them.

The type hierarchy of our Alloy specification consists of seven top-level types: *Time* which is used as a timer which calculates when data has been sent and when data has been received (Line 1). In this protocol, Time is a parameter. We supposed that sending and receiving a message takes place at the same time in the secure communication. There are two cases. In the first, we assume that there has been no interception and no delay in the time (sending and receiving a message takes place at the same time). As a result data has been received equals to data has been sent.

In the second case, if we assume that, if there is an interception, we supposed that sending and receiving a message takes place at different times in insecure communication. This denotes that there is an interception and a delay in time happened (sending and receiving a message takes place at the different time). As a result, data has been received is not equal to the data that has been sent.

For example, if sending a message took place at time `"0"`, receiving a message we supposed to be at time `"0"` in secure communication. So, we see here the time was not delayed. However, if sending a message took place at time `"0"`, but receiving a message was at time `"1"` in insecure communication. So, we see here a time delayed.

ISP and *Channel* are required to show how two parties exchange data over a single channel which is opened after two parties are connected to the selected ISP in the first status (Lines 2,3); *Data* is used as a measure of whether a modification has been done or not. This is the main object the protocol uses to be modelled as an exchanging tool between parties (Lines 4); *ConnectionStatus* is required to express each status individually (Lines 5); *MAC* address is used to express each visitor to the connection (Line 7); *Communication_Status* is the main part in the model which shows how signatures are related to each other to visualize each status under the restrictions applied in the predicates and facts and checked in the assertion (Line 8).

- Abstraction

The top-level type *ConnectionStatus* in (Line 5) is the basic type. It is labelled with the keyword **abstract** and has no elements except those which are extended which constrains each element to belong to one of its extending subtypes. However, we have just one subtype *Connection_And_Exchanging_Data* (Lines 6), and the idea of the abstract and the extension will be clear when the protocol will be developed to increase the status information. This subtype is for representing the status of the connection and exchanged data between two parties.

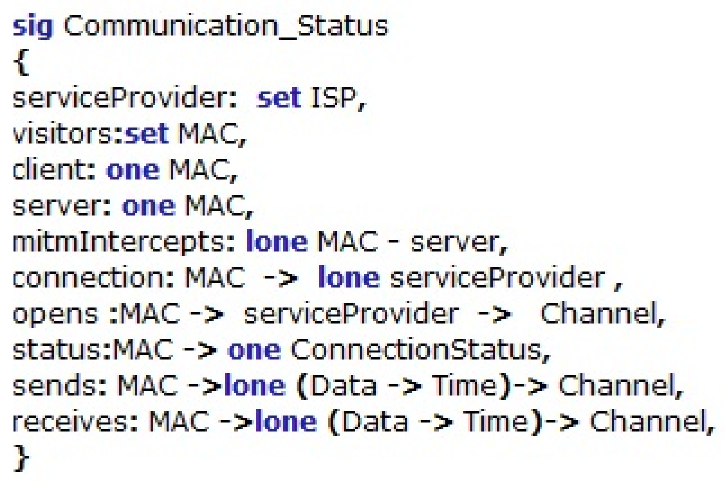
- Extension

Type *ConnectionStatus* in (Line 5) is extended in Alloy using the keyword **extends**. The rest of the top-level types are neither abstract nor extended.

Extending subtype *Connection_And_Exchanging_Data* (Line 6) is restricted to be a singleton using the keyword *one*.

- Relation Declarations, and Multiplicities

Table 6.1 shows the relations and their types.



```
sig Communication_Status
{
serviceProvider:  set ISP,
visitors:set MAC,
client: one MAC,
server: one MAC,
mitmIntercepts: lone MAC - server,
connection: MAC  ->  lone serviceProvider ,
opens :MAC ->  serviceProvider  ->   Channel,
status:MAC -> one ConnectionStatus,
sends: MAC ->lone (Data -> Time)-> Channel,
receives: MAC ->lone (Data -> Time)-> Channel,
}
```

Relations fields name	Related atoms	(Type/size) of relation	No. of atom per tuple
serviceProvider	(Communication$status, ISP$)	Binary	2 atoms
visitors	(Communication$status, MAC$)	Binary	2 atoms
client	(Communication$status, MAC$)	Binary	2 atoms
server	(Communication$status, MAC$)	Binary	2 atoms
mitm	(Communication$status, MAC$)	Binary	2 atoms
connection	(Communication$status, MAC$, serviceProvider)	Ternary	3 atoms
opens	(Communication$status, MAC$, serviceProvider, Channel)	>Ternary	4 atoms
status	(Communication$status, MAC$, ConnectionStatus)	Ternary	3 atoms
sends	(Communication$status, MAC$, Data, Time, Channel)	>Ternary	5 atoms
receives	(Communication$status, MAC$, Data, Time, Channel)	>Ternary	5 atoms

Table 6.1: Relations And Their Types

The fields are represented in Apppendix (D.2).

- The field *serviceProvider* represents all the *ISPs* that exist in the *Communication_Status* and declares a binary relation of type *Communication_Status ->ISP* which maps each element of *Communication_Status* to *set* (arbitrary number/any number) of elements of *ISP*.

  The binary relation *serviceProvider* ⊆ *Communication_Status* × *ISP*. (Lines 8 Appendix C.1, Line 1 Appendix C.2) declares a relation *serviceProvider* with domain *Communication_Status* and range *ISP*. The declaration of *serviceProvider* contains the multiplicity annotation *set* which makes *serviceProvider* a function: a binary relation that associates every *Communication_Status* with a set of *ISP*. For every element cs in *Communication_Status*, the keyword *set* before *ISP* constraints the term *cs.serviceProvider* to be *set*. The declaration of

  $$serviceProvider\; in\; Communication\_Status -> ISP$$

  can thus be expressed as:

  $$\forall\; cs : Communication\_Status |\; cs.serviceProvider \;\subseteq\; (\textbf{set}\; ISP)$$

  We can express the multiplicity constraints using formula as:

  $$all\; cs : Communication\_Status |\; \textbf{set}\; cs.serviceProvider$$

  The multiplicity keyword *set* makes the declaration hold when the relation *serviceProvider* is a function from *Communication_Status* to *ISP*.

- The field *visitors* represents all the *MACs* that exist in the *Communication_Status* and declares a binary relation of type *Communication_Status ->MAC* which maps each element of *Communication_Status* to *set* of elements of *MAC*.

  The binary relation *visitors* ⊆ *Communication_Status* × *MAC*. (Lines 8 Appendix C.1, Line 2 Appendix C.2) declares a relation *visitors* with domain *Communication_Status* and range *MAC*. The declaration of *visitors* contains the multiplicity annotation *set* which makes *visitors* a function: a binary relation that associates every *Communication_Status* with a set of *MAC*. For every element cs in *Communication_Status*, the keyword *set* before *MAC* constraints the term *cs.visitors* to be *set*. The declaration of

$$visitors\ in\ Communication\_Status -> MAC$$

can be expressed as:

$$\forall\ cs : Communication\_Status|\ cs.visitors\ \subseteq\ (\textbf{set}\ MAC)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|\ \textbf{set}\ cs.visitors$$

The multiplicity keyword *set* makes the declaration hold when the relation *visitors* is a function from *Communication_Status* to *MAC*.

- The field *client* represents the *MAC* that exists as a client in the *Communication_Status*, and declares a binary relation of type *Communication_Status* ->*MAC* which maps each element of *Communication_Status* to *one* element of *MAC*.

  The binary relation *client* ⊆ *Communication_Status* × *MAC*. (Lines 8 Appendix C.1, Line 3 Appendix C.2) declares a relation *client* with domain *Communication_Status* and range *MAC*. The declaration of *client* contains the multiplicity annotation *one* which makes *client* a total function: a binary relation that associates every *Communication_Status* with one *MAC*. For every element cs in *Communication_Status*, the keyword *one* before *MAC* constraints the term *cs.client* to be *one*. The declaration of

  $$client\ in\ Communication\_Status -> MAC$$

  can be expressed as:

  $$\forall\ cs : Communication\_Status|\ cs.client\ \subseteq\ (\textbf{one}\ MAC)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status |\ \textbf{one}\ cs.client$$

The multiplicity keyword *one* makes the declaration hold when the relation *client* is a total function from *Communication_Status* to *MAC*.

- The field *server* represents the *MAC* that exists as a server in the *Communication_Status*, and declares a binary relation of type *Communication_Status ->MAC* which maps each element of *Communication_Status* to *one* element of *MAC*.

  The binary relation *server* $\subseteq$ *Communication_Status* $\times$ *MAC*. (Lines 8 Appendix C.1, Line 4 Appendix C.2) declares a relation *server* with domain *Communication_Status* and range *MAC*. The declaration of *server* contains the multiplicity annotation *one* which makes *server* a total function: a binary relation that associates every *Communication_Status* with one *MAC*. For every element cs in *Communication_Status*, the keyword *one* before *MAC* constraints the term *cs.server* to be *one*. The declaration of

$$server\ in\ Communication\_Status -> MAC$$

  can be expressed as:

$$\forall\ cs : Communication\_Status |\ cs.server\ \subseteq\ (\textbf{one}\ MAC)$$

  We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status |\ \textbf{one}\ cs.server$$

  The multiplicity keyword *one* makes the declaration hold when the relation *server* is a total function from *Communication_Status* to *MAC*.

- The field *mitmIntercepts* represents the *MAC* that exists as a MitM in the *Communication_Status*, and declares a binary relation of type *Communication_Status ->MAC* which maps each element of *Communication_Status* to *lone* element of *MAC*, except *server*.

The binary relation *mitmIntercepts* ⊆ *Communication_Status* × *MAC*. (Lines 8 Appendix C.1, Line 5 Appendix C.2) declares a relation *mitmIntercepts* with domain *Communication_Status* and range *MAC*. The declaration of *mitmIntercepts* contains the multiplicity annotation *lone* which makes *mitmIntercepts* a partial function: a binary relation that associates every *Communication_Status* with (*lone*) at most one of *MAC*. For every element cs in *Communication_Status*, the keyword *lone* before *MAC* constraints the term *cs.mitmIntercepts* to be *lone*. The declaration of

$$mitmIntercepts\ in\ Communication\_Status -> MAC$$

can be expressed as:

$$\forall\ cs : Communication\_Status|\ cs.mitmIntercepts\ \subseteq\ (\textbf{lone}\ MAC)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|\ \textbf{lone}\ cs.mitmIntercepts$$

The multiplicity keyword *lone* makes the declaration hold when the relation *mitmIntercepts* is a partial function from *Communication_Status* to *MAC*.

- The field *connection* represents the *MAC* that connects to the *ISP*, and declares a ternary relation of type *Communication_Status ->MAC ->ISP* which maps each element of *Communication_Status* to an element of *MAC*; and each element of *MAC* to *lone* element of *cs.serviceProvider*; each element of *cs.serviceProvider* to an element of *MAC*; each element of *MAC* to an element of *Communication_Status*.

  The ternary relation *connection* ⊆ *Communication_Status* × *MAC* × *ISP*. (Line 8 Appendix C.1, Line 6 Appendix C.2) declares a relation *connection* with domain *Communication_Status* and range *ISP*. The declaration of *connection* contains the multiplicity annotation *lone* which makes *connection* a partial function: a ternary relation that associates every *Communica-*

tion_Status with *lone MAC.connection*. For every element cs in *Communication_Status* and for every element mac in *MAC*, the keyword *lone* before *cs.serviceProvider* constraints the term *mac.(cs.connection)* to be *lone*. The declaration of

$$connection\ in\ Communication\_Status -{>}MAC -{>}\ serviceProvider$$

can be expressed as:

$$\forall\ cs : Communication\_Status,\ mac : Mac|\ mac.(cs.connection)\ \subseteq$$

$$(\textbf{lone}\ cs.serviceProvider)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|cs.connection\ \&\&$$

$$all\ cs : Communication\_Status,\ mac : Mac|lone\ mac.(cs.connection)$$

The multiplicity keyword *lone* makes the declaration hold when the relation *connection* is a partial function from *mac.(cs.connection)* to *cs.serviceProvider*.

When a field of the same signature appears in another field's declaration, it is interpreted in the context of that signature. Field *serviceProvider* of the same signature *Communication_Status* appears in another field's declaration *connection*. Field *connection* is added to capture that the *lone mac.(cs.connection)* might be *connection*. This declaration says that for every element cs in *Communication_Status*, and mac in MAC, *mac.(cs.connection)* is a subset of cs.serviceProvider because field *serviceProvider* of the same signature *Communication_Status* appears in another field's declaration *connection*.

- The field *opens* represents the *Channel* that has been opened between a *MAC* and an *ISP*, and declares a more than ternary relation (four) of type *Communication_Status ->MAC ->ISP ->Channel* which maps each element of *Communication_Status* to an element of *MAC*; each element of *MAC* to an element of *cs.serviceProvider*; each element of *cs.serviceProvider* to an element of *Channel*; each element of *Channel* to an element of *cs.serviceProvider*; each element of *cs.serviceProvider* to an element of *MAC*; each element of *MAC* to an element of *Communication_Status*.

The (four) relation *opens* ⊆ *Communication_Status* × *MAC* × *ISP* × *Channel*. (Line 8 Appendix C.1, Line 7 Appendix C.2) declares a relation *opens* with domain *Communication_Status* and range *Channel*. The declaration of *opens* is not prefixed with a multiplicity keyword. The declaration of

$$opens\, in\, Communication\_Status{-}{>}MAC{-}{>}\, serviceProvider\, {-}{>}\, Channel$$

can be expressed as:

$$\forall\, cs : Communication\_Status,\ mac : Mac|\ mac.(cs.opens)$$

$$\subseteq\ cs.serviceProvider)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|\ cs.opens\ \&\&$$

$$all\ cs : Communication\_Status,\ mac : Mac|mac.(cs.opens)\ \&\&$$

$$all\ cs : Communication\_Status,\ mac : Mac,\ isp : cs.serviceProvider|$$

$$(isp).mac.(cs.opens)$$

This declaration states that every element cs in *Communication_Status*, and mac in MAC, mac.(cs.opens) is a subset of cs.serviceProvider because field *serviceProvider* of the same signature *Communication_Status* appears in another field's declaration *opens*.

- The field *status* represents the *MAC* statuses, and declares a ternary relation of type *Communication_Status ->MAC ->ConnectionStatus* which maps each element of *Communication_Status* to *one* element of *Connection-Status*.

The binary relation *status* $\subseteq$ *Communication_Status* $\times$ *ConnectionStatus*. (Lines 8 Appendix C.1, Line 8 Appendix C.2) declares a relation *status* with domain *Communication_Status* and range *ConnectionStatus*. The declaration of *status* contains the multiplicity annotation *one* which makes *status* a total function: a ternary relation that associates every *Communication_Status* with exactly *one* of *ConnectionStatus*. For every element cs in *Communication_Status*, the keyword *one* before *ConnectionStatus* constraints the term *cs.status* to be *one*. The declaration of

$$status\ in\ Communication\_Status -> ConnectionStatus$$

can be expressed as:

$$\forall\ cs : Communication\_Status|\ cs.status\ \subseteq\ (\textbf{one}\ ConnectionStatus)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|\ \textbf{one}\ cs.status$$

The multiplicity keyword *one* makes the declaration hold when the relation *status* is a total function from *Communication_Status* to *ConnectionStatus*.

- The field *sends* represents the *Data* that has been sent each *Time* from a *MAC* over a *Channel*. It declares a more than ternary relation (five) of type *Communication_Status ->MAC ->Data ->Time ->Channel* which maps each element of *Communication_Status* to an element of *MAC*; each element of *MAC* to *lone* element of *Data* that is mapped to an element of *Time*; each element of *Data* that is mapped to an element of *Time* to an element of *Channel*; each element of *Channel* to an element of *Data* that is mapped to an element of *Time*; each element of *Data* that is mapped to an element of *Time* to an element of *MAC*; each element of *MAC* to an element of *Communication_Status*.

The (five) relation *sends* $\subseteq$ *Communication_Status* $\times$ *MAC* $\times$ *Data* $\times$ *Time* $\times$ *Channel.* (Line 9 Appendix C.1, Line 9 Appendix C.2) declares a relation *sends* with domain *Communication_Status* and range *Channel.* The declaration of *sends* contains the multiplicity annotation *lone* which makes *sends* a partial function: a five relation that associates every *Communication_Status* and every *MAC* with *lone* associated *data with time.* For every element cs in *Communication_Status*, and for every element mac in *MAC*, the keyword *lone* before *Data with Time* constraints the term *mac.(cs.sends)* to be *lone.* The declaration of

$$sends\ in\ Communication\_Status -> MAC ->$$

$$(Data\ \ -> Time)\ \ -> Channel$$

can be expressed as:

$$\forall\ cs : Communication\_Status,\ mac : MAC,\ d : Data,\ t : Time|$$

$$(t.d).mac.(cs.sends)\ \subseteq\ (\textbf{lone}\ Channel)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|\ cs.sends\ \&\&$$

$$all\ cs : Communication\_Status,\ mac : Mac|\ mac.(cs.sends)\ \&\&$$

$$all\ cs : Communication\_Status,\ mac : Mac,\ d : Data,\ t : Time|$$

$$lone\ (t.d).mac.(cs.sends)$$

The multiplicity keyword *lone* makes the declaration hold when the relation *sends* is a partial function from *Communication_Status* to *Cahnnel.*

- The field *receives* represents the *Data* that has been received from a *MAC* each *Time* over a *Channel*. It declares a more than ternary relation (five) of type *Communication_Status ->MAC ->Data ->Time ->Channel* which maps each element of *Communication_Status* to an element of *MAC*; each element of *MAC* to *lone* element of *Data* that is mapped to an element of *Time*; each element of *Data* that is mapped to an element of *Time* to an element of *Channel*; each element of *Channel* to an element of *Data* that is mapped to an element of *Time*; each element of *Data* that is mapped to an element of *Time* to an element of *MAC*; each element of *MAC* to an element of *Communication_Status*.

  The (five) relation *receives* $\subseteq$ *Communication_Status* $\times$ *MAC* $\times$ *Data* $\times$ *Time* $\times$ Channel. (Line 9 Appendix C.1, Line 9 Appendix C.2) declares a relation *receives* with domain *Communication_Status* and range *Channel*. The declaration of *receives* contains the multiplicity annotation *lone* which makes *receives* a partial function: a five relation that associates every *Communication_Status* and every *MAC* with *lone* associated *data with time*. For every element cs in *Communication_Status*, and for every element mac in *MAC*, the keyword *lone* before *Data with Time* constraints the term *mac.(cs.receives)* to be *lone*. The declaration of

  $$receives\ in\ Communication\_Status -> MAC ->$$

  $$(Data\ -> Time)\ -> Channel$$

  can be expressed as:

  $$\forall\ cs : Communication\_Status,\ mac : MAC,\ d : Data,\ t : Time|$$

  $$(t.d).mac.(cs.receives)\ \subseteq\ (\textbf{lone}\ Channel)$$

  We can express the multiplicity constraints as:

  $$all\ cs : Communication\_Status|\ cs.receives\ \&\&$$

$$all\ cs : Communication\_Status,\ mac : Mac|\ mac.(cs.receives)\ \&\&$$

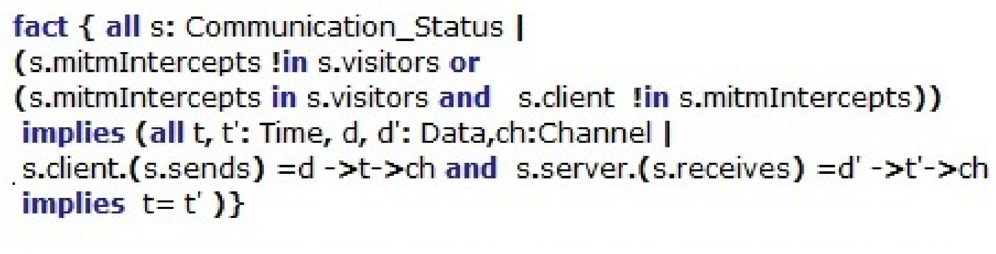
$$all\ cs : Communication\_Status,\ mac : Mac,\ d : Data,\ t : Time|$$

$$lone\ (t.d).mac.(cs.receives)$$

The multiplicity keyword *lone* makes the declaration hold when the relation *receives* is a partial function from *Communication_Status* to *Cahnnel.*

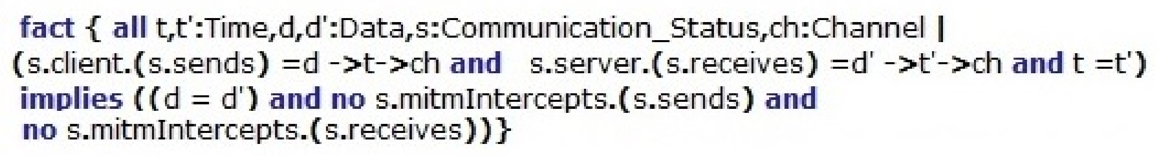
- **Fact**

The three facts in our model Appendix (C.3) representing a constraint on formulas. The first fact (Line 1) states that for all *Communication_Status*, such that a MitM does not exist in the set of *visitors* for the first aspect, or for the second aspect, a MitM exists as a visitor (belongs to the visitors), but he does not intercept the client because the client does not belong to the MitM. This implies for all time *t,t'* which represents sending and receiving time respectively, and for all data *d,d'* which represent sending and receiving data respectively, and for all channels *ch1* which represents that there is a channel to exchange data over, such that what the client sent during *Communication_Status* is data *d* in time *t* over a channel *ch1* and what the server receives during *Communication_Status* is data *d'* in time *t'* over the same channel *ch1* implies that sending and receiving time are equals which means there is no time delay occurred.



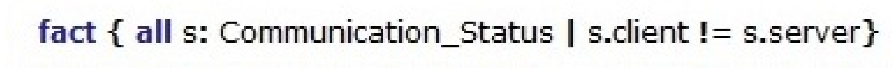
```
fact { all s: Communication_Status |
(s.mitmIntercepts !in s.visitors or
(s.mitmIntercepts in s.visitors and  s.client !in s.mitmIntercepts))
 implies (all t, t': Time, d, d': Data,ch:Channel |
 s.client.(s.sends) =d ->t->ch and  s.server.(s.receives) =d' ->t'->ch
 implies  t= t' )}
```

The second fact states the *truth* of the first fact to imply that data that has been sent *d* equals data that has been received *d'* and because the MitM does not exist, he has no ability to send or receive as a client (Line 2).



```
fact { all t,t':Time,d,d':Data,s:Communication_Status,ch:Channel |
(s.client.(s.sends) =d ->t->ch and   s.server.(s.receives) =d' ->t'->ch and t =t')
 implies ((d = d') and no s.mitmIntercepts.(s.sends) and
 no s.mitmIntercepts.(s.receives))}
```

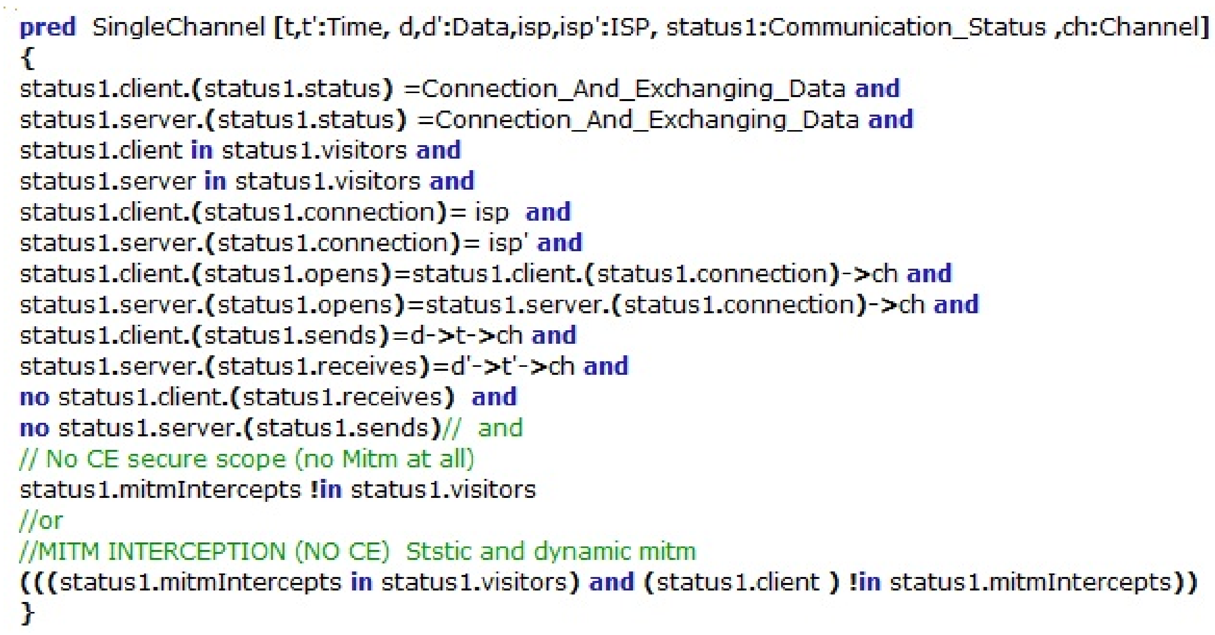
The third fact states that, for all s of Communication_Status, the *client*, and the *server* are not equal as they have different unique MAC address (Line 3). However, the *mitm* has not been constrained to not be equal to the *client*, because in case of an interception (second aspect), he may pretend to be a client to the server using the client's MAC address.



```
fact { all s: Communication_Status | s.client != s.server}
```

- Predicate

Our protocol model as seen in Appendix (C.4) has one predicate *SingleChannel* below which controls *Time t,t', Data, t,t', Internet server provider, isp, isp', Communication_Status status1,* and *channel ch1* (Line 1). The statuses are linked using "and ". The predicate also visualizes the operation of the protocol and is specified using *status1* of *Communication_Status.*



```
pred  SingleChannel [t,t':Time, d,d':Data,isp,isp':ISP, status1:Communication_Status ,ch:Channel]
{
status1.client.(status1.status) =Connection_And_Exchanging_Data and
status1.server.(status1.status) =Connection_And_Exchanging_Data and
status1.client in status1.visitors and
status1.server in status1.visitors and
status1.client.(status1.connection)= isp  and
status1.server.(status1.connection)= isp' and
status1.client.(status1.opens)=status1.client.(status1.connection)->ch and
status1.server.(status1.opens)=status1.server.(status1.connection)->ch and
status1.client.(status1.sends)=d->t->ch and
status1.server.(status1.receives)=d'->t'->ch and
no status1.client.(status1.receives)  and
no status1.server.(status1.sends)//  and
// No CE secure scope (no Mitm at all)
status1.mitmIntercepts !in status1.visitors
//or
//MITM INTERCEPTION (NO CE)  Ststic and dynamic mitm
(((status1.mitmIntercepts in status1.visitors) and (status1.client ) !in status1.mitmIntercepts))
}
```

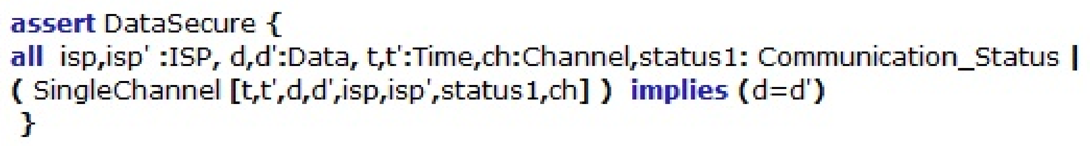
The *SingleChannel* predicate in (Lines 1-19) illustrates that it passes through one status. The *status* (*status1*) begins by determining the current status for the two hosts as they are both in the status of connecting and exchanging data (Lines 2,3) and these two hosts exists as visitors (Lines 4,5). In this status, the two parties are connecting to their selected ISP (isp,isp') (Lines 6,7), thus opening a single channel using their ISP to exchange data between them (Lines 8,9).

After that, the two hosts are now exchanging data at a time over the channel (Lines 10,11). The client never has the function of receiving (Line 12), while the server never has the function of sending (Line 13). For the first aspect, (Line 14) illustrate that there is no MitM as a visitor, while for the second aspect, (Line 15) illustrate that there is a MitM as a visitor but it has not intercepted the client as the client does not belong to the MitM.

When two parties are exchanging their data over the opened single channel, sending data over a channel takes place at sending time, and receiving data takes place over the same channel at receiving time. The assumption of this status is that, the data that has been received equals the data that has been sent if the sending time equals the receiving time and no MitM exist, so it is a secure scope and no MitM intercepts data that has been exchanged and data has neither been eavesdropped nor modified.

- Assertion

The assertion in Appendix (C.5), states that, if *SingleChannel* holds in *times* t,t', *Data* d,d',*ISP* isp,isp', and *Communication_Status* status1,status2, then the equalities of sending and receiving data hold as well.



```
assert DataSecure {
all  isp,isp' :ISP, d,d':Data, t,t':Time,ch:Channel,status1: Communication_Status |
( SingleChannel [t,t',d,d',isp,isp',status1,ch] )  implies (d=d')
 }
```

The user limits the scope of analysis to 3 atoms of MAC, 2 atoms of Time, 2 atoms of ISP, 2 atoms of Data, 1 atom of ConnectionStatus, 1 atom of Channel, and 1 atom of Communication_Status.

6.5 Results

In this section, we present the results we achieved from modelling, analysing, and checking the four properties of the first protocol for two aspects using the Alloy Analyser and bounded SAT solver. These two aspects are transmitting data securely, and transmitting data insecurely with the presence of a MitM.

For the first aspect: **the first property** is there is no MitM; **the second property** is opening one channel to exchange data between two hosts (client, server); **the third property** is the time for sending and receiving data over the opened channel has not been delayed which implies that the data received by the server equals the data sent by the client; **the fourth property** is MAC addresses is one and has to be unique. The second aspect has the same properties as the first aspect except **the first property** which requires that there is a MitM,

Before checking the satisfiability of the model, we need first to check its consistency. If the model is inconsistent, the analyser cannot work efficiently for detecting a counterexample.

The **run** command runs the SAT4J solver. The command asks the analyser to search for instances to visualize them. These instances assign sets and relations that their size is limited to be 3 atoms for *MAC*, 2 atoms for *Time*, 2 atoms for *ISP*, 2 atoms for *Data*, 1 atom for *ConnectionStatus*, 1 atom for *Channel*, 1 atom for *Communication_Status*. The visualised instances for the first protocol model are acceptable. They correspond to the declarations of the fields and signatures, and satisfy the predicate and the Alloy model together which means the model is consistent.

The **check** command searches for a counterexample showing an execution path that caused an error if one exist. The command looks for an instance that violates the assertion. This analysis is implemented with respect to the bounded scope of 3 atoms for *MAC*, 2 atoms for *Time*, 2 atoms for *ISP*, 2 atoms for *Data*, 1 atom for *ConnectionStatus*, 1 atom for *Channel*, 1 atom for *Communication_Statu*. Only a finite number of elements for each type is taken into account. Therefore, the absence of an instance does not include checking satisfiability.

The analyser spent 0.119s checking the four properties of the first protocol for the first aspect to generate counterexamples with respect to the finite scope.

The automatic check detects that, for the first aspect, when the predicate shows that there is no MitM as a visitor there is no counterexample to the assertion within the limited scope. The results is acceptable because the analyser found an instance as seen in Figures D.1 that satisfied the specification but did not violate the *DataSecure assertion*. The model allows us to visualise the model states where the status is shown in Figure 6.1 indicated by $. We conclude that the property that has been analysed holds in the model within the provided scopes.

Figure 6.1, visualises the first aspect of transmitting data securely over a single channel. As we see, transmitting data over a single channel requires one status. Both the client *mac1* and the server *mac0* have the same status which is *Connec-*

tion_And_Exchanging_Data. Both the client and the server appeared as visitors and there is no MitM to intercept the client. Each connects to its Internet Service Provider as the client connects to *isp1*, while the server connects to *isp0*. In this connection status we notice that a channel *ch* is opened between two hosts to exchange data between them. As we see, the client sends *data1* at *time1*, and the sever receives the same data *data1* at the same sending time *time1*. The non existence of the interception of the MitM led to the receiving data being equal to the sending data with the assumption that the time for sending and receiving data are equal. So, we notice that sending and receiving time still remained the same which corresponds with our assumption that they should be equals.

So, from the figure, we achieved all the properties that we wanted to assert for the first aspect. The result we achieved made the system properties satisfy the system requirements. Thus, the submitted message has not been modified or read.

For the second aspect (insecure protocol) this analysis is implemented with respect to a bounded scope of 3 atoms for *MAC*, 2 atoms for *Time*, 2 atoms for *ISP*, 2 atoms for *Data*, 1 atom for *ConnectionStatus*, 1 atom for *Channel*, 1 atom for *Communication_Statu* in which only a finite number of elements for each type is taken into account; and therefore absence of an instance does not include checking satisfiability.

The Alloy Analyser spent 0.211s checking the four properties of the first protocol for the second aspect. **The first property** is that there is a MitM; **the second property** is opening one channel to exchange data between two hosts (client, server); **the third property** is the time for sending and receiving data over the opened channel has not been delayed which implies to received data by the server equals to sent data by the client; **the fourth property** is MAC addresses are one and unique.

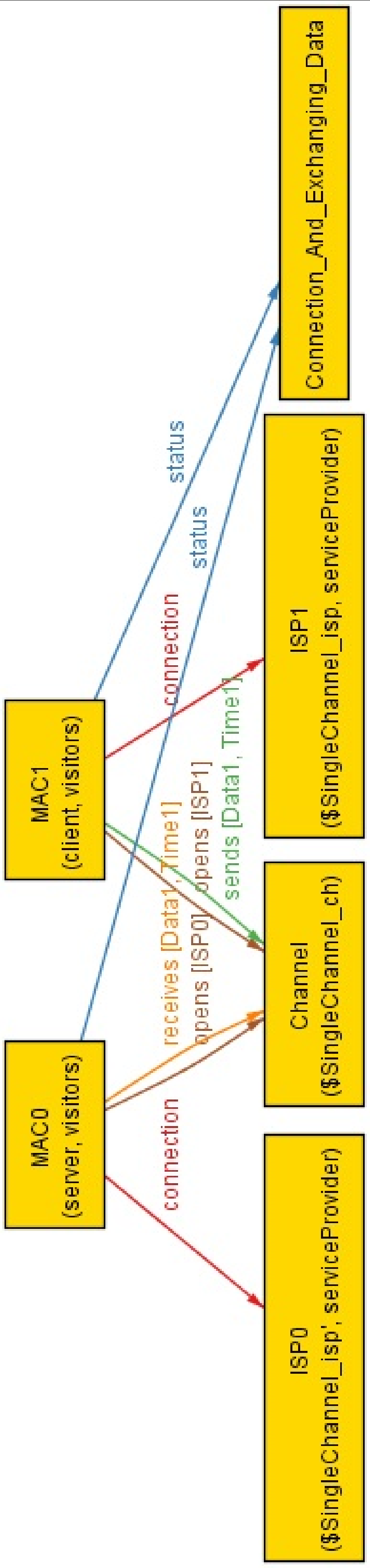


Figure 6.1: An Instance of The First Aspect (Secure Protocol)

The automatic check detects that there is a MitM as visitor which intercepts the client's MAC. There are two kinds of counterexample to the assertion within the limited scope checked. The results is acceptable because the analyser found an instance as seen in Figures 6.2 for dynamic MitM, and (6.3) for static MitM that violated the *DataSecure assertion*. The model allows us to visualise the model states where the status is shown in Figures 6.2, 6.3 indicated by \$. It concludes that the property that has been analysed does not hold in the model within the provided scopes.

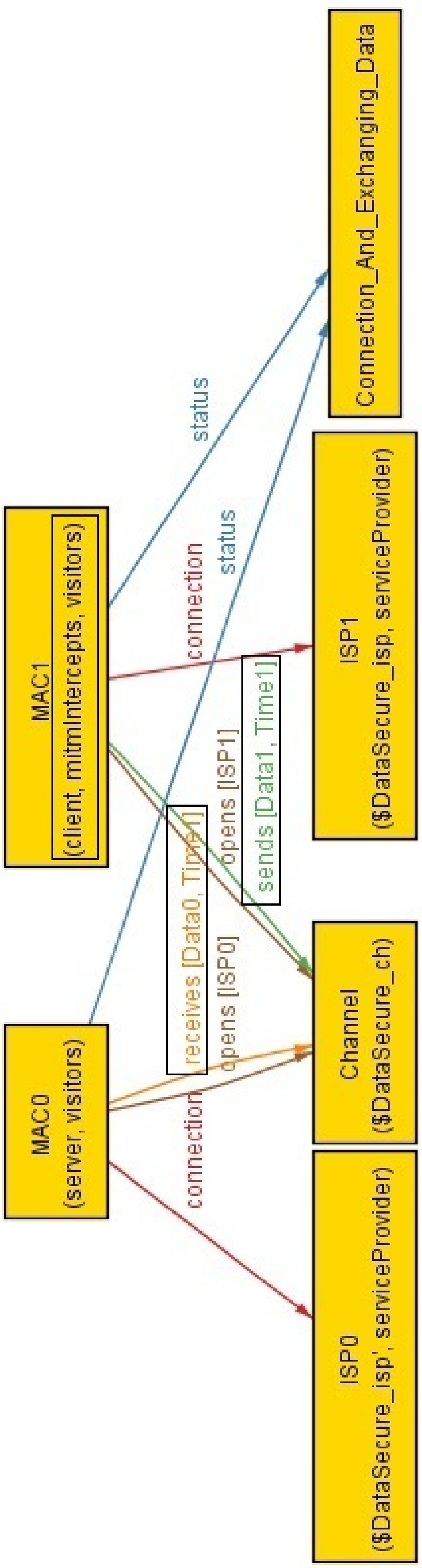


Figure 6.2: Generating A Counterexample for The Second Aspect (Insecure Protocol) Dynamic MitM

Figure 6.2 visualises the second aspect of transmitting data over a single channel which requires one status. Both the client *mac1* and the server *mac0* have the same status which is *Connection_And_Exchanging_Data*. Both the client and the server appeared as visitors and there is a *dynamic* MitM which intercepts the client. The client and MitM connect to *isp1*, while the server connects to *isp0*. In this connection status we notice that a channel *ch* is opened between two hosts to exchange data between them. As we see, the client sends *data1* at *time1*, and the server receives different data *data0* that has been sent at the same sending time *time1*. The interception of the MitM led to receiving data that is not equal to the sending data assuming that the time for sending and receiving data is not equal because of the interception and modifying data. However, we notice that sending and receiving time still remained the same which conflicts with our assumption that they should be different, and here is the counterexample.

So, from the figure we achieved all the properties that we wanted to assert for the second aspect except the third property which is that the time for sending and receiving data over the opened channel has not been delayed. The result we achieved shows that system properties do not satisfy system requirements. Thus, the submitted message has been modified.

Figure 6.3, visualises the second aspect of transmitting data over a single channel which requires one status. Both the client *mac1* and the server *mac0* have the same status which is *Connection_And_Exchanging_Data*. Both the client and the server appeared as visitors and there is a *static* MitM which intercepts the client. The client and MitM connect to *isp1*, while the server connects to *isp0*. In this connection status we notice that a channel *ch* is opened between two hosts to exchange data between them. As we see, the client sends *data1* at *time1*, and the sever receives the same data *data1* at the same sending time *time1*. The interception of the MitM led to receiving data that is equal to the sending data, assuming that the time for sending and receiving data is not equal because of the interception and reading data. However, we notice that time still remained the same which conflicts with our assumption that they should be different, and here is the counterexample.

So, from the figure we achieved all the properties that we wanted to assert for the second aspect except the third property which is that the time for sending and receiving data over the opened channel has not been delayed. These requirements are that the equality of sending and receiving time leads to the data that has been received being equal to the data that has been sent. Thus, the submitted message has been eavesdropped.

By examining the trace of the instance that the SAT solver found with counterexamples as shown in Figures 6.2, 6.3, we can see that, even with the assumption that there is a MitM as a visitor and he intercepts the client, data has not been exchanged securely, as data that has been received does or does not equal data that has been sent. Also there is no time delay in the two counterexamples which conflicts with our assumption that when exchanged data is insecure then time should not be equal which shows that interception happened.

To achieve the the third property, we needed to add constraints as facts. After adding the three facts as seen in Appendix (C.3), the Alloy Analyser checking found that there is no counterexample in 0.409s even with the large scope in checking the *DataSecure* assertion.

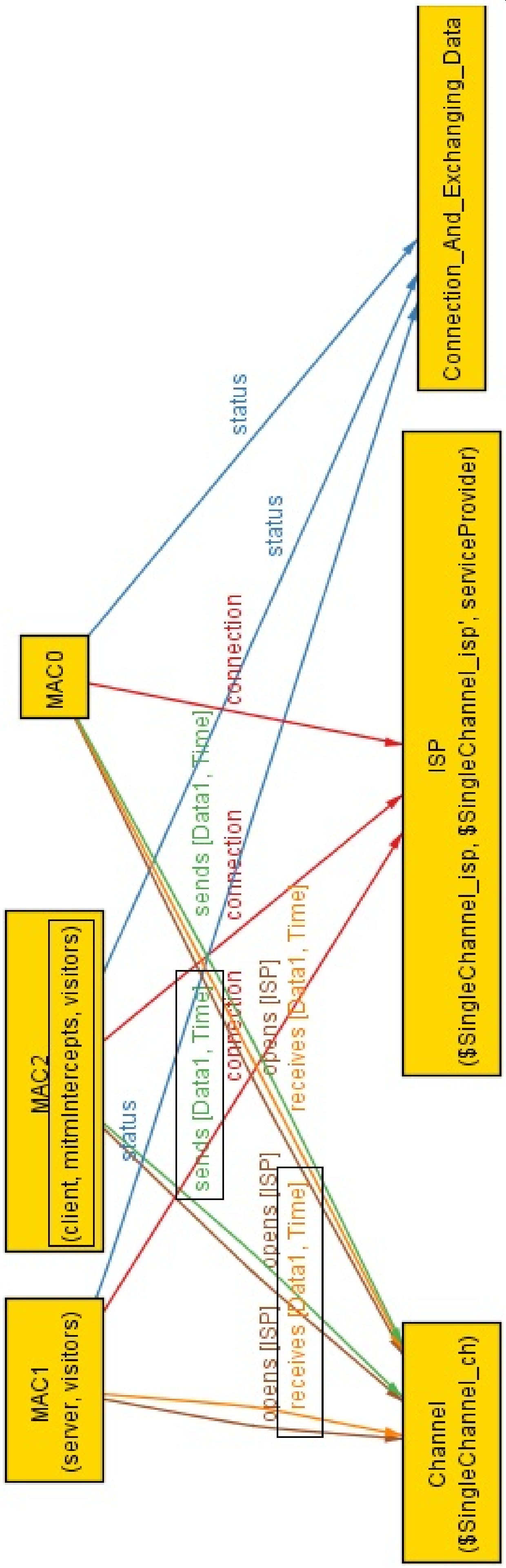


Figure 6.3: Generating A Counterexample for The Second Aspect (Insecure Protocol) Static MitM

We now explore the model further to identify the problem that data that has been received is equal to data that has been sent because of the receiving time being equal to the sending time; MitM did not intercept any MAC address because we restricted that he is not in the visitors. Thus there is no counterexample as seen in Figure 6.4 because the analyser did not find an instance that satisfied the specification, with each constraint formula in *facts* but violating the *DataSecure assertion*. We conclude that the property that has been analysed holds in the model only within the provided scopes and a MitM in a single channel may cause readable or modified data without showing any delay time unless the MitM exists as a visitor without interception as seen in Figure 6.4.

Figure 6.4, visualises the second aspect of transmitting data over a single channel with no MitM interception. As we see, transmitting data over a single channel requires one status. Both the client *mac2* and the server *mac0* have the same status which is *Connection_And_Exchanging_Data*. The client, the server, and a MitM *mac1* appeared as visitors and the MitM did not intercept the client. The client connect to *isp1*, while the server connects to *isp0*. In this status we notice that a channel *ch* is opened between the two hosts to exchange data. As we see, the client sends *data1* at *time1*, and the sever receives the same data *data1* at the same sending time *time1*. The existence of the MitM as a visitor with no interception led to received data being equal to the sent data assuming that the time for sending and receiving data is equal. So, we notice that time still remained the same which corresponds with our assumption that they should be equal.

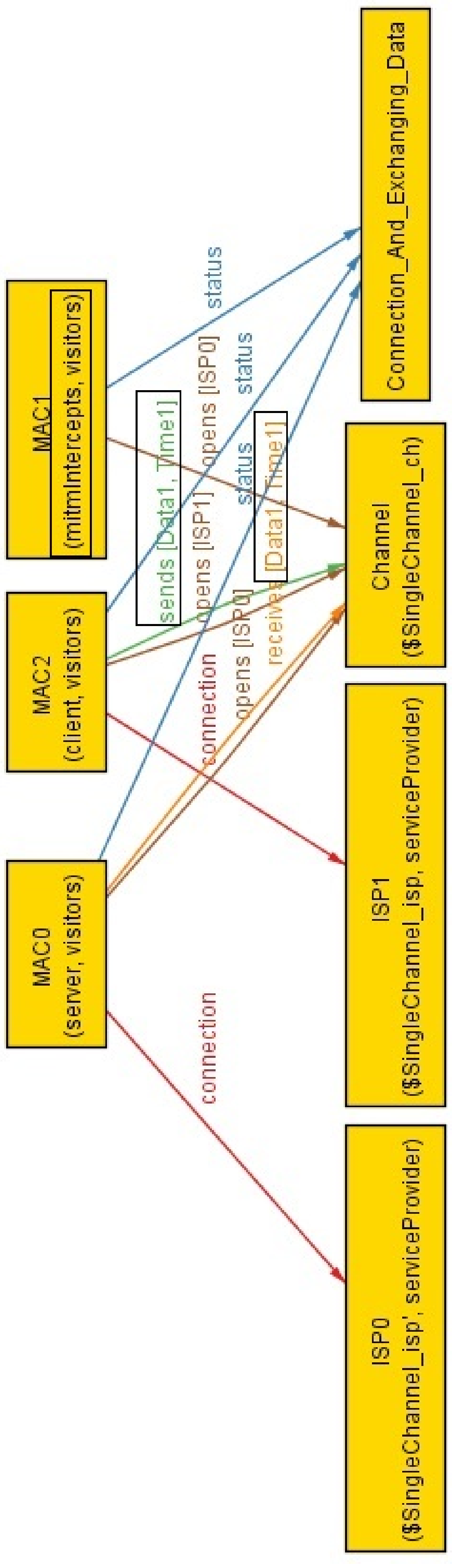


Figure 6.4: AA, No Counterexample, A valid Instance

So, from the figure we achieved all the properties that we wanted to assert for the second aspect. These properties are **the first property** is that there is no MitM, **the second property** is opening one channel to exchange data between two hosts (client, server), **the third property** is the time for sending and receiving data over the opened channel has not been delayed which implies that the received data equals the sent data, **the fourth property** is MAC addresses is one and unique. Thus, the submitted message has not been modified or read.

Next, we will see how the first protocol is developed when the scopes of *Communication_Status, channels, ISP, MACs, data*, and *time* are increased.

6.6 Protocol of Transmitting Data Over Multichannel Case Study

6.6.1 Second Protocol: Describing Data Transmitting Securely Over Multichannel WLANs in The presence of MitM

Our second protocol cares about protecting data to make it unreadable based on submitting it over a multichannel instead of one channel. The protocol is developed from the first protocol by separating transmitted data into letters and its matching indices, and transmitting each over different channel connected with different MAC addresses with different ISPs. Therefore, the properties and requirements are also developed. The protocol considers there being a MitM to intercept data transmitting over multichannel between the two parties communication. As a result, data that has been received may have not been read when data over a channel is intercepted because it includes either letters or indices, the consideration of the different MAC addresses, and connecting to different ISPs. Intercepting one of two channels may cause a delay time for the intercepted one. Thus, if no modification occurs for the data that is transmitted over an intercepted channel, data that has been received equals the data that has been sent which leads to data not being read.

6.6.1.1 Protocol Model Properties and Requirements

The model of the protocol can be enhanced by adding six properties. **The first property** is that data (indices) and (letters) are transmitted over two different channels. **The second property** is that only one channel is able to be intercepted during the two communication statuses, i.e no two channels have been intercepted. **The third property** is that the MAC address is changeable and no MAC addresses are equal. **The fourth property** is that there is one MitM. **The fifth property** is that two channels are opened to exchange data between two hosts (client, server), one channel for exchanging letters and the other for exchanging indices. **The sixth property** is that the time for sending and receiving data over the opened channel has not been delayed which implies that data received by the server equals to data sent by the client.

The protocol requirements are that if one of two channels has been intercepted, data that has been received equals data that has been sent over this intercepted channel although the time for sending and receiving data should not be equal which refers to time delays which are caused by the interception.

6.6.1.2 Model Structure Description

As seen in Appendix (E), modelling this protocol passes through two statuses. The first status is when two parties connect to their first ISP using the original MAC address, thus opening a single channel between them to exchange indices of data. The second status is when two parties connect to their different second ISP, thus opening a different single channel between them to exchange letters with the original MAC address changing to the fake one.

The assumption of these statuses is that data that has been received equals data that has been sent if the sending time equals the receiving time. Furthermore no MitM exists to act as a client or a server using their MAC addresses to pretend to be a client to the server and a server to the client. So this is a secure scope, no MitM intercepts data that has been exchanged and data has neither been eavesdropped nor modified.

Another assumption is that data that has been received equals data that has been sent if one sending time equals to one of receiving time and there is a MitM. This shows that this is still a secure scope even if the MitM intercepts one of the data (indices or letters) over one of the multichannels because the whole data is kept unreadable.

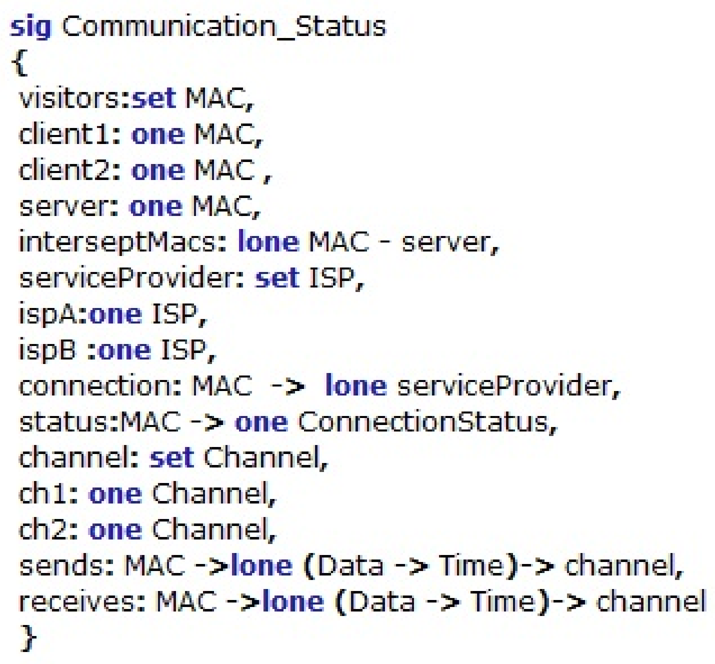
6.6.1.3 Modelling and Checking The Protocol Using Alloy

Modelling the protocol is similar to the previous protocol in terms of the interacting signatures. However we add some relations and facts to correspond with the new properties. We also add new predicates and the assertion that we want to check. The predicate is developed to include the behaviour of two statuses. The assertion is developed to check that sending and receiving data are equal.

- Relation Declarations, and Multiplicities

  The protocol as seen in Appendix (E.1) has some relations that are common with the first protocol. Here, we are going to express the developed relations in the second protocol.

  - The field *client1* represents the first (the original) *MAC* that exists as client in the *Communication_Status*, and declares a binary relation of type *Communication_Status ->MAC* which maps each element of *Communication_Status* to *one* element of *MAC*.



```
sig Communication_Status
{
 visitors:set MAC,
 client1: one MAC,
 client2: one MAC ,
 server: one MAC,
 interseptMacs: lone MAC - server,
 serviceProvider: set ISP,
 ispA:one ISP,
 ispB :one ISP,
 connection: MAC  ->  lone serviceProvider,
 status:MAC -> one ConnectionStatus,
 channel: set Channel,
 ch1: one Channel,
 ch2: one Channel,
 sends: MAC ->lone (Data -> Time)-> channel,
 receives: MAC ->lone (Data -> Time)-> channel
 }
```

The binary relation $client1 \subseteq Communication\_Status \times MAC$. (Lines 9 Appendix E.1, Line 2 Appendix E.2) declares a relation *client1* with domain *Communication_Status* and range *MAC*. The declaration of *client1* contains the multiplicity annotation *one* which makes *client1* a total function: a binary relation that associates every *Communication_Status* with *one* of *MAC*. For every element *cs* in *Communication_Status*, the keyword *one* before *MAC* constrains the term *cs.client1* to be *one*. The declaration of

$$client1\ in\ Communication\_Status -> MAC$$

can be expressed as:

$$\forall\ cs : Communication\_Status |\ cs.client1\ \subseteq\ (\textbf{one}\ MAC)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status |\ \textbf{one}\ cs.client1$$

The multiplicity keyword *one* makes the declaration hold when the relation *client1* is a total function from *Communication_Status* to *MAC*.

- The field *client2* represents the second (the fake) *MAC* that exists as a client in the *Communication_Status*, and declares a binary relation of type *Communication_Status ->MAC* which maps each element of *Communication_Status* to *one* element of *MAC*.

  The binary relation *client2* ⊆ *Communication_Status* × *MAC*. (Lines 9 Appendix E.1, Line 3 Appendix E.2) declares a relation *client2* with domain *Communication_Status* and range *MAC*. The declaration of *client2* contains the multiplicity annotation *one* which makes *client2* a total function: a binary relation that associates every *Communication_Status* with (*one*) exactly one of *MAC*. For every element *cs* in *Communication_Status*, the keyword *one* before *MAC* constraint]s the term *cs.client2* to be *one*. The declaration of

  $$client2\ in\ Communication\_Status -> MAC$$

  can be expressed as:

  $$\forall\ cs : Communication\_Status|\ cs.client2\ \subseteq\ (\textbf{one}\ MAC)$$

  We can express the multiplicity constraints as:

  $$all\ cs : Communication\_Status|\ \textbf{one}\ cs.client2$$

  The multiplicity keyword *one* makes the declaration hold when the relation *client2* is a total function from *Communication_Status* to *MAC*.

- The field *ispA* represents the first *ISP* that has been connected to, and declares a binary relation of type *Communication_Status ->ISP* which maps each element of *Communication_Status* to *one* element of *ISP*.

  The binary relation *ispA* ⊆ *Communication_Status* × *ISP*. (Lines 9 Appendix E.1, Line 7 Appendix E.2) declares a relation *ispA* with domain *Communication_Status* and range *ISP*. The declaration of *ispA* contains the multiplicity annotation *one* which makes *ispA* a total function: a binary relation that associates every *Communication_Status* with *one* of *ISP*. For every element *cs* in *Communication_Status*, the keyword *one* before *ISP* constrains the term *cs.ispA* to be *one*. The declaration of

$$ispA\ in\ Communication\_Status -> ISP$$

can be expressed as:

$$\forall\ cs : Communication\_Status|\ cs.ispA\ \subseteq\ (\mathbf{one}\ ISP)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|\ \mathbf{one}\ cs.ispA$$

The multiplicity keyword *one* makes the declaration hold when the relation *ispA* is a total function from *Communication_Status* to *ISP*.

– The field *ispB* represents the second *ISP* that has been connected to, and declares a binary relation of type *Communication_Status ->ISP* which maps each element of *Communication_Status* to *one* element of *ISP*.

  The binary relation *ispB* $\subseteq$ *Communication_Status* $\times$ *ISP*. (Lines 9 Appendix E.1, Line 8 Appendix E.2) declares a relation *ispB* with domain *Communication_Status* and range *ISP*. The declaration of *ispB* contains the multiplicity annotation *one* which makes *ispB* a total function: a binary relation that associates every *Communication_Status* with *one* of *ISP*. For every element *cs* in *Communication_Status*, the keyword *one* before *ISP* constrains the term *cs.ispB* to be *one*. The declaration of

$$ispB\ in\ Communication\_Status -> ISP$$

  can be expressed as:

$$\forall\ cs : Communication\_Status|\ cs.ispB\ \subseteq\ (\mathbf{one}\ ISP)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status |\ \textbf{one}\ cs.ispB$$

The multiplicity keyword *one* makes the declaration hold when the relation *ispB* is a total function from *Communication_Status* to *ISP*.

- The field *channel* represents the available *Channel* in the *Communication_Status*, and declares a binary relation of type *Communication_Status ->Channel* which maps each element of *Communication_Status* to *set* of elements of *Channel*.

  The binary relation *channel* ⊆ *Communication_Status* × *Channel*. (Lines 9 Appendix E.1, Line 11 Appendix E.2) declares a relation *channel* with domain *Communication_Status* and range *Channel*. The declaration of *channel* contains the multiplicity annotation *set* which makes *channel* a function: a binary relation that associates every *Communication_Status* with set of *Channel*. For every element *cs* in *Communication_Status*, the keyword *set* before *Channel* constrains the term *cs.channel* to be *set*. The declaration of

  $$channel\ in\ Communication\_Status - > Channel$$

  can be expressed as:

  $$\forall\ cs : Communication\_Status|\ cs.channel\ \subseteq\ (\textbf{set}\ Channel)$$

  We can express the multiplicity constraints as:

  $$all\ cs : Communication\_Status|\ \textbf{set}\ cs.channel$$

  The multiplicity keyword *set* makes the declaration hold when the relation *channel* is a function from *Communication_Status* to *Channel*.

- The field *ch1* represents the first *Channel* that transmits the first data (indices) over, and declares a binary relation of type *Commu-*

nication_Status ->*Channel* which maps each element of *Communication_Status* to *one* element of *Channel*.

The binary relation *ch1* $\subseteq$ *Communication_Status* $\times$ *Channel*. (Lines 9 Appendix E.1, Line 12 Appendix E.2) declares a relation *ch1* with domain *Communication_Status* and range *Channel*. The declaration of *ch1* contains the multiplicity annotation *one* which makes *ch1* a total function: a binary relation that associates every *Communication_Status* with *one* of *Channel*. For every element *cs* in *Communication_Status*, the keyword *one* before *Channel* constrains the term *cs.ch1* to be *one*. The declaration of

$$ch1\ in\ Communication\_Status -> Channel$$

can be expressed as:

$$\forall\ cs : Communication\_Status|\ cs.ch1\ \subseteq\ (\mathbf{one}\ Channel)$$

We can express the multiplicity constraints as:

$$all\ cs : Communication\_Status|\ \mathbf{one}\ cs.ch1$$

The multiplicity keyword *one* makes the declaration hold when the relation *ch1* is a total function from *Communication_Status* to *Channel*.

– The field *ch2* represents the first *Channel* that transmits the second data (letters) over, and declares a binary relation of type *Communication_Status* ->*Channel* which maps each element of *Communication_Status* to *one* element of *Channel*.

  The binary relation *ch2* $\subseteq$ *Communication_Status* $\times$ *Channel*. (Lines 9 Appendix E.1, Line 13 Appendix E.2) declares a relation *ch2* with domain *Communication_Status* and range *Channel*. The declaration of *ch2* contains the multiplicity annotation *one* which makes *ch2* a total function: a binary relation that associates every *Communication_Status* with *one* of *Channel*. For every element *cs* in *Communication_Status*, the keyword *one* before *Channel* constrains the term *cs.ch2* to be *one*. The declaration of

$$ch2\ in\ Communication\_Status -> Channel$$

can be expressed as:

$$\forall\ cs : Communication\_Status|\ cs.ch2\ \subseteq\ (\textbf{one}\ Channel)$$

We can express the multiplicity constraints as:

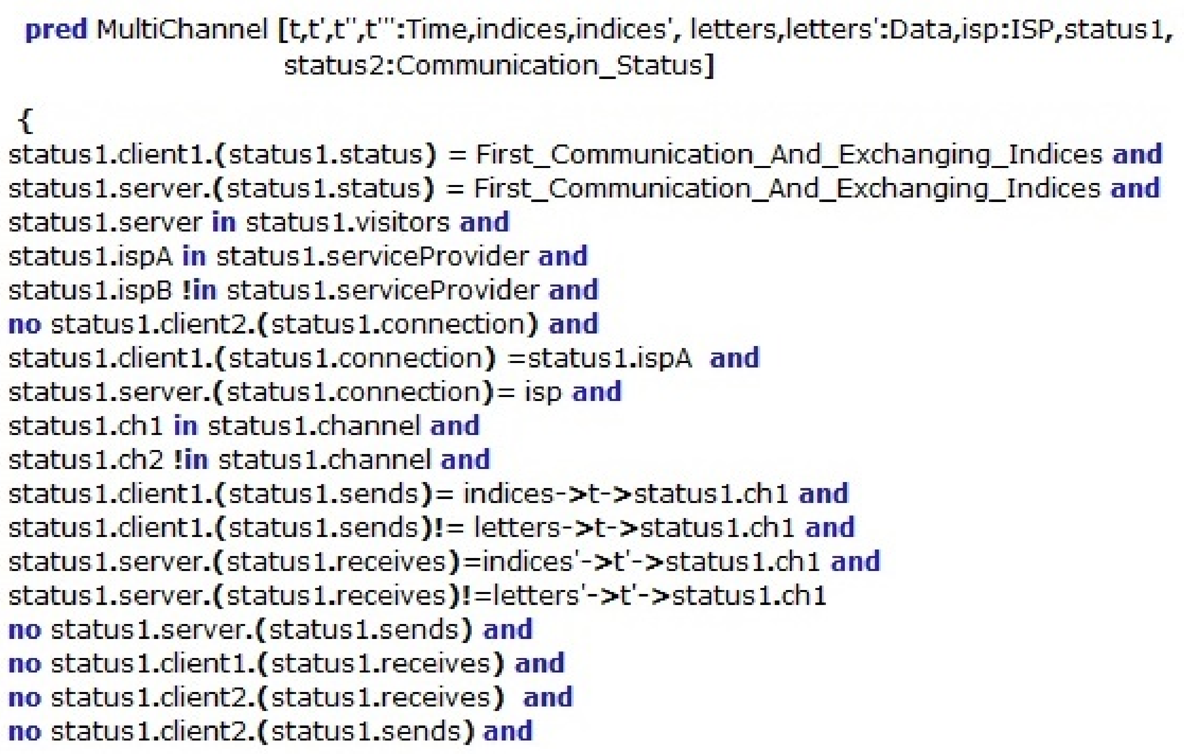
$$all\ cs : Communication\_Status|\ \textbf{one}\ cs.ch2$$

The multiplicity keyword *one* makes the declaration hold when the relation *ch2* is a total function from *Communication_Status* to *Channel.*

- Predicate

Our protocol as seen in Appendix (E.4) has one predicate *MultiChannel* which controls *Time t,t',t",t"', Data, indices,indices', letters,letters', Internet server provider, isp, Communication_Status status1, status2* (Line 1). The *MultiChannel* predicate is the operation of the protocol and is specified using *status1*, and *status2* of *Communication_Status. status1*, and *status2* are instances of *Communication_Status* showing the statuses of *Communication_Status*, first and second operations respectively.

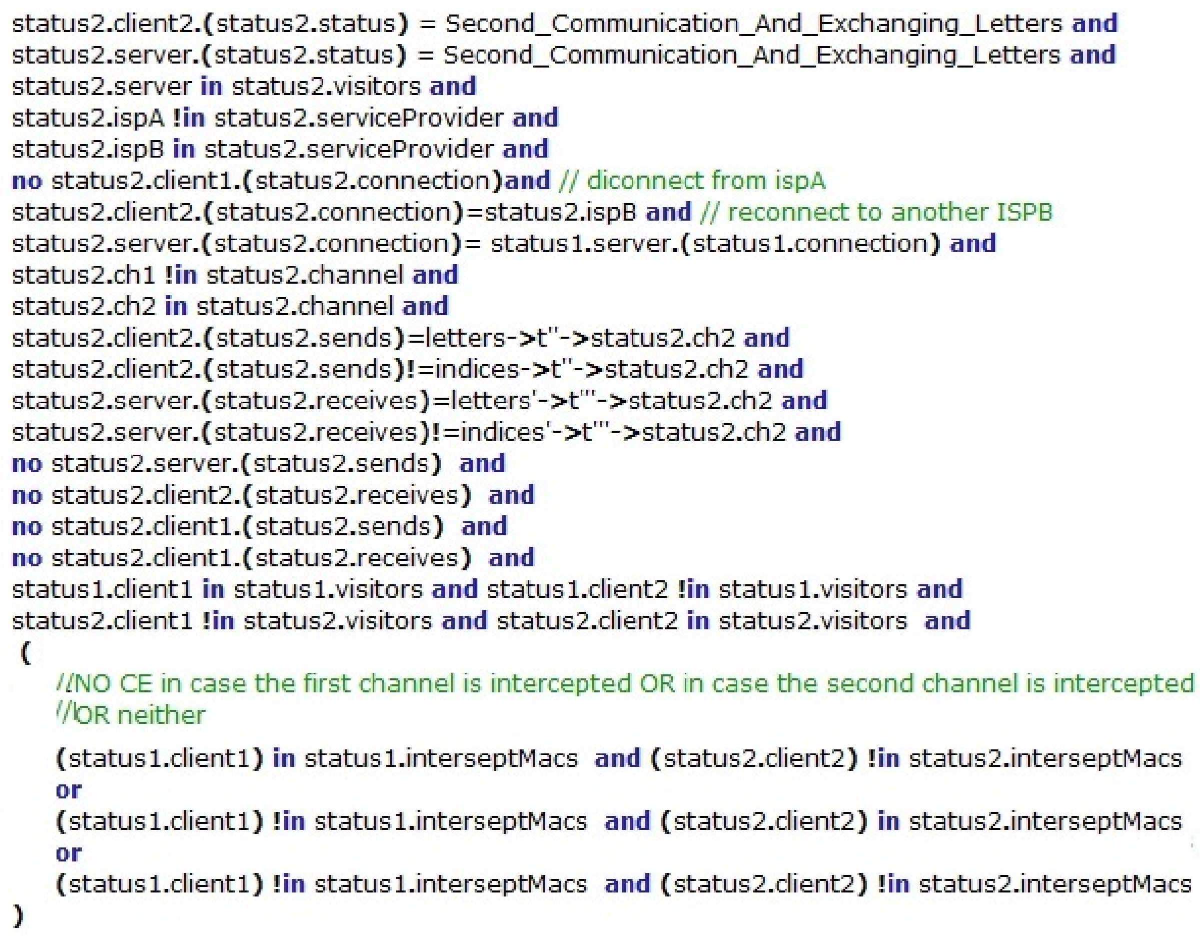
The MultiChannel predicate illustrates that it passes through two statuses. The first status occurs from (Line 2-19). The status begins by showing that two parties *client1* and *server* are starting the communication in terms of communication and exchanging indices (Line 2,3). The server belongs to the visitors (Line 4). The first *ISPA* belongs to the service provider making *client1* connects to (Lines 5,8) while the second *ISPB* does not belong to service provider (Line 6). The second Client *Client2* is not connected (Line 7). The first channel belongs to the channels to exchange indices between *client1* and the server over this first channel at a time (Lines 10,12, and 14), while the second channel does not belong to the channels (Line 11). Also, in this status no exchange of letters (sending or receiving) has occurred (Line 13,15). Moreover, neither *client1* has a receiving function nor server has a sending function (Line 16,17). Also, *client2* has no ability to send or receive (Line 18,19). In *status1*, *client1* belongs to the visitors (Lines 38) while *client2* does not belong to the visitors (Lines 39).



```
pred MultiChannel [t,t',t'',t''':Time,indices,indices', letters,letters':Data,isp:ISP,status1,
                    status2:Communication_Status]
 {
status1.client1.(status1.status) = First_Communication_And_Exchanging_Indices and
status1.server.(status1.status) = First_Communication_And_Exchanging_Indices and
status1.server in status1.visitors and
status1.ispA in status1.serviceProvider and
status1.ispB !in status1.serviceProvider and
no status1.client2.(status1.connection) and
status1.client1.(status1.connection) =status1.ispA  and
status1.server.(status1.connection)= isp and
status1.ch1 in status1.channel and
status1.ch2 !in status1.channel and
status1.client1.(status1.sends)= indices->t->status1.ch1 and
status1.client1.(status1.sends)!= letters->t->status1.ch1 and
status1.server.(status1.receives)=indices'->t'->status1.ch1 and
status1.server.(status1.receives)!=letters'->t'->status1.ch1
no status1.server.(status1.sends) and
no status1.client1.(status1.receives) and
no status1.client2.(status1.receives)  and
no status1.client2.(status1.sends) and
```

The second status occurs from (Line 20-45). The status begins by showing that two parties *client2* and *server* are starting the communication in terms of communication and exchanging letters (Line 20,21). The server belongs to the visitors (Line 221). The second *ISPB* belongs to the service provider making *client2* connect to (Line 24,26) while the first *ISPA* does not belong to the service provider (Line 23). The server still connects to its (isp) that it is already connected to in the first status (Line 27). The second channel belongs to the channels making *client2* and the server exchange letters over this second channel at the time (Line 29,30, and 32), while the first channel does not belong to the channels (Line 28). Also, in this status no exchange of indices (sending or receiving) has occurred

(Line 31,33). Moreover, neither *client2* has a receiving function nor server has a sending function (Line 34,35). Also, *client1* has no ability to send or receive in the second status (Line 36,37). In *status2*, *client2* belongs to the visitors (Lines 41) while *client1* does not belong to the the visitors (Lines 40).

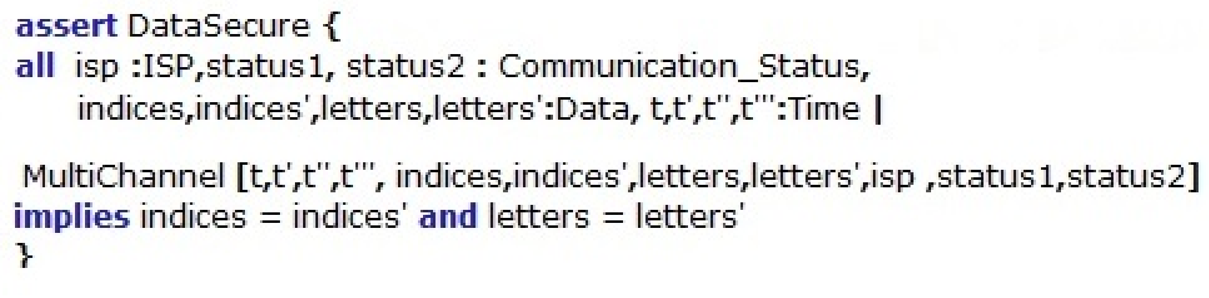


```
status2.client2.(status2.status) = Second_Communication_And_Exchanging_Letters and
status2.server.(status2.status) = Second_Communication_And_Exchanging_Letters and
status2.server in status2.visitors and
status2.ispA !in status2.serviceProvider and
status2.ispB in status2.serviceProvider and
no status2.client1.(status2.connection)and // diconnect from ispA
status2.client2.(status2.connection)=status2.ispB and // reconnect to another ISPB
status2.server.(status2.connection)= status1.server.(status1.connection) and
status2.ch1 !in status2.channel and
status2.ch2 in status2.channel and
status2.client2.(status2.sends)=letters->t''->status2.ch2 and
status2.client2.(status2.sends)!=indices->t''->status2.ch2 and
status2.server.(status2.receives)=letters'->t'''->status2.ch2 and
status2.server.(status2.receives)!=indices'->t'''->status2.ch2 and
no status2.server.(status2.sends)  and
no status2.client2.(status2.receives)  and
no status2.client1.(status2.sends)  and
no status2.client1.(status2.receives)  and
status1.client1 in status1.visitors and status1.client2 !in status1.visitors and
status2.client1 !in status2.visitors and status2.client2 in status2.visitors  and
 (
    //NO CE in case the first channel is intercepted OR in case the second channel is intercepted
    //OR neither
    (status1.client1) in status1.interseptMacs  and (status2.client2) !in status2.interseptMacs
    or
    (status1.client1) !in status1.interseptMacs  and (status2.client2) in status2.interseptMacs
    or
    (status1.client1) !in status1.interseptMacs  and (status2.client2) !in status2.interseptMacs
 )
```

In this protocol, transmitting data may continue through one of three operations. Either there is a *MitM* and *client1* in *status1* belongs to the intercepts and *client2* in *status2* does not belong to the intercepts (Line 43) which means the interception takes place in the first status for the first client. Or,there is a *MitM* and *client1* in *status1* which does not belong to the intercepts and *client2* in *status2* belongs to the intercepts (Line 44) which means the interception takes place in the second status for the second client. Or there is a *MitM* and *client1* in *status1* which does not belong to the intercepts and *client2* in *status2* does not belong to the intercepts (Line 45) which means that both channels have not been intercepted.

- Assertion

The protocol as seen in Appendix (E.5) has one assertion *DataSecure*. As seen in the assertion (Line 1), it states that, if *MultiChannel* meets the facts at *times* t,t’,t”,t”’, *Data* indices,indices’, letters,letters’,*ISP* isp, and *Communication_Status* (status1), (status2). This implies that the equalities of sending and receiving indices and letters hold as well.



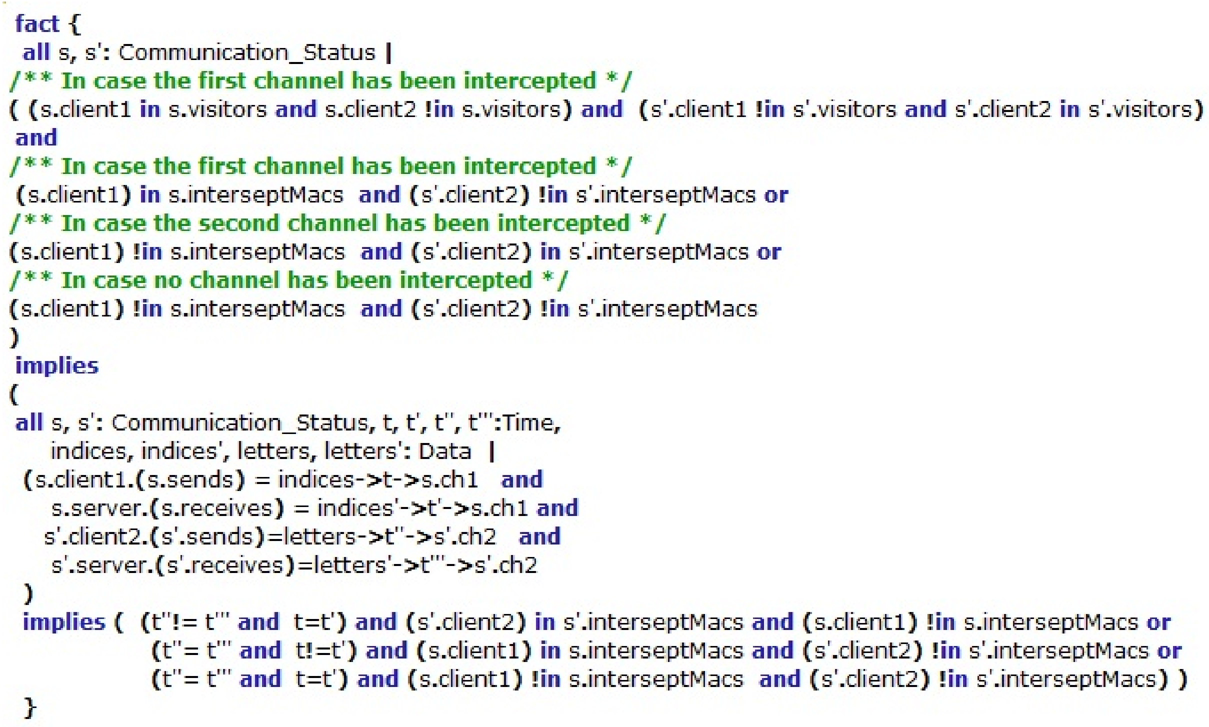
```
assert DataSecure {
all  isp :ISP,status1, status2 : Communication_Status,
     indices,indices',letters,letters':Data, t,t',t'',t''':Time |

 MultiChannel [t,t',t'',t''', indices,indices',letters,letters',isp ,status1,status2]
implies indices = indices' and letters = letters'
}
```

- Fact

The protocol as seen in Appendix (F.2) has three facts (Line 1-10). In the first fact (Lines 1-5) we need to constrain that, for all *s,s’*, which represent the first and the second communication in *Communication_Status* respectively, as follows. The first status *s* the first client *client1* belongs to the visitors, while the second client *client2* does not belong to the visitors, and in the second status *s’* the first client (client1) does not belong to the visitors, while the second client (client2) belongs to the visitors.

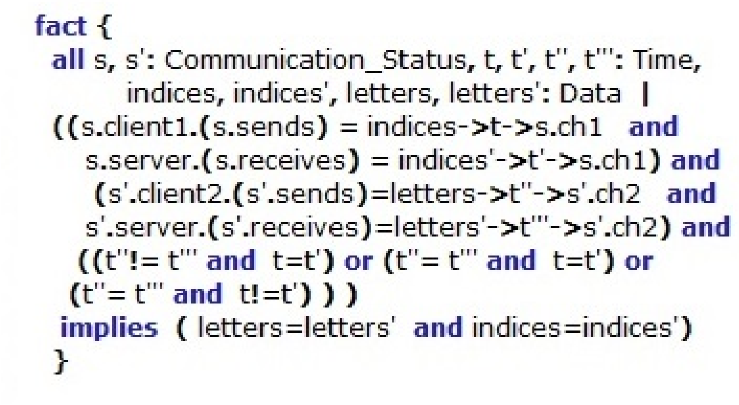
We have three cases, either: 1) the first channel has been intercepted and the second channel has not been intercepted when *client1* in *status1* has been intercepted, i.e it belongs to the *MitM*, while *client2* in *status2* has not been intercepted, i.e it does not belong to the *MitM*; or 2) the first channel has not been intercepted and the second channel has been intercepted when *client1* in *status1* has not been intercepted, i.e it does not belong to the *MitM*, while *client2* in *status2* has been intercepted, i.e it belongs to the *MitM*; or 3) the first and the second channel have not been intercepted when *client1* in *status1* and *client2* in *status2* have not been intercepted, i.e both of them do not belong to the *MitM*.

These three cases above imply that, for all *s,s'* which represent the first and the second communication in *Communication_Status* respectively, and all time *t,t',t",t"'* which represent sending and receiving indices,letters time respectively, and for all data i*ndices, indices',letters,letters'* which represent sending and receiving data indices and letters respectively as follows. The data *indices'* the server receives at time *t'* over the first channel *ch1* equals the data *indices* the *client1* sends at time *t* over the first channel *ch1* and the data *letters'* the server receives at time *t"'* over the second channel *ch2* equals the data *letters* the *client2* sends at time *t"* over the second channel *ch2* implies that the receiving time *t'* is not equal to the sending time *t* and the receiving time *t"'* is equal to the sending time *t"* when there is an interception in the first channel, or the receiving time *t'* is equal to the sending time *t* and the receiving time *t"'* is not equal to the sending time *t"* when there is an interception in the second channel, or the receiving time *t'* is equal to the sending time *t* and the receiving time *t"'* is equal to the sending time *t"* when there is no interception in both channels.

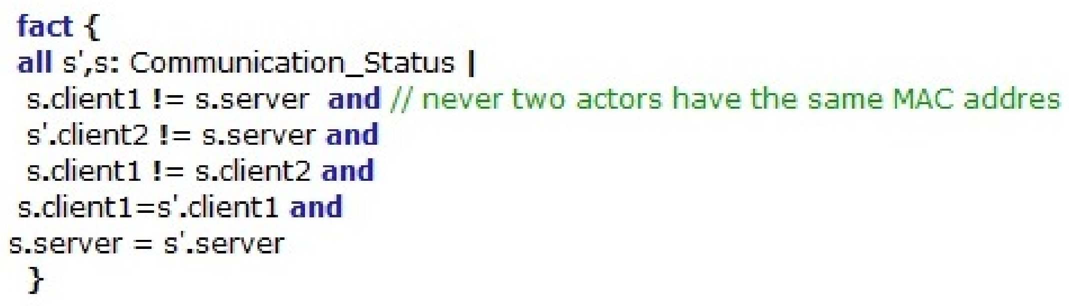


```
 fact {
  all s, s': Communication_Status |
/** In case the first channel has been intercepted */
( (s.client1 in s.visitors and s.client2 !in s.visitors) and  (s'.client1 !in s'.visitors and s'.client2 in s'.visitors)
 and
/** In case the first channel has been intercepted */
 (s.client1) in s.interseptMacs  and (s'.client2) !in s'.interseptMacs or
/** In case the second channel has been intercepted */
(s.client1) !in s.interseptMacs  and (s'.client2) in s'.interseptMacs or
/** In case no channel has been intercepted */
(s.client1) !in s.interseptMacs  and (s'.client2) !in s'.interseptMacs
)
 implies
(
 all s, s': Communication_Status, t, t', t'', t''':Time,
     indices, indices', letters, letters': Data  |
  (s.client1.(s.sends) = indices->t->s.ch1   and
     s.server.(s.receives) = indices'->t'->s.ch1 and
    s'.client2.(s'.sends)=letters->t''->s'.ch2   and
     s'.server.(s'.receives)=letters'->t'''->s'.ch2
  )
  implies (  (t''!= t''' and  t=t') and (s'.client2) in s'.interseptMacs and (s.client1) !in s.interseptMacs or
               (t''= t''' and  t!=t') and (s.client1) in s.interseptMacs and (s'.client2) !in s'.interseptMacs or
               (t''= t''' and  t=t') and (s.client1) !in s.interseptMacs  and (s'.client2) !in s'.interseptMacs) )
  }
```

In the second fact (Lines 6-8) we need to constrain that, the *truth* of the first fact implies that the data (indices'/ letters') the server receives equals the data (indices/letters) the client sends, i.e achieving the assertion if one of the channels has been intercepted causing a delay time in one channel and as a result data is unreadable.


```
fact {
 all s, s': Communication_Status, t, t', t'', t''': Time,
        indices, indices', letters, letters': Data  |
 ((s.client1.(s.sends) = indices->t->s.ch1   and
    s.server.(s.receives) = indices'->t'->s.ch1) and
     (s'.client2.(s'.sends)=letters->t''->s'.ch2   and
    s'.server.(s'.receives)=letters'->t'''->s'.ch2) and
   ((t''!= t''' and  t=t') or (t''= t''' and  t=t') or
  (t''= t''' and  t!=t') ) )
  implies  ( letters=letters'  and indices=indices')
 }
```

In the third fact (Lines 9,10) we need to constrain that, for all Communication_Status, no two MACs (client1's mac, client2's mac, or server's mac) are equals. Macs should be unique.


```
fact {
all s',s: Communication_Status |
 s.client1 != s.server  and // never two actors have the same MAC addres
 s'.client2 != s.server and
 s.client1 != s.client2 and
s.client1=s'.client1 and
s.server = s'.server
 }
```

6.7 Results

In this section, we present the results we achieved from modelling, analysing, and checking the properties of the second protocol using the Alloy Analyser, and bounded SAT solver.

After running the model, we got that the predicate **MultiChannel** is consistent and there are instances that are available according to the finite scopes. This analysis is implemented with respect to a bounded scope of 3 atoms of *MAC*, 4 atoms of *Time*, 2 atoms of *ISP*, 2 atoms of *Channel*, 4 atoms of *Data*, 1 atom of *ConnectionStatus*, 2 atoms of *Communication_Status* in which only a finite number of elements for each type is taken into account; and therefore absence of an instance does not include checking satisfiability.

The Alloy Analyser spent 1.703s checking the properties of the third protocol. These properties are data (indices) and (letters) transmit over two different channels, only one channel is able to be intercepted during the two communication statuses, i.e no two channels have been intercepted, the MAC address is changeable and there are no MAC addresses that are equal, there is one MitM, opening two channels to exchange data between two hosts, one channel for exchanging letters and the other for exchanging indices, and the time for sending and receiving data over the opened channel has not been delayed which implies that received data by the server equals sent data from the client.

The automatic check detects that there is a counterexample to the assertion as shown in Figure 6.5. The result is acceptable because the analyser found an instance that satisfied the specification but violated the *DataSecure* assertion. We conclude that the properties that have been analysed do not hold in the model within the provided scopes.

Figure 6.5, shows two statuses, *status1* and *status2*. The first status shows that status2 *mac1* and server have the same status *First_Communication _And_Exchanging_Indices*. *Client1* and server belong to the visitors, while *client2* does not. *Client1* connects to the first *ispA*. The server connects to an *isp*. The first channel *ch1* is opened between two hosts to exchange indices. A *MitM* intercepts *client1*.

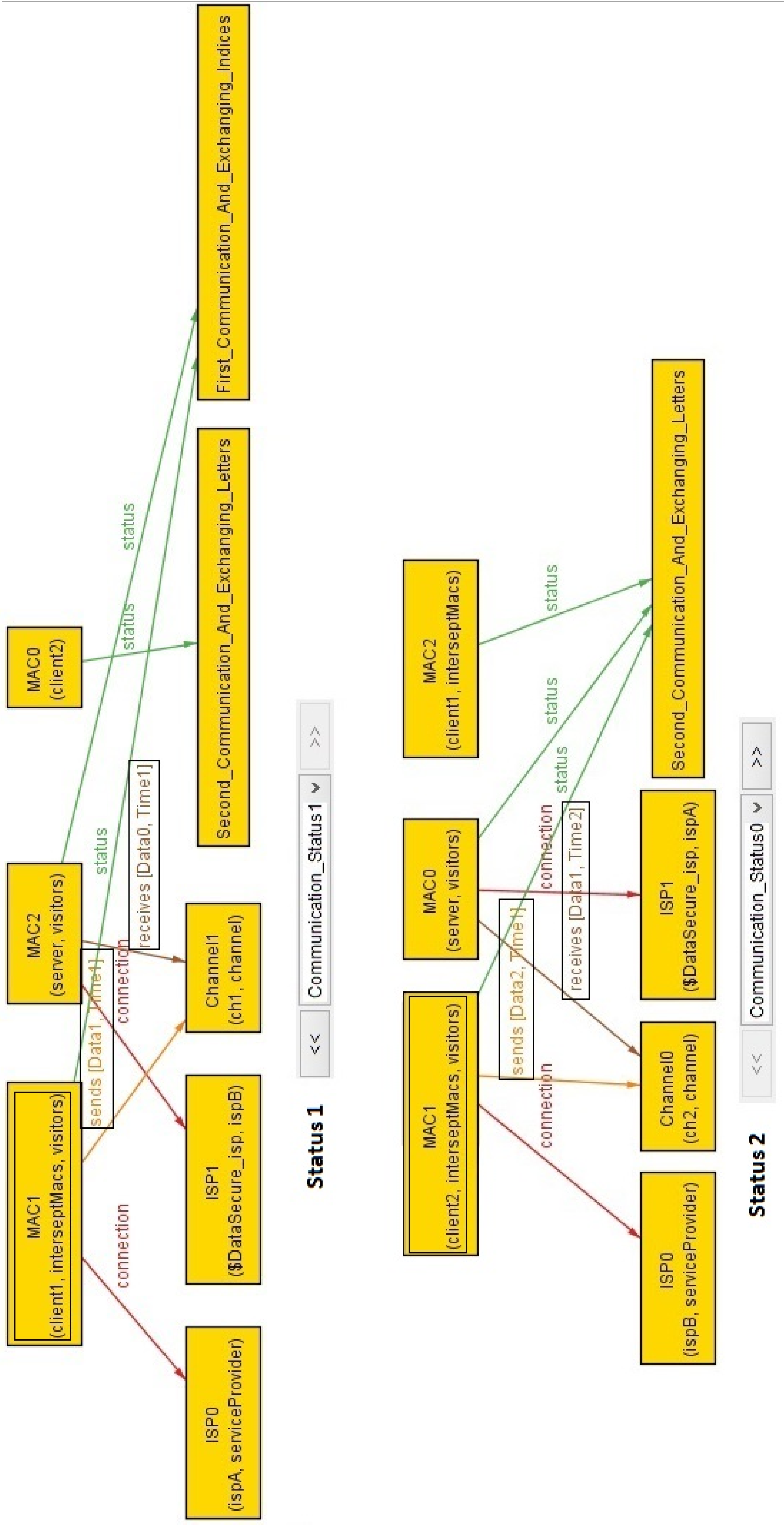

MAC0 (client2)
MAC2 (server, visitors)
MAC1 (client1, interseptMacs, visitors)
ISP0 (ispA, serviceProvider)
ISP1 ($DataSecure_isp, ispB)
Channel1 (ch1, channel)
Second_Communication_And_Exchanging_Letters
First_Communication_And_Exchanging_Indices
status
connection
sends [Data1, Time1]
receives [Data0, Time1]
<<
Communication_Status1
>>
Status 1
MAC2 (client1, interseptMacs)
MAC0 (server, visitors)
MAC1 (client2, interseptMacs, visitors)
ISP0 (ispB, serviceProvider)
Channel0 (ch2, channel)
ISP1 ($DataSecure_isp, ispA)
Second_Communication_And_Exchanging_Letters
status
connection
sends [Data2, Time1]
receives [Data1, Time2]
<<
Communication_Status0
>>
Status 2

Figure 6.5: An Instance of Generated A Counterexample

We notice that, the *client1* sends *data1* at *time1*, and the server receives different data *data0* that has been sent at same time *time1*. The interception of the *MitM* led to receiving data not being equal to the sending data. However, we notice that sending and receiving time still remained the same which conflicts with our assumption that they should be different, and here is the counterexample.

The second status shows that *client2 mac1* and server have the same status *Second_Communication_And_Exchanging_Letters*. *Client2* and server belong to the visitors, while *client1* does not. *Client2* connects to the second *ispB*. The server connects to an *isp*. The second channel *ch2* is opened between two hosts to exchange letters. A MitM intercepts *client2*.

We notice that, the *client2* sends *data2* at *time1*, and the server receives different data *data1* that has been sent at a different sending time *time2*. The interception of the *MitM* led to receiving data not being equal to the sending data and the time for sending and receiving data are not equal as well. However, we notice that because the client did not change the MAC address when moving from *status1* to *status2* it may be noticeable by the *MitM*, the *MitM* could intercept it again and get the data and match between letters and indices, and here is the counterexample.

So, from the figure, we did not achieved the sixth property in the first status which is that the time for sending and receiving data over the opened channel has not been delayed which implies that received data by the server equals sent data from the client. Also, we did not achieve the second property in the second status which is that only one channel is able to be intercepted during the two communication statuses, and the third property in the second status which is that the MAC address is changeable and there is no MAC addresses are equals. The result we achieved showed that system properties do not satisfy system requirements.

By examining the trace of the instance that SAT solver found, the counterexample is as shown in Figure 6.5. We can see that, the sixth property does not hold: time for exchanging data (indices) still remains the same despite the presence of a MitM which intercepted the channel that exchanged data in the first status, and exchanging data (indices) are not equal as well. Thus, an interception took place but there is no delay so the data that has been received does not equal the data that has been sent. Also, the MAC addresses in two channels did not change which conflict with the third property.

To achieve the model goal in the assertion, we need to add constraints or properties as facts as seen in Appendix (E.3). After adding the three facts, if no such counterexample has been found it is still possible that one could exist in a larger scope. However the Alloy Analyser spent 1.594s to gain the result that there is no counterexample even with the large scope in checking this assertion.

After correcting the counterexample, we now explore the model further to identify the problem in three hypotheses. The first hypotheses is that when a MitM intercept the first channel but not the second channel: as seen in Figure 6.6, the first status shows that *client1 mac2* and *server mac0* are in the same status *First_Communication_And_Exchanging_Indices*. *Client1* and *server* belong to the visitors, while *client2* does not. *Client1* connects to the first *ispA*. The server connects to an *isp*. The first channel *ch1* is opened between two hosts to exchange indices. A MitM intercepts *client1*. We notice that, as we see, the *client1* sends *data2* at *time3*, and the sever receives the same data *data2* that has been sent at a different sending time *time2*. The interception of the MitM led to receiving data being equal to the sending data assuming that the time for sending and receiving data are not equal as well because of the interception and eavesdrop data. We notice that our assumption is achieved and the time is different because of the interception of a MitM, and there is no counterexample.

The second status shows that *client2 mac1* and *server mac0* have the same status *Second_Communication_And_Exchanging_Letters*. *Client2* and *server* belongs to the visitors, while *client1* does not. *Client2* connects to the second *ispB*. The server connects to an *isp*. The second channel *ch2* is opened between two hosts to exchange letters. There is no MitM which intercepts *client2*. We notice that the *client2* sends *data1* at *time3*, and the server receives the same data *data1* at the same sending time *time3*. The non existence of the interception of the MitM led to the receiving data being equal to the sending data, and the time for sending and receiving data are equal. We notice that because the client changed the MAC address when moving from *status1* to *status2* this lead to it not being noticeable by the MitM, MitM could not intercept it again and get the data and match between letters and indices, and there is no counterexample.

The second hypotheses is that when a MitM intercept the second channel but not the first channel : as seen in Figure 6.7, the first status shows that *client1 mac2* and server *mac0* are in the same status *First_Communication_And_Exchanging_Indices*. *Client1* and *server* belong to the *visitors*, while *client2* does not. *Client1* connects to the first *ispA*. The server connects to an *isp*. The first channel *ch1* is opened between two hosts to exchange indices. No MitM intercepts *client1*. We notice that, as we see, the *client1* sends *data3* at *time3*, and the sever receives the same data *data3* at the same sending time (time3). The non existence of the interception of the MitM led to receiving data being equal to the sending data assuming that the time for sending and receiving data are equal as well. We notice that our assumption is achieved and the time is equal because there is no interception by a MitM, and there is no counterexample.

The second status shows that *client2 mac1* and server *mac0* are in the same status *Second_Communication_And_Exchanging_Letters*. *Client2* and *server* belong to the *visitors*, while *client1* does not. *Client2* connects to the second *ispB*. The server connects to an isp. The second channel *ch2* is opened between two hosts to exchange letters. A MitM intercepts *client2*. We notice that the *client2* sends *data2* at *time2*, and the server receives the same data *data2* at a different

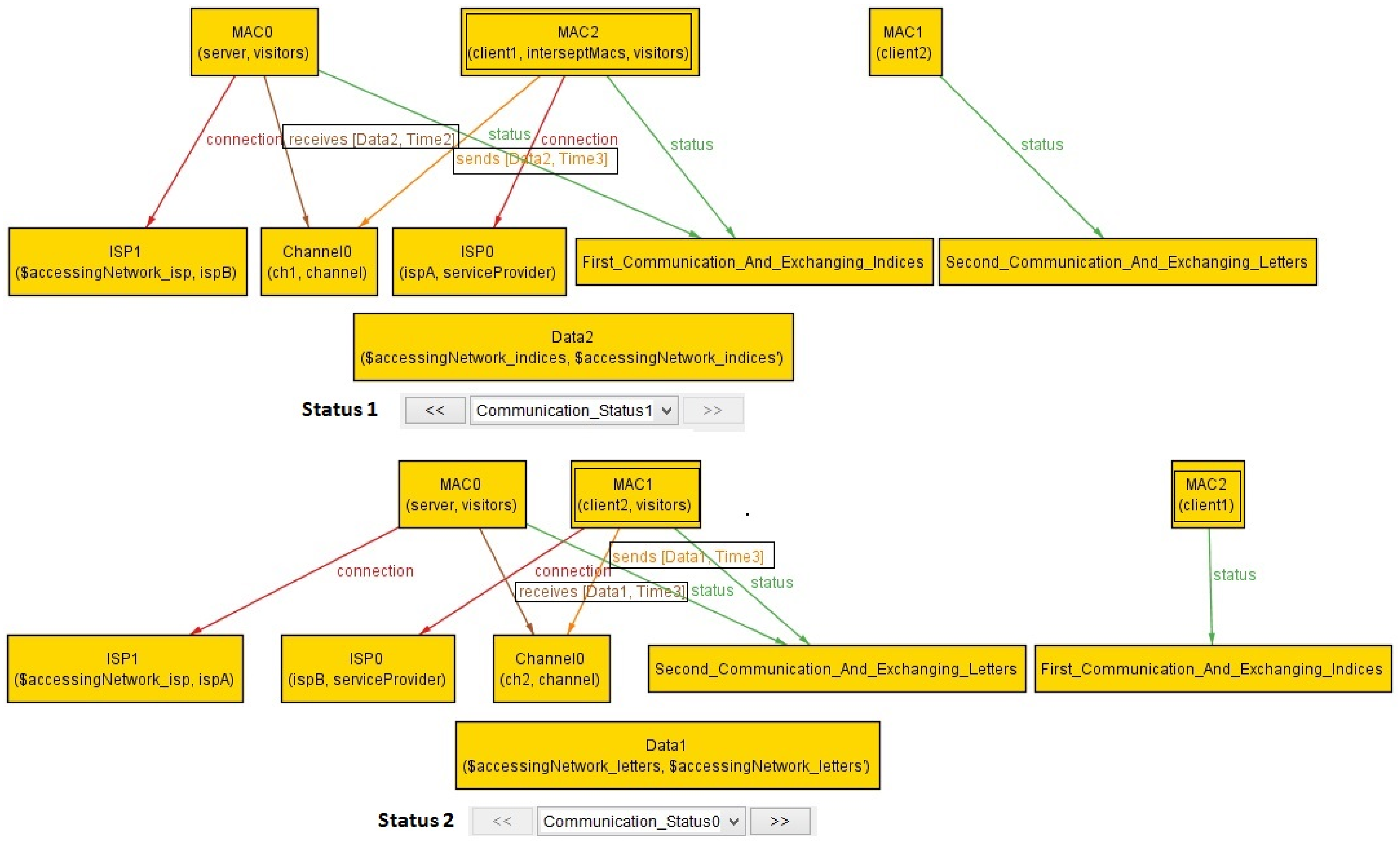


Figure 6.6: An Instance of Meta Model First Hypothesis: There is A MitM Intercepts the First Channel, While The Second Channel Has Not Been Intercepted in The Second Status

sending time *time1*. The existence of the interception of the MitM led to receiving data being equal to the sending data and the time for sending and receiving data are different. We notice that our assumption is achieved and the time is different because there is an interception by MitM, and there is no counterexample.

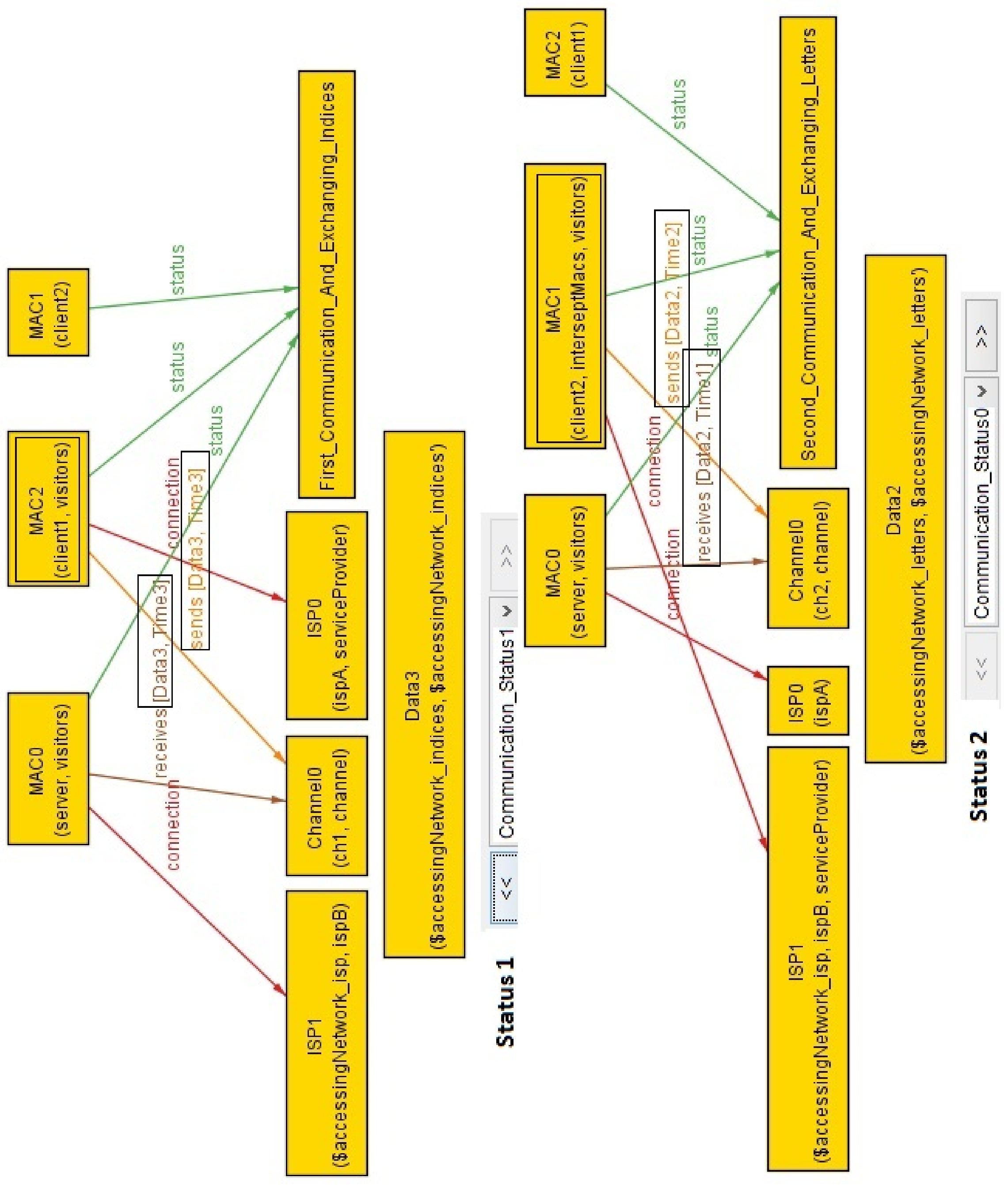


Figure 6.7: An Instance of Meta Model Second Hypothesis: There is A MitM Intercepts the Second Channel, While The First Channel Has Not Been Intercepted in The First Status

The third hypotheses is that when a MitM neither intercepts the first channel nor the second channel: as seen in Figure 6.8, the first status shows that *client1 mac2* and *server mac0* are in the same status *First_Communication_And_Exchanging_Indices*. *Client1* and *server* belong to the *visitors*, while *client2* does not. *Client1* connects to the first *ispA*. The server connects to an *isp*. The first channel *ch1* is opened between two hosts to exchange indices. No MitM intercepts *client1*. We notice that the *client1* sends *data1* at *time1*, and the server receives the same data *data1* at the same sending time *time2*. The non existence of the interception of the MitM led to receiving data being equal to the sending data assuming that the time for sending and receiving data are equal as well. We notice that our assumption is achieved and the time is equal because there is no interception by a MitM, and there is no counterexample.

The second status shows that *client2 mac1* and *server mac0* are in the same status *Second_Communication_And_Exchanging_Letters*. *Client2* and *server* belong to the *visitors*, while *client1* does not. *Client2* connects to the second *ispB*. The server connects to an *isp*. The second channel *ch2* is opened between two hosts to exchange letters. There is no MitM which intercepts *client2*. We notice that the *client2* sends *data0* at *time0*, and the sever receives the same data *data0* at the same sending time *time0*. The no existence of the interception of the MitM led to receiving data being equal to the sending data and the time for sending and receiving data are equal as well. We notice that our assumption is achieved and the time is equal because there is no interception by a MitM, and there is no counterexample.

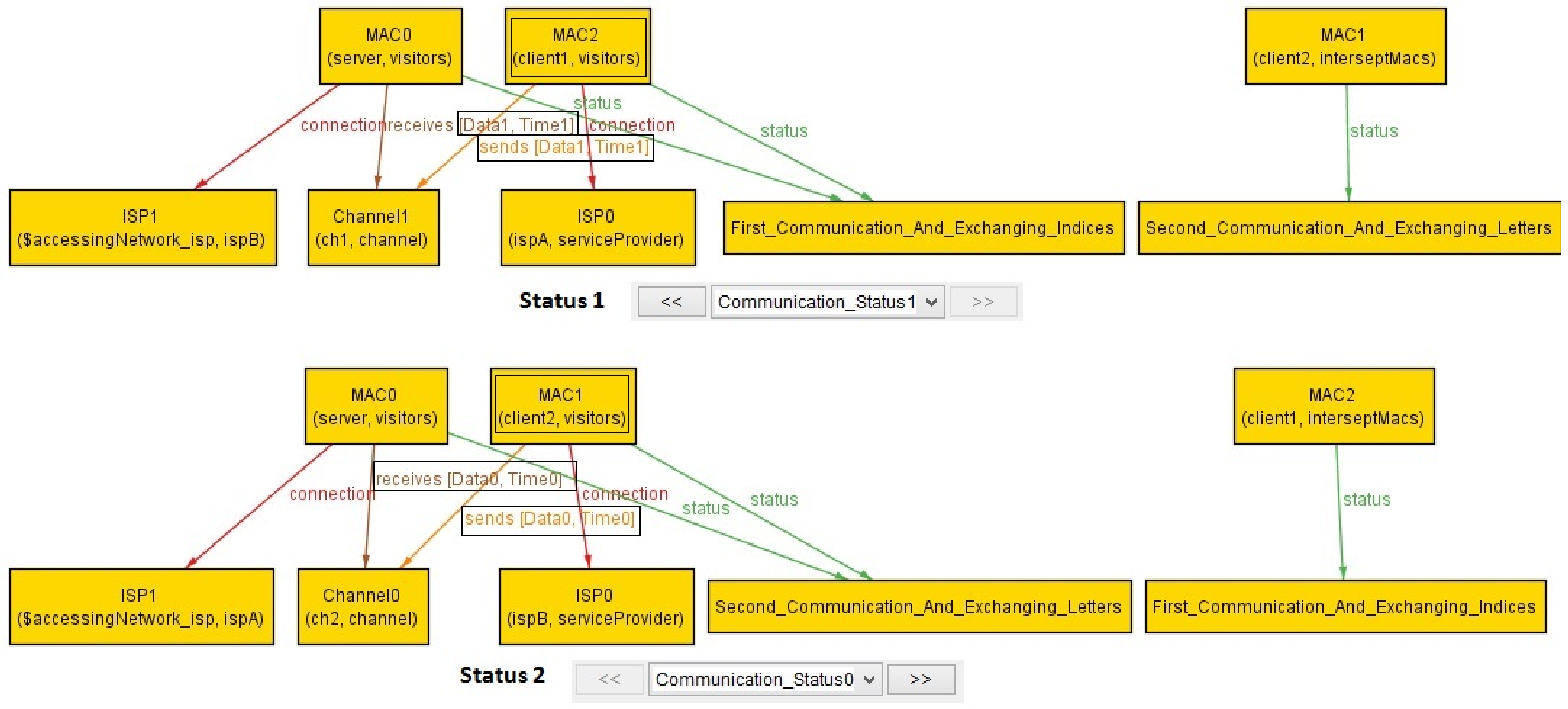


Figure 6.8: An Instance of Meta Model Third Hypothesis: Neither the First Channel nor The Second Channel Have Been Intercepted in Both Statuses

Chapter 7

Multichannel Security Protocol Proving Using Z3

7.1 Introduction

In this chapter, we translate the specification of the protocols from Alloy into Z3 FOL such that if there is a counterexample in Alloy in finite scopes, this is supposed to be a counterexample in Z3 in infinite scopes and vice versa. We must ensure that our formulation of the two specifications in Z3 are satisfiability equivalent to Alloy which means if a formula in Alloy is satisfiable, then the formula in Z3 is satisfiable and vice versa. We choose this methods as it is easy and we can trust it to provide good results [63]. It should be noted that the translation presented within this thesis is has not been formally verified.

7.1.1 First Protocol: Transmitting Data Over Single Channel

7.1.1.1 Type and Subtype Declarations

In Appendix (D) we give full details of the Z3 models with annotations to show the equivalent Alloy.

The hierarchical type system is translated implicitly. However, because the SMT language does not support subtypes, we use membership functions to enforce type hierarchy declarations. Consequently, top-level types are translated to uninterpreted sorts, while the top-level (super-type) of a type is translated to an uninterpreted membership function *isType* to indicate which elements of the super-type belongs to the type. It is not necessarily to declare the membership functions of top-level types, but we declared them to determine the semantics of the subtype.

As seen in Appendix (D.1), top-level types *Time*, *Channel*, *Card*, *ISP*, *Data*, *ConnectionStatus*, *MAC*, and *CommunicationStatus* are declared as uninterpreted sorts Lines (1-7). The membership function in Appendix (D.2) Line (11) *isConnection_And_Exchanging_Data* is declared to specify the semantics of subtypes *Connection_And_Exchanging_Data*.

7.1.1.2 Properties Of The Sub-signatures

As seen in Appendix (D.5), (Line 1) we adjust the returns types of the “oneOf ” functions/constants by specifying that each return a value of type *ConnectionStatus*. For example: line (1) calls function *oneOf_Connection_And_Exchanging_Data* in Line (1) in Appendix (D.3) which calls function *isConnection_And_Exchanging_Data* in Line (11) in Appendix (D.2) which returns *one* Operation type *isConnection_And_Exchanging_Data* in Line (11) Appendix (D.2) which is already specified as a return type in Line (1) Appendix (D.5).

Sub-signatures (Lines 1) Appendix (D.3) declare functions of the property *some* for the sub signature *Connection_And_Exchanging_Data* of the super signature *ConnectionStatus* because *ConnectionStatus* has at least one element of the sub signatures. In (Line 1) Appendix (D.6) we need to assert the *lone* property of the previous sub signature because *ConnectionStatus* has at most one element of the sub signature. The assertion (Line 1) Appendix (D.6) is expressed to the formula A below:

Formula A:

$\forall$ *o1, o2: ConnectionStatus.(o1* $\in$ *isConnection_And_Exchanging_Data* $\wedge$ *o2* $\in$ *isConnection_And_Exchanging_Data) =>o1 = o2*

The formula A specifies constrains that for each *ConnectionStatus*, there is at most only one corresponding *ConnectionStatus*: if there exist two *Connection-Status* belonging to *isConnection_And_Exchanging_Data* for example, then these two *ConnectionStatus* should be equal. i.e we restrict the characteristics of *lone ConnectionStatus* of type *isConnection_And_Exchanging_Data* for each status to avoid the inconsistency.

7.1.1.3 Abstraction and Extension

In the first protocol, we do not need to prove the abstract and extension formulas as we have only one status. The expression and the prove of the abstract and extension will be developed in the protocol which contains two statuses.

7.1.1.4 Facts

Facts are assumed to be true. They represents the protocol properties. As seen in Appendix (D.8), Line (1) declares quantifiers to restrict the first fact. Line (2) declares quantifiers to restrict the second fact. Line (3) declares quantifiers to restrict the third fact.

The first fact as seen below is expressed in formula B below:

First fact:

```
(forall ((status1 CommunicationStatus))(implies(forall ((cmac MAC)(mmac MAC))
(or(=>(mitmIntercepts status1 mmac)(not(visitors status1 mmac)))
(and(=>(mitmIntercepts status1 mmac)(visitors status1 mmac))
(=>(client1 status1 cmac)(not(mitmIntercepts status1 cmac))))))
(forall ((t Time)(t1 Time)(d Data)(d1 Data)(ch1 Channel))
(implies (and (forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= d d11)(= t t11)(= ch1 ch11))))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 d t ch1)))
(forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= d1 d11)(= t1 t11) (= ch1 ch11))))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 d1 t1 ch1))))(= t t1)))))
```

Formula B:

$(\forall$ *status1:Communication_Status, cmac,mmac:MAC.*$((status1,mmac) \in$
$(mitmIntercepts \Rightarrow (status1,mmac) \notin visitors \vee$
$((status1,mmac) \in mitmIntercepts \Rightarrow (status1,mmac) \in visitors$
$\wedge\ (status1,cmac) \in client1 \Rightarrow (status1,cmac) \notin mitmIntercepts) \Rightarrow$
$\forall\ t,t1:Time,\ d,d1:Data,\ ch1:Channel$
$(\ \forall\ t11:Time,\ d11:Data,\ ch11:Channel$
$\exists\ cmac:MAC.(status1,cmac) \in client1$
$\wedge\ (status1,cmac,d11,t11,ch11) \in sends \Rightarrow$
$(d = d11) \wedge (t = t11) \wedge (ch1 = ch11))) \wedge$
$\exists\ cmac1:MAC.(status1,cmac1) \in client1$
$\wedge\ (status1,cmac1,d,t,ch1) \in sends \wedge$
$\forall\ t11:Time,\ d11:Data,\ ch11:Channel$
$\exists\ smac:MAC.(status1,smac) \in server$
$\wedge\ (status1,smac,d11,t11,ch11) \in receives \Rightarrow$
$(d1 = d11) \wedge (t1 = t11) \wedge (ch1 = ch11))) \wedge$
$\exists\ smac1:MAC.(status1,smac1) \in server$
$\wedge\ (status1,smac1,d1,t1,ch1) \in receives\ ) \Rightarrow t = t1\ )$

The formula B specifies constraints that: for all atoms *status1* in *Communication_Status*, *cmac* (client's mac) in *MAC*, and *mmac* (MitM's mac) in *MAC* such that if a MitM *mmac* belongs to *mitmIntercepts* in the *status*, then the MitM *mmac* does not belong to the *visitors* in the *status* (secure scope), or if the MitM *mmac* belongs to *mitmIntercepts* in the *status*, then the MitM *mmac* belongs to the *visitors* in the *status*, and if a client *cmac* belongs to *client1* in the *status*, then the client *cmac* does not belong to *mitmIntercepts* in the *status*.

It follows that for all atoms *t* and *t1* in *Time*; *d* and *d1* in *Data*; and *ch1* in *Channel*, for all atoms *t11* in *Time*; *d11* in *Data*; and *ch11* in *Channel*, if there exists *cmac* in *MAC* such that, *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends*, then *d* equals *d11*, *t* equals *t11*, and *ch1* equals *ch11*. So, if there exists *cmac1* in *MAC*, then *status1* and *cmac1* belong to *client1*; and *status1*, *cmac1*, *d*, *t*, and *ch1* belong to *sends*.

And for all atoms *t11* in *Time*; *d11* in *Data*; and *ch11* in *Channel*, if there exists *smac* in *MAC* such that, *status1* and *smac* belong to *server*; and *status1*, *smac*, *d11*, *t11*, and *ch11* belong to *receives*, then *d1* equals *d11*, *t1* equals *t11*, and *ch1* equals *ch11*. So, if there exists *smac1* in *MAC*, then *status1* and *smac1* belong to *server*; and *status1*, *smac1*, *d1*, *t1*, and *ch1* belong to *receives*; then *t* equals *t1*.

The second fact as seen below is expressed to formula C below:

Second fact:

```
 (forall ((t Time)(t1 Time)(d Data)(d1 Data)(ch1 Channel))
(implies (and (forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= d d11)(= t t11)(= ch1 ch11))))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 d t ch1)))
(forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= d1 d11)(= t1 t11) (= ch1 ch11))))
```

(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 d1 t1 ch1))))(= t t1))(= d d1)))

Formula C:

$\forall$ *t,t1:Time, d,d1:Data, ch1:Channel*
($\forall$ *t11:Time, d11:Data, ch11:Channel*
$\exists$ *cmac:MAC.(status1,cmac)* $\in$ *client1*
$\wedge$ *(status1,cmac,d11,t11,ch11)* $\in$ *sends =>*
(d = d11) $\wedge$ *(t = t11)* $\wedge$ *(ch1 = ch11)))* $\wedge$
$\exists$ *cmac1:MAC.(status1,cmac1)* $\in$ *client1*
$\wedge$ *(status1,cmac1,d,t,ch1)* $\in$ *sends* $\wedge$
$\forall$ *t11:Time, d11:Data, ch11:Channel*
$\exists$ *smac:MAC.(status1,smac)* $\in$ *server*
$\wedge$ *(status1,smac,d11,t11,ch11)* $\in$ *receives =>*
(d1 = d11) $\wedge$ *(t1 = t11)* $\wedge$ *(ch1 = ch11)))* $\wedge$
$\exists$ *smac1:MAC.(status1,smac1)* $\in$ *server*
$\wedge$ *(status1,smac1,d1,t1,ch1)* $\in$ *receives* $\wedge$
t = t1) =>d = d1

The formula C specifies constraints that: for all atoms *t* and *t1* in *Time*; *d* and *d1* in *Data*; and *ch1* in *Channel*, for all atoms *t11* in *Time*; *d11* in *Data*; and *ch11* in *Channel*, if there exists *cmac* in *MAC* such that, *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends*, then *d* equals *d11*, *t* equals *t11*, and *ch1* equals *ch11*. So, if there exists *cmac1* in *MAC*, then *status1* and *cmac1* belong to *client1*; and *status1*, *cmac1*, *d*, *t*, and *ch1* belong to *sends*.

And for all atoms *t11* in *Time*; *d11* in *Data*; and *ch11* in *Channel*, if there exists *smac* in *MAC* such that, *status1* and *smac* belong to *server*; and *status1*, *smac*, *d11*, *t11*, and *ch11* belong to *receives*, then *d1* equals *d11*, *t1* equals *t11*, and *ch1* equals *ch11*. So, if there exists *smac1* in *MAC*, then *status1* and *smac1* belong to *server*; and *status1*, *smac1*, *d1*, *t1*, and *ch1* belong to *receives*; and *t* equals *t1*, then *d* equals *d1*

The third fact as seen below is expressed to formula D below:

Third fact:

(forall ((s CommunicationStatus))(forall ((mac1 MAC) (mac2 MAC)) (=>(and(client1 s mac1) (server s mac2))(not(= mac1 mac2)))))

Formula D:

$\forall$ *s:Communication_Status, cmac1,cmac2:MAC.(s,cmac1)* $\in$ *client1* $\wedge$ *(s,cmac2)* $\in$ *server =>cmac1* $\neq$ *cmac2*

The formula D specifies constraints that: all atoms *s* in *Communication_Status*, *cmac1* (client1's mac) in *MAC* and *cmac2* (server's mac) in *MAC* such that client1's mac in *status1* is not equal to server's mac in *status1.*

7.1.1.5 Relation Declarations

Relations are translated to Boolean-valued, uninterpreted, membership functions. As seen in Appendix (D.2) these functions are declared over top-level types because only top-level types are declared as sorts. Because all SMT functions are total function, relations are specified utilizing three parts: function name, received sorts, and returned value of Boolean type. The Boolean type includes two kinds of value, true for the tuples that are involved in the declared relation, or false for all others that are not involved.

Lines (1-10) Appendix (D.2) declare functions of relations. These relations are: *serviceProvider*, which is declared as Boolean-valued function over *Communication_Status*; *ISP* (Line 1), *visitors*, *client*,*server*, and *mitmIntercepts* which are declared as Boolean-valued function over *Communication_Status* and *MAC* (Lines 2-5); *connection* which is declared as Boolean-valued functions over *Communication_Status*, *MAC*, and *ISP* (Line 6); *opens* which is declared as Boolean-valued functions over *Communication_Status*, *MAC*, *ISP*, and *Channel* (Line 7); *status* which is declared as Boolean-valued functions over *Communication_Status*, *MAC*, and *ConnectionStatus* (Line 8); *sends, receives* which are declared as Boolean-valued functions over *Communication_Status*, *MAC*,*Data*,*Time*, and *Channel* (Lines 9,10).

Lines (1-8) Appendix (D.7) declare more constraints that guarantee that each relation is defined for its specific types considering the multiplicity keywords constraints.

The relations *serviceProvider: set ISP* and *visitors:set MAC* are not required to be translated to formula as the *set* keyword constrains and allows any number of elements. Thus, its defined Boolean-valued function (Lines 1, and 2) Appendix (D.2) is equivalent to their meaning.

Line (1) for the relation below and as seen in Appendix (D.2) is expressed in formula E below:

client: one MAC

(forall ((cs CommunicationStatus))(and (exists ((mac1 MAC))(client1 cs mac1))
(forall ((mac3 MAC)(mac2 MAC))(=>(and(client1 cs mac2)(client1 cs mac3))
(= mac2 mac3)))))

Formula E:

$\forall$ *cs:CommunicationStatus* $\exists$ *mac1:MAC.(cs,mac1)* $\in$
client1 $\wedge$ $\forall$ *mac3,mac2:MAC.(cs,mac2)* $\in$ *client1*
$\wedge$ *(cs,mac3)* $\in$ *client1 =>mac2 = mac3*

The formula E specifies constraints that: for all atoms *cs* in *Communication-Status*, if there exists *mac1* in *MAC*, such that, the *cs* and *mac1* belong to the first client *client1*, and for all *mac3* and *mac2*:MAC, such that *cs* and *mac2* belong to the client (client1); and *cs* and *mac3* belong to the client (client1) then *mac2* equals *mac3*. To restrict the multiplicity to one, we suppose that if two macs belong to the client *client1*, thus these two macs are equal.

Line (2) for the relation below and as seen in Appendix (D.2) is expressed in formula F below:

server: one MAC

(forall ((cs CommunicationStatus))(and(exists ((mac1 MAC))(server cs mac1))
(forall ((mac3 MAC)(mac2 MAC))(=>(and (server cs mac2)(server cs mac3))
(= mac2 mac3)))))

Formula F:

$\forall$ *cs:CommunicationStatus* $\exists$ *mac1:MAC.(cs,mac1)* $\in$ *server* $\wedge$
$\forall$ *mac3,mac2:MAC.(cs,mac2)* $\in$ *server* $\wedge$
(cs,mac3) $\in$ *server =>mac2 = mac3*

The formula F constrains that: for all atoms *cs* in *CommunicationStatus*, if there exists *mac1* in *MAC*, such that the *cs* and *mac1* belong to the server, and for all *mac3* and *mac2*:MAC, such that *cs* and *mac2* belong to the *server*; and *cs* and *mac3* belong to the *server* then *mac2* equals *mac3*. To restrict the multiplicity to one, we suppose that if there are two macs belonging to the server, these two macs are equal.

Line (3) for the relation below and as seen in Appendix (D.2) is expressed in formula G below:

mitmIntercepts: lone MAC-server

*(forall ((cs CommunicationStatus)(mac3 MAC)(mac2 MAC))(=>(and (mitmIntercepts cs mac2)(mitmIntercepts cs mac3))(= mac2 mac3)))
(forall ((cs CommunicationStatus))(exists ((mac1 MAC))
(not(mitmIntercepts cs mac1))))*

Formula G:

$\forall$ *cs:CommunicationStatus, mac3,mac2:MAC.(cs,mac2)* $\in$ *mitmIntercepts* $\wedge$ *(cs,mac3)* $\in$ *mitmIntercepts =>mac2 = mac3*
$\forall$ *cs:CommunicationStatus* $\exists mac1 : MAC.(cs, mac1) \notin$ *mitmIntercepts*

The formula G specifies constraints that: for all atoms *cs* in *CommunicationStatus*, *mac2* and *mac3* in *MAC*, such that the *cs* and *mac2* belong to the *mitmIntercepts*; and *cs* and *mac3* belong to the *mitmIntercepts*, then *mac2* equals *mac3.* For all *cs* in *CommunicationStatus*, there exists *mac1* in MAC, such that *cs* and *mac1* do not belong to the *mitmIntercepts.* To restrict the multiplicity to one, we supposed that if there are two macs belong to the *mitmIntercepts*, these two macs are equal.

Line (4) for the relation below and as seen in Appendix (D.2) is expressed in formula H below:

connection: MAC ->lone serviceProvider

*(forall ((cs CommunicationStatus)(mac MAC))(and(forall ((isp11 ISP))
(=>(connection cs mac isp11)(serviceProvider cs isp11)))
(forall ((isp12 ISP)(isp13 ISP))(=>(and(connection cs mac isp12)
(connection cs mac isp13))(= isp12 isp13)))))*

Formula H:

$\forall$ *cs:CommunicationStatus,mac:MAC*
($\forall$ *isp11:ISP.(cs,mac,isp11)* $\in$ *connection =>*
(cs,isp11) $\in$ *serviceProvider* $\wedge$

∀ isp12,isp13:ISP.(cs,mac,isp12) ∈ connection ∧ (cs,mac,isp13) ∈ connection =>isp12 = isp13)

The formula H specifies constraints that: for all atoms *cs* in *Communication-Status*, *mac* in *MAC*; for all *isp11*:ISP, such that the *cs*, *mac*, and *isp11* belong to *connection*, then *cs* and *isp11* belong to *serviceProvider*, and for all *isp12* and *isp13* in ISP, such that the *cs*, *mac*, and *isp12* belong to *connection*, and *cs*, *mac*, and *isp13* belong to *connection*, then that *isp12* equals *isp13*. To restrict the multiplicity to lone we supposed that if there are two *isp* in the *connection*, they are equal.

Line (5) for the relation below and as seen in Appendix (D.2) is expressed in formula I below:

---

status:MAC ->one ConnectionStatus

(forall ((cs CommunicationStatus)(mac MAC))(and(exists ((co ConnectionStatus))
(status cs mac co))
(forall ((a3 ConnectionStatus)(a2 ConnectionStatus))
(=>(and(status cs mac a2)(status cs mac a3))(= a2 a3)))))

Formula I:

∀ cs:CommunicationStatus,mac:MAC ∃ o:ConnectionStatus.(cs,mac,co) ∈ status ∧ ∀ a2,a3:ConnectionStatus.(cs,mac,a2) ∈ status ∧ (cs,mac,a3) ∈ status =>(a2 = a3)

The formula I specifics constraints that: for all atoms *cs* in *Communication-Status* and *mac* in *MAC*, if there exists an atom *o* in *ConnectionStatus* such that *cs*, *mac*, and *co* belong to *status*, and for all atoms *a2* and *a3* in *Communication-Status* such that *cs*, *mac*, and *a2* belong to *status* and *cs*, *mac*, and *a3* belong to *status* then *a1* equals *a2*. To restrict the multiplicity to one we supposed that if there are two *ConnectionStatus*, thus they are equal.

Line (6) for the relation below and as seen in Appendix (D.2) is expressed in formula J below:

opens :MAC ->serviceProvider ->Channel

(forall ((cs CommunicationStatus) (mac MAC)(isp ISP)(ch Channel))
(=>(opens cs mac isp ch)(serviceProvider cs isp)))

Formula J:

$\forall$ *cs:CommunicationStatus,mac:MAC, isp:ISP,*
ch:Channel.(cs,mac,isp,ch) $\in$ *opens =>(cs,isp)* $\in$ *serviceProvider*

The formula J specifies constraints that: for all atoms *cs* in *Communication-Status*, *mac* in *MAC*, *isp* in *ISP*, and *ch* in *Channel* such that *cs*, *mac*, *isp*,and *ch* belong to *open* then *cs* and *isp* belong to *serviceProvider* .

Line (7) for the relation below and as seen in Appendix (D.2) is expressed in formula K below:

sends: MAC ->lone (Data ->Time)->Channel

(forall ((cs CommunicationStatus)(mac MAC)(ch Channel)
(d1 Data)(d2 Data)(t Time)(t1 Time))(=>
(and(sends cs mac d1 t ch)(sends cs mac d2 t1 ch)) (and(= d1 d2)(= t t1))))

Formula K:

$\forall$ *cs:CommunicationStatus, mac:MAC,ch:Channel, t,t1:Time,*
d1,d2:Data.(cs mac d1 t ch) $\in$ *sends* $\wedge$
(cs mac d2 t1 ch) $\in$ *sends =>(d1 = d2)* $\wedge$ *(t = t1)*

The formula K specifies constraints that: for all atoms *cs* in *Communication-Status*, *mac* in *MAC*, *ch* in *Channel*, *t* and *t1* in *Time* and *d1* and *d2* in *Data* such that *cs*, *mac*, *d1*, *t*, and *ch* belong to *sends*; and *cs*, *mac*, *d2*, *t1*, and *ch* belong to *sends* then *d2* equals *d1* and *t* equals *t1*. To restrict the multiplicity to lone, we suppose that if there are two times and data belonging to the sends, thus these two times and datum are equal.

Line (8) for the relation below and as seen in Appendix (D.2) is expressed to formula L below:

receives: MAC ->lone (Data ->Time)->Channel

(forall ((cs CommunicationStatus)(mac MAC)(ch Channel),
(d1 Data)(d2 Data)(t Time)(t1 Time))(=>
(and(receives cs mac d1 t ch)(receives cs mac d2 t1 ch))
(and(= d1 d2)(= t t1))))

Formula L:

∀ *cs:CommunicationStatus, mac:MAC,ch:Channel* ∃ *t,t1:Time,*
d1,d2:Data.(cs mac d1 t ch) ∈ *receives* ∧
(cs mac d2 t1 ch) ∈ *receives =>(d1 = d2)* ∧ *(t = t1)*

The formula L specifies constraints that: for all atoms *cs* in *CommunicationStatus*, *mac* in *MAC*, *ch* in *Channel*, *t* and *t1* in *Time*; *d1* and *d2* in *Data* such that *cs*, *mac*, *d1*, *t*, and *ch* belong to *receives*; and *cs*, *mac*, *d2*, *t1*, and *ch* belong to *receives* then *d2* equals *d1* and *t* equals *t1*. To restrict the multiplicity to lone, we suppose that if there are two times and data belonging to the receives, thus these two times and datum are equal.

7.1.1.6 Predicates

As seen in Appendix (D.9), the translation focuses on "in-lining " of the predicate *SingleChannel*. In-lining means without explicit declaration. The *SingleChannel* protocol passes through one status.

Line (1) for the predicate below and in Appendix (D.9) is expressed in formula M below:

[t,t':Time, d,d':Data,isp,isp1:ISP, status1:Communication_Status ,ch:Channel]

(forall ((isp ISP)(isp1 ISP)(d Data)(d1 Data)(t Time)(t1 Time)
(ch1 Channel)(status1 CommunicationStatus))

Formula M:

∀ *status1:CommunicationStatus, isp,isp1:ISP, d,d1:Data, t,t1:Time, ch1:Channel*

The formula M declares all variables to be used in the predicate and assertion: *status1* for the status; *d* and *d1* for data that has been sent and received in the status; *t* and *t1* for times for sending and receiving data; *isp* for the internet service provider that the client connects with; *isp1* for the internet service provider that the server connects with; the channel *ch1* for exchange data.

These variables will be used directly in the predicates and assertion below without the need for redeclaration.

Line (2) for the predicate below and in Appendix (D.9) is expressed in formula N below:

status1.client1.(status1.status)=isConnection_And_Exchanging_Data

(=>(forall ((cs1 ConnectionStatus)(cmac MAC))
(=>(and (client1 status1 cmac)(status status1 cmac cs1))
(isConnection_And_Exchanging_Data cs1)))
(forall ((cs2 ConnectionStatus))(cmac MAC))
(=>(isConnection_And_Exchanging_Data cs2)
(and (client1 status1 cmac)(status status1 cmac cs2)))))

Formula N:

$(\forall$ *cs1:ConnectionStatus,cmac:MAC.(status1,cmac)* $\in$ *client1*
$\wedge$ *(status1,cmac,cs1)*$\in$ *status =>(cs1)* $\in$
isConnectionAndExchangingData) =>
($\forall$ *cs2:ConnectionStatus,cmac:MAC.(cs2)* $\in$
isConnectionAndExchangingData =>(status1,cmac) $\in$ *client1*
$\wedge$ *(status1,cmac,cs2)*$\in$ *status)*

The formula N specifics constraints that: for all atoms *cs1* in *ConnectionStatus*, and *cmac* in *MAC*, such that *status1* and *cmac* belong to *client1*, and *status1*, *cmac*, and *cs1* belong to status then *cs1* belongs to *isConnectionAndExchangingData*. It follows that for all atoms *cs2* in *ConnectionStatus*, and *cmac* in *MAC*, such that *cs2* belongs to *isConnectionAndExchangingData* then *status1* and *cmac* belong to *client1*, and *status1*, *cmac*, and *cs2* belong to status.

Line (3) for the predicate below and in Appendix (D.9) is expressed in formula O below:

status1.server.(status1.status)=isConnection_And_Exchanging_Data

```
(=>(forall ((cs1 ConnectionStatus),(smac MAC))
(=>(and (server status1 smac)(status status1 smac cs1))
(isConnection_And_Exchanging_Data cs1)))
(forall ((cs2 ConnectionStatus),(smac MAC))
(=>(isConnection_And_Exchanging_Data cs2)
(and (server status1 smac)(status status1 smac cs2)))))
```

Formula O:

$(\forall$ *cs1:ConnectionStatus,smac:MAC.(status1,smac)* $\in$ *server*
$\wedge$ *(status1,smac,cs1)*$\in$ *status =>(cs1)* $\in$
isConnectionAndExchangingData) =>
($\forall$ *cs2:ConnectionStatus,smac:MAC.(cs2)* $\in$
isConnectionAndExchangingData =>(status1,smac) $\in$ *server*
$\wedge$ *(status1,smac,cs2)*$\in$ *status)*

The formula O specifics constraints that: for all atoms *cs1* in *ConnectionStatus*, an *smac* in *MAC*, such that *status1* and *smac* belong to *server*, and *status1*, *smac*, and *cs1* belong to *status* then *cs1* belongs to *isConnectionAndExchangingData*. It follows that for all atoms *cs2* in *ConnectionStatus*, and *smac* in *MAC*, such that *cs2* belongs to *isConnectionAndExchangingData* then *status1* and *smac* belong to *server*, and *status1*, *smac*, and *cs2* belong to *status*.

Line (4) for the predicate below and in Appendix (D.9) is expressed in formula P below:

status1.client1 in status1.visitors

(forall ((cmac MAC))(=>(client1 status1 cmac)(visitors status1 cmac)))

Formula P:

(∀ cmac:MAC.(status1,cmac) ∈ client1 =>(status1,cmac) ∈ visitors

The formula P specifies constraints that: for all *cmac* in *MAC*, such that *status1* and *cmac* belong to *client1* then *status1* and *cmac* belong to visitors.

Line (5) for the predicate below and in Appendix (D.9) is expressed in formula Q below:

status1.server in status1.visitors

(forall ((smac MAC))(=>(client1 status1 smac)(visitors status1 smac)))

Formula Q:

(forall smac:MAC.(status1,smac) ∈ server =>
(status1,smac) ∈ visitors

The formula Q specifies constraints that: for all *smac* in *MAC*, such that *status1* and *smac* belong to *server* then *status1* and *smac* belong to visitors.

Line (6) for the predicate below and in Appendix (D.9) is expressed in formula R below:

no status1.client1.(status1.receives)

```
 (forall ((cmac MAC))
(and(not(and(client1 status1 cmac)(receives status1 cmac d1 t1 ch1)))
(not(and(client1 status1 cmac)(receives status1 cmac d t ch1)))))
```

Formula R:

(∀ cmac:MAC.(not(status1,cmac) ∈ client1 ∧
((status1,cmac,d1,t1,ch1) ∈ receives) ∧
(not(status1,cmac) ∈ client1 ∧
((status1,cmac,d,t,ch1) ∈ receives)

The formula R specifies constraints that: for all *cmac* in *MAC*, such that *status1* and *cmac* do not belong to *client1* and *status1*, *cmac*, *d1*, *t1*, and *ch1* belong to *receives*; and *status1* and *cmac* do not belong to *client1* and *status1*, *cmac*, *d*, *t*, and *ch1* belong to *receives*.

Line (7) for the predicate below and in Appendix (D.9) is expressed in formula S below:

no status1.server.(status1.sends)

```
 (forall ((smac MAC))
(and(not(and(server status1 smac)(sends status1 smac d1 t1 ch1)))
(not(and(server status1 smac)(sends status1 smac d t ch1)))))
```

Formula S:

(∀ smac:MAC.(not(status1,smac) ∈ server ∧
((status1,smac,d1,t1,ch1) ∈ sends) ∧
(not(status1,smac) ∈ server ∧
((status1,smac,d,t,ch1) ∈ sends)

The formula S specifies constraints that: for all *smac* in *MAC*, such that *status1* and *smac* do not belong to *server* and *status1*, *smac*, *d1*, *t1*, and *ch1* belong to *sends); and status1 and smac do not belong to server and status1, smac, d, t, and ch1 belong to sends.*

Line (8) for the predicate below and in Appendix (D.9) is expressed in formula T below:

status1.client1.(status1.connection)= isp

(forall ((isp11 ISP))(=>(=>(exists ((cmac MAC))(and (client1 status1 cmac)
(connection status1 cmac isp11)))(= isp isp11))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(connection status1 cmac1 isp)))))

Formula T:

(∀ isp11:ISP ∃ cmac:MAC.(status1,cmac) ∈ client1 ∧
(status1,cmac,isp11) ∈ connection =>(= isp isp11))
=>(∃ cmac1:MAC.(status1,cmac1) ∈ client1 ∧
(status1,cmac1,isp) ∈ connection

The formula T specifies constraints that:for all *isp11* in ISP, if there exist *cmac* in *MAC* such that the *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, and *isp11* belong to *connection* then *isp* equals *isp11.* It follows that there exists *cmac1* in *MAC* such that the *status1* and *cmac1* belong to *client1*; and *status1*, *cmac1*, and *isp* belong to *connection.*

Line (9) for the predicate below and in Appendix (D.9) is expressed in formula U below:

---

status1.server.(status1.connection)= isp'

---

*(forall ((isp11 ISP))(=>(=>(exists ((smac MAC))(and (server status1 smac)
(connection status1 smac isp11)))(= isp1 isp11))
(exists ((smac MAC))(and (server status1 smac)
(connection status1 smac isp1)))))*

Formula U:

$(\forall\ isp11{:}ISP\ \exists\ smac{:}MAC.(status1,smac) \in server \wedge$
$(status1,smac,isp11) \in connection => (= isp1\ isp11))$
$=>(\exists\ smac1{:}MAC.(status1,smac1) \in server \wedge$
$(status1,smac1,isp1) \in connection$

The formula U specifies constraints that:for all *isp11* in ISP, if there exist *smac* in *MAC* such that the *status1* and *smac* belong to *server*; and *status1*, *smac*, and *isp11* belong to *connection* then *isp1* equals *isp11*. It follows that there exists *smac1* in *MAC* such that the *status1* and *smac1* belong to *server*; and *status1*, *smac1*, and *isp1* belong to *connection.*

Line (10) for the predicate below and in Appendix (D.9) is expressed in formula V below:

status1.client1.(status1.opens)= status1.client.(status1.connection)->ch1

(forall ((isp11 ISP)(ch Channel))(=>(forall ((cmac MAC))(=>
(and (client1 status1 cmac)(connection status1 cmac isp11))
(and (client1 status1 cmac)(opens status1 cmac isp11 ch))))
(and(= isp isp11)(= ch1 ch))))(forall ((cmac1 MAC))(=>
(and (client1 status1 cmac1)(connection status1 cmac1 isp))
(and (client1 status1 cmac1)(opens status1 cmac1 isp ch1))))

Formula V:

(exists ((mac25 MAC)) *(∀ isp11:ISP, ch:Channel ∀ cmac:MAC.(status1,cmac) ∈ client1 ∧*
(status1,cmac,isp11) ∈ connection =>
(status1,cmac) ∈ client1 ∧ (status1,cmac,isp11,ch) ∈ opens
=> (isp = isp11) ∧ (ch1 = ch) ∧
(∀ cmac1:MAC.(status1,cmac1) ∈ client1 ∧
(status1,cmac1,isp) ∈ connection =>
(status1,cmac1) ∈ client1 ∧ (status1,cmac1,isp,ch1) ∈ opens)

The formula V specifies constraints that: for all *isp11* in *ISP*, *ch* in *Channel*, and *cmac* in *MAC* such that, *status1* and *cmac* belong to *client1* and *status1*, *cmac*, and *isp11* belong to *connection*. It follows that, *status1* and *cmac* belong to *client1* and *status1*, *cmac*, *isp11*, and *ch* belong to *opens*, then *isp* equals *isp11* and *ch1* equals *ch*. For all *cmac1* in *MAC* such that, *status1* and *cmac1* belong to *client1* and *status1*, *cmac1*, and *isp* belong to *connection*. It follows that, *status1* and *cmac1* belong to *client1* and *status1*, *cmac1*, *isp*, and *ch1* belong to *opens*.

Line (11) for the predicate below and in Appendix (D.9) is expressed in formula W below:

status1.server.(status1.opens)= status1.server.(status1.connection)->ch1

(forall ((isp11 ISP)(ch Channel))(=>(forall ((smac MAC))(=>
(and (server status1 smac)(connection status1 smac isp11))
(and (server status1 smac)(opens status1 smac isp11 ch))))

(and(= isp1 isp11)(= ch1 ch))))(forall ((smac1 MAC))(=>
(and (server status1 smac1)(connection status1 smac1 isp))
(and (server status1 smac1)(opens status1 smac1 isp1 ch1))))

Formula W:

(exists ((mac25 MAC)) *(∀ isp11:ISP, ch:Channel ∀ smac:MAC.(status1,smac)*
∈ server ∧(status1,smac,isp11) ∈ connection =>
(status1,smac) ∈ server ∧ (status1,smac,isp11,ch) ∈ opens
=> (isp1 = isp11) ∧ (ch1 = ch) ∧
(∀ smac1:MAC.(status1,smac1) ∈ server ∧
(status1,smac1,isp) ∈ connection =>
(status1,smac1) ∈ server ∧ (status1,smac1,isp1,ch1) ∈ opens)

The formula W specifies constraints that: for all *isp11* in *ISP*, *ch* in *Channel*,and *smac* in *MAC* such that, *status1* and *smac* belong to *server* and *status1*, *smac*, and *isp11* belong to *connection*. It follows that, *status1* and *smac* belong to *server* and *status1*, *smac*, *isp11*, and *ch* belong to *opens*, then *isp1* equals *isp11* and *ch1* equals *ch*. For all *smac1* in *MAC* such that, *status1* and *smac1* belong to *server* and *status1*, *smac1*, and *isp* belong to *connection*. It follows that, *status1* and *smac1* belong to *server* and *status1*, *smac1*, *isp1*, and *ch1* belong to *opens*.

Line (12) for the predicate below and in Appendix (D.9) is expressed in formula x below:

---

status1.client1.(status1.sends)=d->t->ch1

(forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= d d11)(= t t11)(= ch1 ch11))))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 d t ch1)))

Formula x:

(∀ t,t1:Time, d,d1:Data, ch1:Channel
(∀ t11:Time, d11:Data, ch11:Channel
∃ cmac:MAC.(status1,cmac) ∈ client1
∧ (status1,cmac,d11,t11,ch11) ∈ sends =>
(d = d11) ∧ (t = t11) ∧ (ch1 = ch11)))) ∧
∃ cmac1:MAC.(status1,cmac1) ∈ client1
∧ (status1,cmac1,d,t,ch1) ∈ sends)))

The formula x specifies constraints that: for all atoms *t* and *t1* in *Time*; *d* and *d1* in *Data*; and *ch1* in *Channel*, for all atoms *t11* in *Time*; *d11* in *Data*; and *ch11* in *Channel*, if there exists *cmac* in *MAC* such that, *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends*, then *d* equals *d11*, *t* equals *t11*, and *ch1* equals *ch11*. So, if there exists *cmac1* in *MAC*, then *status1* and *cmac1* belong to *client1*; and *status1*, *cmac1*, *d*, *t*, and *ch1* belong to *sends*.

Line (13) for the predicate below and in Appendix (D.9) expressed to formula Y below:

status1.server.(status1.receives)=d1 ->t1 ->ch1

(forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= d1 d11)(= t1 t11) (= ch1 ch11))))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 d1 t1 ch1)))

Formula Y:

(∀ t11:Time, d11:Data, ch11:Channel
∃ smac:MAC.(status1,smac) ∈ server
∧ (status1,smac,d11,t11,ch11) ∈ receives =>
(d1 = d11) ∧ (t1 = t11) ∧ (ch1 = ch11))) ∧
∃ smac1:MAC.(status1,smac1) ∈ server
∧ (status1,smac1,d1,t1,ch1) ∈ receives)))

The formula Y specifies constraints that: for all atoms *t11* in *Time*; *d11* in *Data*; and *ch11* in *Channel*, if there exists *smac* in *MAC* such that, *status1* and *smac* belong to *server*; and *status1*, *smac*, *d11*, *t11*, and *ch11* belong to *receives*, then *d1* equals *d11*, *t1* equals *t11*, and *ch1* equals *ch11.* So, if there exists *smac1* in *MAC*, then *status1* and *smac1* belong to *server*; and *status1*, *smac1*, *d1*, *t1*, and *ch1* belong to *receives*.

Lines (14, 15) for the predicate below and in Appendix (E.9) is expressed in formula Z below:

status1.mitmIntercepts !in status1.visitors or
(((status1.mitmIntercepts in status1.visitors) and
(status1.client) !in status1.mitmIntercepts))

(or(=>(mitmIntercepts status1 mmac)(not(visitors status1 mmac)))
(and(=>(mitmIntercepts status1 mmac)(visitors status1 mmac))
(=>(client1 status1 cmac)(not(mitmIntercepts status1 cmac))))))

Formula Z:

(status1,mmac) $\in$ *mitmIntercepts =>(status1,mmac)* $\notin$ *visitors* $\vee$
((status1,mmac) $\in$ *mitmIntercepts =>(status1,mmac)* $\in$ *visitors*
$\wedge$ *(status1,cmac)* $\in$ *client1 =>(status1,cmac)* $\notin$ *mitmIntercepts)*
=>

The formula Z specifies constraints that: if a MitM *mmac* belongs to *mitmIntercepts* in the *status*, then the MitM *mmac* does not belong to the *visitors* in the *status* (secure scope), or if the MitM *mmac* belongs to *mitmIntercepts* in the *status*, then the MitM *mmac* belongs to the *visitors* in the *status*, and if a client *cmac* belongs to *client1* in the *status*, then the client *cmac* does not belong to *mitmIntercepts* in the *status*.

7.1.1.7 Assertion

Lines (24) in Appendix (D.9) is expressed in formula A1 below:

d = d1
= *d d1*
Formula A1:
d = *d1*

The formula A1 specifies constraints that: sending and receiving data should be equal.

7.1.2 Second Protocol: Transmitting Data Over Multichannel

7.1.2.1 Type and Subtype Declarations

In Appendix (F) we give full details of the Z3 models with annotations to show the equivalent Alloy.

As seen in Appendix F, the hierarchical type system is translated implicitly. However, because the SMT language does not support subtypes, we use membership functions to enforce type hierarchy declarations. Consequently, top-level types are translated to the uninterpreted sorts, while the top-level (super-type) of a type is translated to uninterpreted membership function *isType* to indicate which elements of the super-type belongs to the type. It is not necessary to declare the membership functions of top-level types, but we declared them to determine the semantics of the subtype.

As seen in Appendix (F.1), top-level types are the same as the top-level type in the first protocol. Membership functions in Appendix (F.2) Lines (16, 17) *isFirst-CommunicationAndExchangingIndices*, and *isSecondCommunicationAndExchang-ingLetters* are declared to specify the semantics of subtypes *FirstCommunicatio-nAndExchangingIndices*, and *SecondCommunicationAndExchangingLetters*. I.e all membership functions in Lines (16, 17) are disjoint subsets of the declared type *ConnectionStatus*.

7.1.2.2 Properties Of The Sub-signatures

As seen in Appendix (F.5), in (Lines 1, 2) we adjust the return types of the "oneOf " functions/constants by specifying that each returns a value of type *Con-nectionStatus*. For example: Line (1) calls function *oneOfFirstCommunicatio-nAndExchangingIndices* in Line (1) in Appendix (F.3) which calls function *is-FirstCommunicationAndExchangingIndices* in Line (16) in Appendix (F.2) which returns one *ConnectionStatus* type *isFirstCommunicationAndExchangingIndices* in Line (16) Appendix (F.2) which is already specified as return type in Line (1) Appendix (F.5).

Sub-signatures (Lines 1, 2) Appendix (F.5) declare functions of the property *some* for each of the sub signatures *FirstCommunicationAndExchangingIndices*, and *SecondCommunicationAndExchangingLetters* of the super signature *Connec-tionStatus* because *ConnectionStatus* has at least one element of the sub signa-tures. The functions restrict the super signature to have at least *one* element in each *ConnectionStatus*. In (Lines 1, 2) Appendix (F.6) we need to assert the *lone* property of the previous sub signatures because *ConnectionStatus* has at most one element of the sub signatures. The assertion (Lines 1, 2) Appendix (F.6) are expressed in the formula B1 below:

Formula B1:

$\forall$ *f1, f2: ConnectionStatus.(f1* $\in$ *isFirstCommunicationAndExchangingIndices* $\wedge$
f2 $\in$ *isFirstCommunicationAndExchangingIndices) =>f1 = f2* $\wedge$
$\forall$ *s1, s2: ConnectionStatus.(s1* $\in$ *isSecondCommunicationAndExchangingLetters* $\wedge$
s2 $\in$ *isSecondCommunicationAndExchangingLetters) =>s1 = s2*

The formula B1 specifies constraints that for each *ConnectionStatus*, there is at most only one corresponding *ConnectionStatus*: if there exist two *ConnectionStatus* belongs to *isFirstCommunicationAndExchangingIndices* for example, then these two *ConnectionStatus* should be equals. i.e we restrict the characteristics of multiplicity *lone* which is (at most one) *ConnectionStatus* of type *isFirstCommunicationAndExchangingIndices* for each statues to avoid the inconsistency.

7.1.2.3 Abstraction

As seen in Appendix (F.7), an abstract type is the union of its subtypes. Thus it constrains every element of a type to belong to one of its extending subtypes. Line (1) is expressed in the formula C1 below:

Formula C1:

$\forall$ *co:ConnectionStatus.*$\neg$*(co* $\in$ *isFirstCommunicationAndExchangingIndices* $\wedge$
co $\in$ *isSecondCommunicationAndExchangingLetters)*

The formula C1 specifies constraints that for each *ConnectionStatus*, there is only one corresponding *ConnectionStatus*, and this *ConnectionStatus* is either *isFirstCommunicationAndExchangingIndices* or *isSecondCommunicationAndExchangingLetters* to avoid inconsistency. No two *ConnectionStatus* occur at one time.

7.1.2.4 Extension

As seen in Appendix (F.8), the extends types are mutually disjoint. Line (1) is expressed in the formula D1 below:

Formula D1:

$\forall$ *co:ConnectionStatus.(co* $\in$ *isFirstCommunicationAndExchangingIndices* $\vee$ *co* $\in$ *isSecondCommunicationAndExchangingLetters)*

The formula D1 specifies constraints that for all *ConnectionStatus* , *ConnectionStatus co* does not belong to *isFirstCommunicationAndExchangingIndices* and *isSecondCommunicationAndExchangingLetters*; it only belongs to one *ConnectionStatus.*

7.1.2.5 Facts

Alloy facts are assumed to be true. They represent the multichannel protocol properties. As seen in Appendix (F.10), Line (1) declares quantifiers that restrict the first fact, line (2) declares quantifiers that restrict the second fact, and line (3) declares quantifiers that restrict the third fact.

The first fact as seen below is expressed in formula E1 below:

First fact:

```
(forall ((status1 CommunicationStatus)(status2 CommunicationStatus)
(implies(forall ((cmac1 MAC)(cmac2 MAC))(and(and (=>
(client1 status1 cmac1)(visitors status1 cmac1))(=>(client2 status1 cmac2)
(not(visitors status1 cmac2)))(=>(client1 status2 cmac1)(not(visitors status2 cmac1)))
(=>(client2 status2 cmac2)(visitors status2 cmac2)))
(or (and (=>(client1 status1 cmac1)(interseptMacs status1 cmac1))
(=>(client2 status2 cmac2)(not(interseptMacs status2 cmac2))))
(and (=>(client1 status1 cmac1)(not(interseptMacs status1 cmac1)))
(=>(client2 status2 cmac2)(interseptMacs status2 cmac2)))
(and(=>(client1 status1 cmac1)(not(interseptMacs status1 cmac1)))
(=>(client2 status2 cmac2)(not(interseptMacs status2 cmac2)))))))
(forall ((status1 CommunicationStatus)(status2 CommunicationStatus)(t Time)(t1 Time)
(t2 Time)(t3 Time)(indices Data)(indices1 Data)(letters Data)(letters1 Data))
(implies (and
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((cmac MAC))
(and (client1 status1 cmac)(sends status1 cmac d11 t11 ch11)))
(and(= indices d11)(= t t11)(ch1 status1 ch11)))(exists ((cmac1 MAC))
(and (client1 status1 cmac1)(sends status1 cmac1 indices t ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((smac MAC))
(and (server status1 smac)(receives status1 smac d11 t11 ch11)))
(and(= indices1 d11)(= t1 t11)(ch1 status1 ch11)))(exists ((smac1 MAC))
(and (server status1 smac1)(receives status1 smac1 indices1 t1 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((cmac MAC))
(and (client2 status2 cmac)(sends status2 cmac d11 t11 ch11)))
(and(= letters d11)(= t2 t11)(ch2 status2 ch11)))(exists ((cmac2 MAC))
(and (client2 status2 cmac2)(sends status2 cmac2 letters t2 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((smac MAC))
(and (server status2 smac)(receives status2 smac d11 t11 ch11)))
(and(= letters1 d11)(= t3 t11)(ch2 status2 ch11)))(exists ((smac2 MAC))
(and (server status2 smac2)(receives status2 smac2 letters1 t3 ch11))))))
(forall ((cmac1 MAC)(cmac2 MAC))
(or (and(and(= t2 t3)(not(= t t1)))(and (=>(client1 status1 cmac1)
(interseptMacs status1 cmac1))(=>(client2 status2 cmac2)
(not(interseptMacs status2 cmac2)))))(and(and(not(= t2 t3))(= t t1))
(and (=>(client1 status1 cmac1)(not(interseptMacs status1 cmac1)))
(=>(client2 status2 cmac2)(interseptMacs status2 cmac2))))
(and(and(= t t1 )(= t2 t3 ))(and(=>(client1 status1 cmac1)
(not(interseptMacs status1 cmac1)))(=>(client2 status2 cmac2))))
(not(interseptMacs status2 cmac2)))))))))))
```

Formula E1:

$\forall$ *status1,status2:CommunicationStatus, cmac1, cmac2:MAC.((status1,cmac1)* $\in$ *client1 =>(status1,cmac1)* $\in$ *visitors* $\wedge$ *(status1,cmac2)* $\notin$ *client2 =>(status1,cmac2)* $\notin$ *visitors* $\wedge$ *(status2,cmac1)* $\notin$ *client1 =>(status2,cmac1)* $\notin$ *visitors* $\wedge$ *(status2,cmac2)* $\in$ *client2 =>(status2,cmac2)* $\in$ *visitors* $\wedge$
((status1,cmac1) $\in$ *client1 =>(status1,cmac1)* $\in$ *interseptMacs* $\wedge$
(status2,cmac2) $\in$ *client2 =>(status2,cmac2)* $\notin$ *interseptMacs* $\vee$
(status1,cmac1) $\in$ *client1 =>(status1,cmac1)* $\notin$ *interseptMacs* $\wedge$
(status2,cmac2) $\in$ *client2 =>(status2,cmac2)* $\in$ *interseptMacs* $\vee$
(status1,cmac1) $\in$ *client1 =>(status1,cmac1)* $\notin$ *interseptMacs* $\wedge$
(status2,cmac2) $\in$ *client2 =>(status2,cmac2)* $\notin$ *interseptMacs) =>*
$\forall$*status1,status2:CommunicationStatus, t,t1,t2,t3:Time,indices, indices1,letters,letters1:Data*

(∀ d11:Data, t11:Time,ch11:Channel (∃ cmac:MCAC.(ststus1, cmac) ∈ client1 ∧
(ststus1,cmac,d11,t11,ch11) ∈ sends =>(indices = d11) ∧ (t = t11) ∧
(ststus1,ch11) ∈ ch1) =>∃ cmac1:MCAC.(ststus1, cmac1) ∈ client1 ∧
(ststus1,cmac1,indices,t,ch11) ∈ sends ∧
∀ d11:Data, t11:Time,ch11:Channel (∃ smac:MCAC.(ststus1, smac) ∈ server ∧
(ststus1,smac,d11,t11,ch11) ∈ receives =>(indices1 = d11) ∧ (t1 = t11) ∧
(ststus1,ch11) ∈ ch1) =>∃ smac1:MCAC.(ststus1, smac1) ∈ server ∧
(ststus1,smac1,indices1,t1,ch11) ∈ receives ∧
(∀ d11:Data, t11:Time,ch11:Channel (∃ cmac:MCAC.(ststus2, cmac) ∈ client2 ∧
(ststus2,cmac,d11,t11,ch11) ∈ sends =>(letters = d11) ∧ (t2 = t11) ∧
(ststus2,ch11) ∈ ch2) =>∃ cmac2:MCAC.(ststus2, cmac2) ∈ client2 ∧
(ststus2,cmac2,letters,t2,ch11) ∈ sends ∧
∀ d11:Data, t11:Time,ch11:Channel (∃ smac:MCAC.(ststus2, smac) ∈ server ∧
(ststus2,smac,d11,t11,ch11) ∈ receives =>(letters1 = d11) ∧ (t3 = t11) ∧
(ststus2,ch11) ∈ ch2) =>∃ smac2:MCAC.(ststus2, smac2) ∈ server ∧
(ststus2,smac2,letters1,t2,ch11) ∈ receives) =>
(∀ cmac1:MCAC,cmac2:MCAC.(t2 = t3) ∧ (t ≠ t1) ∧
((status1,cmac1) ∈ client1 =>(status1,cmac1) ∈ interseptMacs ∧
(status2,cmac2) ∈ client2 =>(status2,cmac2) ∉ interseptMacs ∧
(t2 ≠ t3) ∧ (t = t1) ∧
(status1,cmac1) ∈ client1 =>(status1,cmac1) ∉ interseptMacs ∧
(status2,cmac2) ∈ client2 =>(status2,cmac2) ∈ interseptMacs ∧
(t2 = t3) ∧ (t = t1) ∧
(status1,cmac1) ∈ client1 =>(status1,cmac1) ∉ interseptMacs ∧
(status2,cmac2) ∈ client2 =>(status2,cmac2) ∉ interseptMacs))))))))))

The formula E1 specifies constraints that: for all atoms *status1* and *status2* in *CommunicationStatus*, *cmac1*, *cmac2* in *MAC*, such that if the first client belongs to *client1* in *status1*, the first client belongs to *visitors* in *status1*, and if the second client does not belong to *client2* in *status1*, the second client does not belong to *visitors* in *status1*, and if the first client does not belong to *client1* in *status2*, the first client does not belong to *visitors* in *status2*, and if the second client belongs to *client2* in *status2*, the second client belongs to *visitors* in *status2*.

And, if the first client belongs to *client1* in *status1*, the first client belongs to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, the second client does not belong to *interseptMacs* in *status2*. Or, if the first client belongs to *client1* in *status1*, the first client does not belong to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, the second client belongs to *interseptMacs* in *status2*. Or, if the first client belongs to *client1* in *status1*, the first client does not belong to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, the second client does not belong to *interseptMacs* in *status2*.

It follows that for all *status1* and CommunicationStatus in *status1*; *t*, *t1*, *t2*, and *t3* in *Time*; *indices*, *indices1*, *letters*, and *letters1* in *Data*, for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *cmac* in *MAC*, such that *status1* and *cmac* belong to *client1* and *status1*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *indices* equals *d11*, *t* equals *t11*, and *ststus1*, *ch11* belong to *ch1*, then if there exists a *cmac1* in *MAC*, such that *status1* and *cmac1* belong to *client1* and *status1*, *cmac1*, *indices*, *t*, and *ch11* belong to *sends*.

Then for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *smac* in *MAC*, such that *status1* and *smac* belong to *server* and *status1*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *indices1* equals *d11*, *t1* equals *t11* ,and *ststus1*, *ch11* belong to *ch1*, then if there exists a *smac1* in *MAC*, such that *status1* and *smac1* belong to *server* and *status1*, *smac1*, *indices1*, *t1*, and *ch11* belong to *receives*.

Then for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *cmac* in *MAC*, such that *status2* and *cmac* belong to *client2* and *status2*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *letters* equals *d11*, *t2* equals *t11*, and *ststus2*, *ch11* belong to *ch2*, then if there exists a *cmac2* in *MAC*, such that *status2* and *cmac2* belong to *client2* and *status2*, *cmac2*, *letters*, *t2*, and *ch11* belong to *sends*.

Then for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *smac* in *MAC*, such that *status2* and *smac* belong to *server* and *status2*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *letters1* equals *d11*, *t3* equals *t11* ,and *ststus2*, *ch11* belong to *ch2*, then if there exists a *smac2* in *MAC*, such that *status2* and *smac2* belong to *server* and *status2*, *smac2*, *letters1*, *t3*, and *ch11* belong to *receives*.

Then for all *cmac1*, and *cmac2* in *MAC*, such that if times for sending and receiving indices are not equal and times for sending and receiving letters are equal and if the first client belongs to *client1* in *status1*, then the first client belongs to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, then the second client does not belong to *interseptMacs* in *status2*. And, times for sending and receiving indices are equal and times for sending and receiving letters are not equal and if the first client belongs to *client1* in *status1*, then the first client does not belong to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, then the second client belongs to *interseptMacs* in *status2*. And, times for sending and receiving indices are equal and times for sending and receiving letters are equal and if the first client belongs to *client1* in *stastus1*, then the first client does not belong to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, then the second client does not belong to *interseptMacs* in *status2*.

The second fact as seen below is expressed in formula F1 below:

Second fact:

(forall ((status1 CommunicationStatus)(status2 CommunicationStatus)(t Time)(t1 Time)
(t2 Time)(t3 Time)(indices Data)(indices1 Data)(letters Data)(letters1 Data))
(implies (and
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((cmac MAC))
(and (client1 status1 cmac)(sends status1 cmac d11 t11 ch11)))
(and(= indices d11)(= t t11)(ch1 status1 ch11)))(exists ((cmac1 MAC))
(and (client1 status1 cmac1)(sends status1 cmac1 indices t ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((smac MAC))
(and (server status1 smac)(receives status1 smac d11 t11 ch11)))
(and(= indices1 d11)(= t1 t11)(ch1 status1 ch11)))(exists ((smac1 MAC))
(and (server status1 smac1)(receives status1 smac1 indices1 t1 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((cmac MAC))
(and (client2 status2 cmac)(sends status2 cmac d11 t11 ch11)))
(and(= letters d11)(= t2 t11)(ch2 status2 ch11)))(exists ((cmac2 MAC))
(and (client2 status2 cmac2)(sends status2 cmac2 letters t2 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>(exists ((smac MAC))
(and (server status2 smac)(receives status2 smac d11 t11 ch11)))
(and(= letters1 d11)(= t3 t11)(ch2 status2 ch11)))(exists ((smac2 MAC))
(and (server status2 smac2)(receives status2 smac2 letters1 t3 ch11)))))
(or(and(= t t1)(not(= t2 t3)))(and(= t2 t3)(not(= t t1)))(and(= t2 t3)(= t t1))))
(and(= indices indices1)(= letters letters1))))

Formula F1:

$\forall$*status1,status2:CommunicationStatus, t,t1,t2,t3:Time,indices,*
indices1,letters,letters1:Data
($\forall$ *d11:Data, t11:Time,ch11:Channel (*$\exists$ *cmac:MCAC.(ststus1, cmac)* $\in$ *client1*
$\wedge$ *(ststus1,cmac,d11,t11,ch11)* $\in$ *sends =>(indices = d11)* $\wedge$ *(t = t11)*
$\wedge$ *(ststus1,ch11)* $\in$ *ch1) =>*$\exists$ *cmac1:MCAC.(ststus1, cmac1)* $\in$ *client1*
$\wedge$ *(ststus1,cmac1,indices,t,ch11)* $\in$ *sends* $\wedge$
$\forall$ *d11:Data, t11:Time,ch11:Channel (*$\exists$ *smac:MCAC.(ststus1, smac)* $\in$ *server*
$\wedge$ *(ststus1,smac,d11,t11,ch11)* $\in$ *receives =>(indices1 = d11)* $\wedge$ *(t1 = t11)*
$\wedge$ *(ststus1,ch11)* $\in$ *ch1) =>*$\exists$ *smac1:MCAC.(ststus1, smac1)* $\in$ *server*

∧ (ststus1,smac1,indices1,t1,ch11) ∈ receives ∧
(∀ d11:Data, t11:Time,ch11:Channel (∃ cmac:MCAC.(ststus2, cmac) ∈ client2
∧ (ststus2,cmac,d11,t11,ch11) ∈ sends =>(letters = d11) ∧ (t2 = t11)
∧ (ststus2,ch11) ∈ ch2) =>∃ cmac2:MCAC.(ststus2, cmac2) ∈ client2
∧ (ststus2,cmac2,letters,t2,ch11) ∈ sends ∧
∀ d11:Data, t11:Time,ch11:Channel (∃ smac:MCAC.(ststus2, smac) ∈ server
∧ (ststus2,smac,d11,t11,ch11) ∈ receives =>(letters1 = d11) ∧ (t3 = t11)
∧ (ststus2,ch11) ∈ ch2) =>∃ smac2:MCAC.(ststus2, smac2) ∈ server
∧ (ststus2,smac2,letters1,t2,ch11) ∈ receives) ∧
(t ≠ t1) ∨ (t2 ≠ t3) ∧ (t = t1) ∨ (t2 = t3) ∧ (t = t1) =>
(indices = indices1) ∧ (letters = letters1))))

The formula F1 specifies constraints that: for all *status1* and *Communication-Status* in *status1*; *t*, *t1*, *t2*, and *t3* in *Time*; *indices*, *indices1*, *letters*, an *letters1* in *Data*, for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *cmac* in *MAC*, such that *status1* and *cmac* belong to *client1* and *status1*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *indices* equals *d11*, *t* equals *t11* ,and *ststus1*, *ch11* belong to *ch1*, then if there exists a *cmac1* in *MAC*, such that *status1* and *cmac1* belong to *client1* and *status1*, *cmac1*, *indices*, *t*, and *ch11* belong to *sends.*

Then for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *smac* in *MAC*, such that *status1* and *smac* belong to *server* and *status1*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *indices1* equals *d11*, *t1* equals *t11* ,and *ststus1*, *ch11* belong to *ch1*, then if there exists a *smac1* in *MAC*, such that *status1* and *smac1* belong to *server* and *status1*, *smac1*, *indices1*, *t1*, and *ch11* belong to *receives.*

Then for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *cmac* in *MAC*, such that *status2* and *cmac* belong to *client2* and *status2*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *letters* equals *d11*, *t2* equals *t11* ,and *ststus2*, *ch11* belong to *ch2*, then if there exists a *cmac2* in *MAC*, such that *status2* and *cmac2* belong to *client2* and *status2*, *cmac2*, *letters*, *t2*, and *ch11* belong to *sends.*

Then for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists a *smac* in *MAC*, such that *status2* and *smac* belong to *server* and *status2*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *letters1* equal *d11*, *t3* equal *t11* ,and *ststus2*, *ch11* belong to *ch2*, then if there exists a *smac2* in *MAC*, such that *status2* and *smac2* belong to *server* and *status2*, *smac2*, *letters1*, *t3*, and *ch11* belong to *receives*, and if times for sending and receiving indices are not equal, and times for sending and receiving letters are equal, or times for sending and receiving indices are equal, and times for sending and receiving letters are not equal, or times for sending and receiving indices are equal, and times for sending and receiving letters are equal, then sent and received indices are equal, and sent and received letters are equal.

The third fact as seen below are expressed in formula G1 below:

Third fact:

(forall ((s CommunicationStatus)(s1 CommunicationStatus)(mac1 MAC)
(mac2 MAC)(mac3 MAC)(mac4 MAC))
(and (=>(and(client1 s mac1)(server s mac2))(not(= mac1 mac2)))
(=>(and(client2 s mac1)(server s mac2))(not(= mac1 mac2)))
(=>(and(client1 s mac1)(client2 s1 mac2))(not(= mac1 mac2)))
(=>(and(client1 s mac1)(client1 s1 mac2))(= mac1 mac2))
(=>(and(server s mac3)(server s1 mac4))(= mac3 mac4))))

Formula G1:

$\forall$ *s,s1:CommunicationStatus,mac1, mac2, mac3, mac4.((s,mac1)*
$\in$ *client1* $\wedge$ *(s,mac2)* $\in$ *server =>mac1* $\neq$ *mac2* $\wedge$
((s,mac1) $\in$ *client2* $\wedge$ *(s,mac2)* $\in$ *server =>mac1* $\neq$ *mac2* $\wedge$
((s,mac1) $\in$ *client1* $\wedge$ *(s1,mac2)* $\in$ *client2 =>mac1* $\neq$ *mac2* $\wedge$
((s,mac1) $\in$ *client1* $\wedge$ *(s1,mac2)* $\in$ *client1 =>mac1 = mac2* $\wedge$
((s,mac3) $\in$ *server* $\wedge$ *(s1,mac4)* $\in$ *server =>mac3 = mac4*

The formula G1 specifies constraints that: for all atoms *s* and *s1* in *CommunicationStatus*, and *mac1*; *mac2*; *mac3*; and *mac4* in *MAC* such that *mac1* and *s* belong to *client1* and *s*, *mac2* belong to *server* then, *mac1* does not equal *mac2*; and *s*, *mac1* belong to *client2*, and *s*, *mac2* belong to the *server* then, *mac1* does not equal *mac2*; and *s*, *mac1* belong to *client1*, and *s1*, *mac2* belong to the *client2* then, *mac1* does not equal *mac2*; and *s*, *mac1* belong to *client1*, and *s1*, *mac2* belong to the *client1* then, *mac1* equals *mac2*; and *s*, *mac3* belong to *server*, and *s1*, *mac4* belong to the *server* then, *mac3* equals *mac4*.

7.1.2.6 Relation Declarations

Relations are translated to Boolean-valued, uninterpreted, membership functions. As seen in Appendix (F.2), these functions are declared over top-level types because only top-level types are declared as sorts.

Lines (1-15) Appendix (F.2) declare functions of relations. These relations are the same as the relations in the first protocol. However, more relations are required for developing from the first protocol to the second protocol. These relations are: *client1* and *client2*, which are declared as Boolean-valued function over *CommunicationStatus* and *MAC* (Lines 3,4); *ispA* and*ispB*, which are declared as Boolean-valued function over *CommunicationStatus* and *ISP* (Lines 6-8); *channel*, *ch1*, and *ch2* which are declared as boolean-valued functions over *ConnectionStatus* and *Channel* (Lines 11-13).

Lines (1-12) Appendix (F.9) declare constraints guaranteeing that each relation is defined for its specific types considering the multiplicity keywords constraints.

The relations *visitors:set MAC*, *serviceProvider: set ISP*, and *channel: set Channel* are not required to be translated to formulas to show constraints as the *set* keyword constrains and allows any number of elements. Thus, its defined

Boolean-valued function (Lines 1, 2, and 11) Appendix (F.2) are equivalent to their meanings.

Line (1) for the relation below and as seen in Appendix (F.9) is expressed in formula H1 below:

client1: one MAC

*(forall ((cs CommunicationStatus))(and(exists ((mac1 MAC))(client1 cs mac1))
(forall ((mac3 MAC)(mac2 MAC))(=>(and(client1 cs mac2)(client1 cs mac3))
(= mac2 mac3)))))*

Formula H1:

∀ *status:CommunicationStatus* ∃ *mac1:MAC.(status,mac1)* ∈
client1 ∧ ∀ *mac3,mac2:MAC.(status,mac3)* ∈ *client1*
∧ *(status,mac2)* ∈ *client1 =>mac2 = mac3*

The formula H1 specifies constraints that: for all atoms *status* in *CommunicationStatus*, if there exists *mac1* in *MAC*, such that the *status* and *mac1* belong to the first client *client1*, and for all *mac3* and *mac2*:MAC such that *mac3* and *status* belong to the first client (client1) and *status* and *mac2* belong to the first client (client1) then *mac2* equals *mac3*. To restrict the multiplicity to be one, we suppose that if there are two macs belonging to the first client *client1*, these two macs are equal.

Line (2) for the relation below and as seen in Appendix (F.9) is expressed in formula I1 below:

client2: one MAC

*(forall ((cs CommunicationStatus))(and(exists ((mac1 MAC))(client2 cs mac1))
(forall ((mac3 MAC)(mac2 MAC))(=>(and(client2 cs mac2)(client2 cs mac3))
(= mac2 mac3)))))*

Formula I1:

$\forall$ *status:CommunicationStatus* $\exists$ *mac1:MAC.(status,mac1)* $\in$
client2 $\wedge$ $\forall$ *mac3,mac2:MAC.(status,mac3)* $\in$ *client2*
$\wedge$ *(status,mac2)* $\in$ *client2 =>mac2 = mac3*

The formula I1 specifies constraints that: for all atoms *status* in *CommunicationStatus*, if there exists *mac1* in *MAC*, such that the *status* and *mac1* belong to the second client *client2*, and for all *mac3* and *mac2*:MAC such that *mac3* and *status* belong to the second client (client2) and *status* and *mac2* belong to the second client (client2) then *mac2* equals *mac3*. To restrict the multiplicity to be one, we suppose that if there are two macs belonging to the second client *client1*, these two macs are equal.

Line (5) for the relation below and as seen in Appendix (F.9) is expressed in formula J1 below:

ispA:one ISP

*(forall ((cs CommunicationStatus))(and (exists ((isp1 ISP))(ispA cs isp1))
(forall ((isp2 ISP)(isp3 ISP))(=>(and (ispA cs isp2)(ispA cs isp3))
(= isp2 isp3)))))*

Formula J1:

$\forall$ *status:CommunicationStatus* $\exists$ *isp1:ISP.(status,isp1)* $\in$ *ispA* $\wedge$
$\forall$ *isp3,isp2:ISP.(status,isp3)* $\in$ *isbA* $\wedge$ *(status,isb2)* $\in$ *isbA =>isp2 = isp3*

The formula J1 specifics constraints that: for all atoms *status* in *CommunicationStatus*, if there exists *isp1* in *ISP*, such that the *status* and *isp1* belong to *isbA* and for all *isp3* and *isp2* in *ISP*, such that the *status* and *isp3* belong to *ispA* and *status* and *isb2* belong to *ispA* then *isp2* equals *isp3*. To restrict the multiplicity to be one, we suppose that if there are two *isps* belonging to the *ISP*, these two *isps* are equal.

Line (6) for the relation below and as seen in Appendix (F.9) is expressed in formula K1 below:

ispB:one ISP

(forall ((cs CommunicationStatus))(and (exists ((isp1 ISP))(ispB cs isp1))
(forall ((isp2 ISP)(isp3 ISP))(=>(and (ispB cs isp2)(ispB cs isp3))
(= isp2 isp3)))))

Formula K1:

∀ *status:CommunicationStatus* ∃ *isp1:ISP.(status,isp1)* ∈
isbB ∧ ∀ *isp3,isp2:ISP.(status,isp3)* ∈ *isbB*
∧ *(status,isb2)* ∈ *isbB =>isp2 = isp3*

The formula K1 specifies constraints that: for all atoms *status* in *CommunicationStatus*, if there exists *isp1* in *ISP*, such that the *status* and *isp1* belong to *ispB* and for all *isp3* and *isp2* in *ISP*, such that the *status* and *isp3* belong to *ispB* and *status* and *isb2* belong to *ispB* then *isp2* equals *isp3*. To restrict the multiplicity to be one, we suppose that if there are two *isps* belonging to the *ISP*, these two isps are equal.

Line (9) for the relation below and as seen in Appendix (F.9) is expressed in formula L1 below:

ch1: one Channel

(forall ((cs CommunicationStatus))(and(exists ((ch11 Channel))(ch1 cs ch11))
(forall ((ch22 Channel)(ch33 Channel))(=>(and (ch1 cs ch22)(ch1 cs ch33))
(= ch22 ch33)))))

Formula L1:

$\forall$ *status:CommunicationStatus* $\exists$ *ch11:Channel.(status,ch11)* $\in$ *ch1*
$\wedge$ $\forall$ *ch22,ch33:Channel.(status,ch22)* $\in$ *ch1* $\wedge$
(status,ch33) $\in$ *ch1 =>(ch33 = ch22)*

The formula L1 specifies constraints that: for all atoms *status* in *CommunicationStatus*, if there exists *ch11* in *Channel*, such that the *status* and ch11 belongs to *ch1*; and for all *ch22* and *ch33* in *Channel*, such that, the *status* and *ch22* belong to *ch1* and *ch33* and *status* belong to *ch1* then *ch33* equals *ch22.* To restrict the multiplicity to be one, we suppose that if there are two channels belonging to the *ch1*, two channels are equal.

Line (10) for the relation below and as seen in Appendix (F.9) is expressed in formula M1 below:

ch2: one Channel

(forall ((cs CommunicationStatus))(and(exists ((ch11 Channel))(ch2 cs ch11))
(forall ((ch22 Channel)(ch33 Channel))(=>(and (ch2 cs ch22)(ch2 cs ch33))
(= ch22 ch33)))))

Formula M1:

$\forall$ *status:CommunicationStatus* $\exists$ *ch11:Channel.(status,ch11)* $\in$ *ch2*
$\wedge$ $\forall$ *ch22,ch33:Channel.(status,ch22)* $\in$ *ch2* $\wedge$
(status,ch33) $\in$ *ch2 =>(ch33 = ch22)*

The formula M1 specifies constraints that: for all atoms *status* in *CommunicationStatus*, if there exists *ch11* in *Channel*, such that the *status* and *ch11* belongs to *ch2*; and for all *ch22* and *ch33* in *Channel*, such that, the *status* and *ch22* belong to *ch2* and *ch33* and *status* belong to *ch2* then *ch33* equals *ch22*. To restrict the multiplicity to be one, we suppose that if there are two channels belonging to the *ch2*, two channels are equal.

7.1.2.7 Predicates

A predicate is a logical formula with declaring parameters used to specified operations. As seen in Appendix (F.11), the translation focuses on "inlining " of the predicate *MultiChannel*. As the multichannel protocol passes through two statuses, we illustrate each status individually:

Line (1) for the predicate below in Appendix (F.11) is expressed in formula N1 below:

pred MultiChannel [t,t',t",t"':Time,indices,indices', letters,letters':Data,isp:ISP,status1,status2:Communication_Status]

(forall ((status1 CommunicationStatus)(isp ISP)(status2 CommunicationStatus) (mac MAC) (t Time)(t1 Time)(t2 Time)(t3 Time)(indices Data)(indices1 Data) (letters Data)(letters1 Data))

Formula N1:

∀ *status1,status2:CommunicationStatus , indices,indices1,letters,letters1:Data, t,t1,t2,t3:Time, isp:ISP, mac:MAC*

The formula N1 declared all variables to be used in the predicate and assertion: *status1* and *status2* for the first and second status; *indices* and *indices1* for data have been sent and received in the first status; *letters* and *letters1* for data have been sent and received in the second status; *t* and *t1* times are for sending and receiving indices in the first status; *t2* and *t3* times are for sending and receiving letters in the second status; *isp* for the internet server provider that the server connects with; and *mac* for the set of macs that belong to the visitors.

These variables will be used directly in the predicates and assertion below without the need for redeclaration.

Line (2) for the predicate below in Appendix (F.11) is expressed in formula O1 below:

status1.client1.(status1.status) = First_Communication_And_Exchanging_Indices and

(=>(forall ((cs1 ConnectionStatus)(cmac MAC))
(=>(and (client1 status1 cmac)(status status1 cmac cs1))
(isFirst_Communication_And_Exchanging_Indices cs1))
(forall ((cs2 ConnectionStatus))(cmac MAC))
(=>(isFirst_Communication_And_Exchanging_Indices cs2)
(and (client1 status1 cmac)(status status1 cmac cs2)))

Formula O1:

($\forall$ *cs1:ConnectionStatus,cmac:MAC.(status1,cmac)* $\in$ *client1*
$\wedge$ *(status1,cmac,cs1)*$\in$ *status =>(cs1)* $\in$
isFirstCommunicationAndExchangingIndices) =>
($\forall$ *cs2:ConnectionStatus,cmac:MAC.(cs2)* $\in$
isFirstCommunicationAndExchangingIndices =>(status1,cmac) $\in$ *client1*
$\wedge$ *(status1,cmac,cs2)*$\in$ *status)*

The formula O1 specifies constraints that: for all atoms *cs1* in *ConnectionStatus*, and *cmac* in *MAC*, such that *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, and *cs1* belong to *status* then *cs1* belongs to *isFirstCommunicationAndExchangingIndices*. It follows that for all atoms *cs2* in *ConnectionStatus*, and *cmac* in *MAC*, such that *cs1* belongs to *isFirstCommunicationAndExchangingIndices* then *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, and *cs1* belong to *status*.

Line (3) for the predicate below in Appendix (F.11) is expressed in formula P1 below:

status1.server.(status1.status) =
First_Communication_And_Exchanging_Indices and

(=>(forall ((cs1 ConnectionStatus)(smac MAC))
(=>(and (server status1 smac)(status status1 smac cs1))
(isFirst_Communication_And_Exchanging_Indices cs1))
(forall ((cs2 ConnectionStatus))(exists ((smac MAC))
(=>(isFirst_Communication_And_Exchanging_Indices cs2)
(and (server status1 smac)(status status1 smac cs2)))

Formula P1:

($\forall$ *cs1:ConnectionStatus,smac:MAC.(status1,smac)* $\in$ *server*
$\wedge$ *(status1,smac,cs1)*$\in$ *status =>(cs1)* $\in$
isFirstCommunicationAndExchangingIndices) =>
($\forall$ *cs2:ConnectionStatus,smac:MAC.(cs2)* $\in$
isFirstCommunicationAndExchangingIndices =>(status1,smac) $\in$ *server*
$\wedge$ *(status1,smac,cs2)*$\in$ *status)*

The formula P1 specifies constraints that: for all atoms *cs1* in *ConnectionStatus*, and *smac* in *MAC*, such that *status1* and *smac* belong to *server*; and *status1*, *smac*, and *cs1* belong to *status* then *cs1* belongs to *isFirstCommunicationAndExchangingIndices*. It follows that for all atoms *cs2* in *ConnectionStatus*, and *smac* in *MAC*, such that *cs1* belongs to *isFirstCommunicationAndExchangingIndices* then *status1* and *smac* belong to *server*; and *status1*, *smac*, and *cs1* belong to *status*.

Line (4) for the predicate below in Appendix (F.11) is expressed in formula Q1 below:

status1.server in status1.visitors

(forall ((smac MAC))(=>(server status1 smac)(visitors status1 smac)))

Formula Q1:

($\forall$ smac:MAC.(status1,smac) $\in$ server =>(status1,smac) $\in$ visitors

The formula Q1 specifies constraints that: for all *smac* in *MAC*, such that the *status1* and *smac* belong to server then *status1* and *smac* belong to *visitors*.

Line (5) for the predicate below in Appendix (F.11) is expressed in formula R1 below:

status1.client1.(status1.connection) =status1.ispA and

(forall ((isp11 ISP))(=>(=>(exists ((cmac MAC))(and (client1 status1 cmac) (connection status1 cmac isp11)))(ispA status1 isp11))(exists ((cmac1 MAC))(and (client1 status1 cmac1)(connection status1 cmac1 isp11)))))

Formula R1:

($\forall$ isp11:ISP ($\exists$ cmac:MAC.(status1,cmac) $\in$ client1 $\wedge$
(status1,cmac,isp11) $\in$ connection =>(status1,isp11) $\in$ ispA)) =>
($\exists$ cmac1:MAC.(status1,cmac1) $\in$ client1 $\wedge$ (status1,cmac1,isp11) $\in$ connection

The formula R1 specifies constraints that: for all *isp11* in ISP such that the *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, and *isp11* belong to *connection* then *status1* and *isp11* belong to *ispA*. It follows that there exists *cmac1* in *MAC* such that the *status1* and *cmac1* belong to *client1*; and *status1*, *cmac1*, and *isp11* belong to *connection*.

Line (6) for the predicate below in Appendix (F.11) is expressed in formula S1 below:

status1.ispA in status1.serviceProvider and

(forall ((ispa ISP))(=>(ispA status1 ispa)(serviceProvider status1 ispa)))

Formula S1:

(∀ ispa:ISP.(status1,ispa) ∈ ispA =>
(status1,ispa) ∈ serviceProvider

The formula S1 specifies constraints that: for all *ispa* in *ISP*, such that the *status1* and *ispa* belong to *ispA* then *status1* and *ispa* belong to *serviceProvider*.

Line (7) for the predicate below in Appendix (F.11) is expressed in formula T1 below:

status1.ispB i̸n status1.serviceProvider and

(forall ((ispb ISP))(=>(not(ispB status1 ispb))(serviceProvider status1 ispb)))

Formula T1:

(∀ ispb:ISP.(status1,ispb) ∉ ispB =>(status1,ispb) ∈ serviceProvider

The formula T1 specifies constraints that: for all *ispb* in *ISP*, such that the *status1* and *ispb* do not belong to *ispB* then *status1* and *ispb* belong to *serviceProvider*.

Line (8) for the predicate below in Appendix (F.11) is expressed in formula U1 below:

status1.server.(status1.connection)= isp and

*(forall ((isp11 ISP))(=>(=>(exists ((smac MAC))(and (server status1 smac)
(connection status1 smac isp11)))(= isp isp11))
(exists ((smac1 MAC))(and (server status1 smac1)
(connection status1 smac1 isp)))))*

Formula U1:

*(∀ isp11:ISP ∃ smac:MAC.(status1,smac) ∈ server ∧
(status1,smac,isp11) ∈ connection =>(= isp isp11))
=>(∃ smac1:MAC.(status1,smac1) ∈ server ∧
(status1,smac1,isp) ∈ connection*

The formula U1 specifies constraints that:for all *isp11* in ISP, if there exist *smac* in *MAC* such that the *status1* and *smac* belong to *server*; and *status1*, *smac*, and *isp11* belong to *connection* then *isp* equals *isp11*. It follows that there exists *smac1* in *MAC* such that the *status1* and *smac1* belong to *server*; and *status1*, *smac1*, and *isp* belong to *connection*

Line (9) for the predicate below in Appendix (F.11) is expressed in formula V1 below:

status1.ch1 in status1.channel and

(forall ((ch Channel))(=>(ch1 status1 ch)(channel status1 ch)))

Formula V1:

(∀ ch:Channel.(status1,ch) ∈ ch1 =>(status1,ch) ∈ channel

The formula V1 specifies constraints that: for all *ch* in *Channel*, such that the *status1* and *ch* belong to *ch1* then *status1* and *ch* belong to *Channel*.

Line (10) for the predicate below in Appendix (F.11) is expressed in formula W1 below:

status1.ch2 $\not\in$ status1.channel and

(forall ((ch Channel))(=>(not (ch2 status1 ch))(channel status1 ch)))

Formula W1:

$(\forall\ ch{:}Channel.(status1,ch) \notin ch2 => (status1,ch) \in channel)$

The formula W1 specifies constraints that: for all *ch* in *Channel*, such that the *status1* and *ch* do not belong to *ch2* then *status1* and *ch* belong to *Channel*.

Line (11) for the predicate below in Appendix (F.11) is expressed in formula X1 below:

no status1.client2.(status1.connection) and

(forall ((cmac MAC))(not(and(client2 status1 cmac)(connection status1 cmac isp))))

Formula X1:

$(\forall\ cmac{:}MAC.\ \neg\ ((status1,cmac) \in client2 \wedge (status1,cmac,isp) \in connection))$

The formula X1 specifies constraints that: for all *cmac* in *MAC*, such that there is not *status1* and *cmac* belong to *client2* and; *status1*,*cmac*, and *isp* belong to *connection*.

Line (12) for the predicate below in Appendix (F.11) is expressed in formula Y1 below:

status1.client1.(status1.sends)= indices ->t ->status1.ch1 and

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= indices d11)(= t t11)(ch1 status1 ch11)))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 indices t ch11)))))

Formula Y1:

$(\forall$ *d11:Data,t11:Time,ch11:Channel* $(\exists$ *cmac:MAC.(status1,cmac)*
$\in$ *client1* $\wedge$ *(status1,cmac,d11,t11,ch11)* $\in$ *sends =>*
(indices = d11) $\wedge$ *(t = t11)* $\wedge$ *(status1,ch11)* $\in$ *ch1)*
$(\exists$ *cmac1:MAC.(status1,cmac1)* $\in$ *client1* $\wedge$
(status1,cmac1,indices,t,ch11) $\in$ *sends*

The formula Y1 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *cmac* in *MAC* such that the *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *indices* equals d11, t equals t11, and *status1*, *ch11* belong to *ch1*. It follows that if there exists *cmac1* in *MAC* such that the *status1* and *cmac1* belong to *client1*; and *status1*, *cmac1*, *indices*, *t*, and *ch11* belong to *sends*.

Line (13) for the predicate below in Appendix (F.11) is expressed in formula Z1 below:

status1.client1.(status1.sends) ≠ letters ->t ->status1.ch1 and

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= letters d11)(= t t11)(ch1 status1 ch11)))

(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(not(sends status1 cmac1 letters t ch11))))))

Formula Z1:

(∀ d11:Data,t11:Time,ch11:Channel (∃ cmac:MAC.(status1,cmac)
∈ client1 ∧ (status1,cmac,d11,t11,ch11) ∈ sends =>
(letters = d11) ∧ (t = t11) ∧ (status1,ch11) ∈ ch1)
(∃ cmac1:MAC.(status1,cmac1) ∈ client1 ∧
(status1,cmac1,letters,t,ch11) ∉ sends

The formula Z1 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *cmac* in *MAC* such that the *status1* and *cmac* belong to *client1*; and *status1*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *letters* equals d11, t equals t11, and *status1*, *ch11* belong to *ch1*. It follows that if there exists *cmac1* in *MAC* such that the *status1* and *cmac1* belong to *client1*; and *status1*, *cmac1*, *letters*, *t*, and *ch11* do not belong to *sends*.

Line (14) for the predicate below in Appendix (F.11) is expressed in formula A2 below:

status1.server.(status1.receives) ≠ letters' ->t' ->status1.ch1 and

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= letters1 d11)(= t1 t11)(ch1 status1 ch11)))
(exists ((smac1 MAC))(and (server status1 smac1)
(not(receives status1 smac1 letters1 t1 ch11))))))

Formula A2:

(∀ d11:Data,t11:Time,ch11:Channel (∃ smac:MAC.(status1,smac)
∈ server ∧ (status1,smac,d11,t11,ch11) ∈ receives =>
(letters1 = d11) ∧ (t1 = t11) ∧ (status1,ch11) ∈ ch1) ∧
(∃ smac1:MAC.(status1,smac1) ∈ server ∧
(status1,smac1,letters1,t1,ch11) ∉ receives)

The formula A2 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *smac* in *MAC* such that the *status1* and *smac* belong to *server*; and *status1*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *letters1* equals *d11*, *t1* equals *t11*, and *status1*, *ch11* belong to *ch1*. It follows that if there exists *smac1* in *MAC* such that the *status1* and *smac1* belong to *server*; and *status1*, *smac1*, *letters1*, *t1*, and *ch11* do not belong to *receives*.

Line (15) for the predicate below in Appendix (F.11) is expressed in formula B2 below:

status1.server.(status1.receives)= indices' ->t' ->status1.ch1 and

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= indices1 d11)(= t1 t11)(ch1 status1 ch11)))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 indices1 t1 ch11)))))

Formula B2:

$(\forall$ *d11:Data,t11:Time,ch11:Channel* $(\exists$ *smac:MAC.(status1,smac)*
$\in$ *server* $\wedge$ *(status1,smac,d11,t11,ch11)* $\in$ *receives =>*
(indices1 = d11) $\wedge$ *(t1 = t11)* $\wedge$ *(status1,ch11)* $\in$ *ch1) t11*
$(\exists$ *smac1:MAC.(status1,smac1)* $\in$ *server* $\wedge$
(status1,smac1,indices1,t1,ch11) $\in$ *receives)*

The formula B2 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *smac* in *MAC* such that the *status1* and *smac* belong to *server*; and *status1*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *indices1* equals *d11*, *t1* equals *t11*, and *status1*, *ch11* belong to *ch1*. It follows that if there exists *smac1* in *MAC* such that the *status1* and *smac1* belong to *server*; and *status1*, *smac1*, *indices1*, *t1*, and *ch11* belong to *receives*.

Line (16) for the predicate below in Appendix (F.11) is expressed in formula C2 below:

no status1.server.(status1.sends) and

(forall ((ch Channel)(smac MAC))
(and(not(and(server status1 smac)(sends status1 smac indices t ch)))
(not(and(server status1 smac)(sends status1 smac indices1 t ch)))
(not(and(server status1 smac)(sends status1 smac letters t ch)))
(not(and(server status1 smac)(sends status1 smac letters1 t ch)))))

Formula C2:

(∀ *ch:Channel ,smac:MAC.* ¬ *((status1,smac)* ∈ *server* ∧
(status1,smac,indices,t,ch) in sends) ∧
¬ *((status1,smac)* ∈ *server* ∧ *(status1,smac,indices1,t,ch) in sends)* ∧
¬ *((status1,smac)* ∈ *server* ∧ *(status1,smac,letters,t,ch) in sends)* ∧
¬ *((status1,smac)* ∈ *server* ∧ *(status1,smac,letters1,t,ch) in sends)* ∧

The formula C2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that not *status1* and *smac* belong to *server*; and *status1*, *smac*, *indices*, *t*, and *ch* belong to *sends*; and not *status1* and *smac* belong to *server*; and *status1*, *smac*, *indices1*, *t*, and *ch* belong to *sends*; and not *status1* and *smac* belong to *server*; and *status1*, *smac*, *letters*, *t*, and *ch* belong to *sends*; and not *status1* and *smac* belong to *server*; and *status1*, *smac*, *letters1*, *t*, and *ch* belong to *sends.*

Line (17) for the predicate below in Appendix (F.11) is expressed in formula D2 below:

no status1.client1.(status1.receives) and

(forall ((ch Channel)(smac MAC))
(and(not(and(client1 status1 smac)(receives status1 smac indices t1 ch)))
(not(and(client1 status1 smac)(receives status1 smac indices1 t1 ch)))

(not(and(client1 status1 smac)(receives status1 smac letters t1 ch)))
(not(and(client1 status1 smac)(receives status1 smac letters1 t1 ch)))))

Formula D2:

(∀ ch:Channel ,smac:MAC. ¬ ((status1,smac) ∈ client1 ∧
(status1,smac,indices,t1,ch) in receives) ∧
¬ ((status1,smac) ∈ client1 ∧ (status1,smac,indices1,t1,ch) in receives) ∧
¬ ((status1,smac) ∈ client1 ∧ (status1,smac,letters,t1,ch) in receives) ∧
¬ ((status1,smac) ∈ client1 ∧ (status1,smac,letters1,t1,ch) in receives) ∧

The formula D2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that *status1* and *smac* do not belong to *client1*; and *status1*, *smac*, *indices*, *t1*, and *ch* belong to *receives*; and *status1* and *smac* do not belong to *client1*; and *status1*, *smac*, *indices1*, *t1*, and *ch* belong to *receives*; and *status1* and *smac* do not belong to *client1*; and *status1*, *smac*, *letters*, *t1*, and *ch* belong to *receives*; and *status1* and *smac* do not belong to *client1*; and *status1*, *smac*, *letters1*, *t1*, and *ch* belong to *receives*.

Line (18) for the predicate below in Appendix (F.11) is expressed in formula E2 below:

no status1.client2.(status1.receives) and

(forall ((ch Channel)(smac MAC))
(and(not(and(client2 status1 smac)(receives status1 smac indices t1 ch)))
(not(and(client2 status1 smac)(receives status1 smac indices1 t1 ch)))
(not(and(client2 status1 smac)(receives status1 smac letters t1 ch)))
(not(and(client2 status1 smac)(receives status1 smac letters1 t1 ch)))))

Formula E2:

(∀ ch:Channel ,smac:MAC. ¬ ((status1,smac) ∈ client2 ∧
(status1,smac,indices,t1,ch) in receives) ∧
¬ ((status1,smac) ∈ client2 ∧ (status1,smac,indices1,t1,ch) in receives) ∧
¬ ((status1,smac) ∈ client2 ∧ (status1,smac,letters,t1,ch) in receives) ∧
¬ ((status1,smac) ∈ client2 ∧ (status1,smac,letters1,t1,ch) in receives) ∧

The formula E2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *indices*, *t1*, and *ch* belong to *receives*; and *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *indices1*, *t1*, and *ch* belong to *receives*; and *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *letters*, *t1*, and *ch* belong to *receives*; and *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *letters1*, *t1*, and *ch* belong to *receives*.

Line (19) for the predicate below in Appendix (F.11) is expressed in formula F2 below:

no status1.client2.(status1.sends) and

(forall ((ch Channel)(smac MAC))
(and(not(and(client2 status1 smac)(sends status1 smac indices t ch)))
(not(and(client2 status1 smac)(sends status1 smac indices1 t ch)))
(not(and(client2 status1 smac)(sends status1 smac letters t ch)))
(not(and(client2 status1 smac)(sends status1 smac letters1 t ch)))))

Formula F2:

(∀ *ch:Channel ,smac:MAC.* ¬ *((status1,smac)* ∈ *client2* ∧
(status1,smac,indices,t,ch) in sends) ∧
¬ *((status1,smac)* ∈ *client2* ∧ *(status1,smac,indices1,t,ch) in sends)* ∧
¬ *((status1,smac)* ∈ *client2* ∧ *(status1,smac,letters,t,ch) in sends)* ∧
¬ *((status1,smac)* ∈ *client2* ∧ *(status1,smac,letters1,t,ch) in sends)* ∧

The formula F2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *indices*, *t*, and *ch* belong to *sends*; and *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *indices1*, *t*, and *ch* belong to *sends*; and *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *letters*, *t*, and *ch* belong to *sends*; and *status1* and *smac* do not belong to *client2*; and *status1*, *smac*, *letters1*, *t*, and *ch* belong to *sends*.

Line (20) for the predicate below in Appendix (F.11) is expressed in formula G2 below:

status2.client2.(status2.status) = Second_Communication_And_Exchanging_Letters and

(=>(forall ((cs1 ConnectionStatus)(cmac MAC))
(=>(and (client2 status2 cmac)(status status2 cmac cs1))
(isSecond_Communication_And_Exchanging_Letters cs1))
(forall ((cs2 ConnectionStatus))(cmac MAC))
(=>(isSecond_Communication_And_Exchanging_Letters cs2)
(and (client2 status2 cmac)(status status2 cmac cs2)))

Formula G2:

(∀ cs1:ConnectionStatus,cmac:MAC.(status2,cmac) ∈ client2
∧ (status2,cmac,cs1)∈ status =>(cs1) ∈
isSecondCommunicationAndExchangingLetters) =>
(∀ cs2:ConnectionStatus,cmac:MAC.(cs2) ∈
isSecondCommunicationAndExchangingLetters =>(status2,cmac) ∈ client2
∧ (status2,cmac,cs2)∈ status)

The formula G2 specifies constraints that: for all atoms *cs1* in *ConnectionStatus*, and *cmac* in *MAC*, such that *status2* and *cmac* belong to *client2*; and *status2*, *cmac*, and *cs1* belong to *status* then *cs1* belongs to *isSecondCommunicationAndExchangingLetters*. It follows that for all atoms *cs2* in *ConnectionStatus*, and *cmac* in *MAC*, such that *cs1* belongs to *isSecondCommunicationAndExchangingLetters* then *status2* and *cmac* belong to *client2*; and *status2*, *cmac*, and *cs1* belong to *status*.

Line (21) for the predicate below in Appendix (F.11) is expressed in formula H2 below:

status2.server.(status2.status) =
Second_Communication_And_Exchanging_Letters and

(=>(forall ((cs1 ConnectionStatus)(smac MAC))
(=>(and (server status2 smac)(status status2 smac cs1))
(isSecond_Communication_And_Exchanging_Letters cs1))
(forall ((cs2 ConnectionStatus))(smac MAC))
(=>(isSecond_Communication_And_Exchanging_Letters cs2)
(and (server status2 cmac)(status status2 smac cs2)))

Formula H2:

(∀ cs1:ConnectionStatus,smac:MAC.(status2,smac) ∈ server
∧ (status2,smac,cs1)∈ status =>(cs1) ∈
isSecondCommunicationAndExchangingLetters) =>
(∀ cs2:ConnectionStatus,smac:MAC.(cs2) ∈
isSecondCommunicationAndExchangingLetters =>(status2,smac) ∈ server
∧ (status2,smac,cs2)∈ status)

The formula H2 specifies constraints that: for all atoms *cs1* in *ConnectionStatus*, and *smac* in *MAC*, such that *status2* and *smac* belong to *server*; and *status2*, *smac*, and *cs1* belong to *status* then *cs1* belongs to *isSecondCommunicationAndExchangingLetters*. It follows that for all atoms *cs2* in *ConnectionStatus*, and *smac* in *MAC*, such that *cs1* belongs to *isSecondCommunicationAndExchangingLetters* then *status2* and *smac* belong to *server*; and *status2*, *smac*, and *cs1* belong to *status*.

Line (22) for the predicate below in Appendix (F.11) is expressed in formula I2 below:

status2.server in status2.visitors

(forall ((smac MAC))(=>(server status2 smac)(visitors status2 smac)))

Formula I2:

(∀ smac:MAC.(status2,smac) ∈ server =>(status2,smac) ∈ visitors

–

The formula I2 specifies constraints that: for all *smac* in *MAC*, such that the *status2* and *smac* belong to server then *status2* and *smac* belong to *visitors*.

Line (23) for the predicate below in Appendix (F.11) is expressed in formula J2 below:

status2.ispA i̸n status2.serviceProvider and

(forall ((ispa ISP))(=>(not(ispA status2 ispa))(serviceProvider status2 ispa)))

Formula J2:

(∀ ispa:ISP.(status2,ispa) ∉ ispA =>(status2,ispa) ∈ serviceProvider

The formula J2 specifies constraints that: for all *ispa* in *ISP*, such that *status2* and *ispa* do not belong to *ispA* then *status2* and *ispa* belong to *serviceProvider*.

Line (24) for the predicate below in Appendix (F.11) is expressed in formula K2 below:

status2.ispB in status2.serviceProvider and

(forall ((ispb ISP))(=>((ispB status2 ispb)(serviceProvider status2 ispb)))

Formula K2:

(∀ ispb:ISP.(status2,ispb) ∈ ispB =>(status2,ispb) ∈ serviceProvider

The formula K2 specifies constraints that: for all *ispb* in *ISP*, such that *status2* and *ispb* belong to *ispB* then *status2* and *ispb* belong to *serviceProvider*.

Line (25) for the predicate below in Appendix (F.11) is expressed in formula L2 below:

status2.ch1 i̸n status2.channel and

(forall ((ch Channel))(=>(not(ch1 status2 ch))(channel status2 ch)))

Formula L2:

*(∀ ch:Channel.(status2,ch) ∉ ch1 =>
(status2,ch) ∈ channel*

The formula L2 specifies constraints that: for all *ch* in *Channel*, such that *status2* and *ch* do not belong to *ch1* then *status2* and *ch* belong to *Channel.*

Line (26) for the predicate below in Appendix (F.11) is expressed in formula M2 below:

status2.ch2 in status2.channel and

(forall ((ch Channel))(=>(ch2 status2 ch)(channel status2 ch)))

Formula M2:

(∀ ch:Channel.(status2,ch) ∈ ch2 =>(status2,ch) ∈ channel)

The formula M2 specifies constraints that: for all *ch* in *Channel*, such that *status2* and *ch* belong to *ch2* then *status2* and *ch* belong to *Channel.*

Line (27) for the predicate below in Appendix (F.11) is expressed in formula N2 below:

no status2.client1.(status2.connection) and

(forall ((cmac MAC))(not(and(client1 status2 cmac)(connection status2 cmac isp))))

Formula N2:

$(\forall$ *cmac:MAC.* $\neg$ *((status2,cmac)* $\in$ *client1* $\wedge$ *(status2,cmac,isp)* $\in$ *connection))*

The formula N2 specifies constraints that: for all *cmac* in *MAC*, *status2* and *cmac* do not belong to *client1* and *status2*,*cmac*, and *isp* belong to *connection.*

Line (28) shows the predicate below in Appendix (F.11) is expressed in formula O2 below:

status2.client2.(status2.connection) =status2.ispB and

(forall ((isp11 ISP))(=>(=>(exists ((cmac MAC))
(and (client2 status2 cmac)(connection status2 cmac isp11)))(ispB status2 isp11))
(exists ((cmac2 MAC))(and(client2 status2 cmac2)(connection status2 cmac2 isp11)))))

Formula O2:

$(\forall$ *isp11:ISP* $(\exists$ *cmac:MAC.(status2,cmac)* $\in$ *client2* $\wedge$
(status2,cmac,isp11) $\in$ *connection =>(status2,isp11)* $\in$ *ispB)) =>*
$(\exists$ *cmac2:MAC.(status2,cmac2)* $\in$ *client2* $\wedge$ *(status2,cmac2,isp11)* $\in$ *connection*

The formula O2 specifies constraints that: for all *isp11* in ISP such that *status2* and *cmac* belong to *client2*; and *status2*, *cmac*, and *isp11* belong to *connection* then *status2* and *isp11* belong to *ispB*, follows that if there exist *cmac2* in *MAC* such that *status2* and *cmac2* belong to *client2*; and *status2*, *cmac2*, and *isp11* belong to *connection.*

Line (29) shows the predicate below in Appendix (F.11) is expressed in formula P2 below:

status2.server.(status2.connection)= status1.server.(status1.connection) and

(forall ((smac MAC)(isp11 ISP))(=>
(and (server status2 smac)(connection status2 smac isp11))
(and (server status1 smac)(connection status1 smac isp11))))
(forall ((smac1 MAC)(isp12 ISP))(=>
(and (server status1 smac1)(connection status1 smac1 isp12))
(and (server status2 smac1)(connection status2 smac1 isp12))))

Formula P2:

(∀ smac:MAC, isp11:ISP.(status2,smac) ∈ server ∧
(status2,smac,isp11) ∈ connection =>
(status1,smac) ∈ server ∧
(status1,smac,isp11) ∈ connection
(∀ smac1:MAC, isp12:ISP.(status1,smac1) ∈ server ∧
(status1,smac1,isp12) ∈ connection =>
(status2,smac1) ∈ server ∧
(status2,smac1,isp12) ∈ connection

The formula P2 specifies constraints that: for all *smac* in *MAC*, and *isp11* in ISP such that *status2* and *smac* belong to *server*; and *status2*, *smac*, and *isp11* belong to *connection* then *status1* and *smac* belong to *server*; and *status1*, *smac*, and *isp11* belong to *connection.* For all *smac1* in *MAC*, and *isp12* in ISP such that *status1* and *smac1* belong to *server*; and *status1*, *smac1*, and *isp12* belong to *connection* then *status2* and *smac1* belong to *server*; and *status2*, *smac1*, and *isp12* belong to *connection.*

Line (30) for the predicate below in Appendix (F.11) is expressed in formula Q2 below:

status2.client2.(status2.sends)=letters->t”->status2.ch2 and

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((cmac MAC))(and (client2 status2 cmac)
(sends status2 cmac d11 t11 ch11)))
(and(= letters d11)(= t2 t11)(ch2 status2 ch11)))
(exists ((cmac2 MAC))(and (client2 status2 cmac2)
(sends status2 cmac2 letters t2 ch11)))))

Formula Q2:

(∀ d11:Data,t11:Time,ch11:Channel (∃ cmac:MAC.(status2,cmac)
∈ client2 ∧ (status2,cmac,d11,t11,ch11) ∈ sends =>
(letters = d11) ∧ (t = t11) ∧ (status2,ch11) ∈ ch2) ∧
(∃ cmac1:MAC.(status2,cmac1) ∈ client2 ∧
(status2,cmac1,letters,t,ch11) ∈ sends)

The formula Q2 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *cmac* in *MAC* such that the *status2* and *cmac* belong to *client2*; and *status2*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *letters* equals *d11*, *t2* equals *t11*, and *status2*, *ch11* belong to *ch2*, follows that there exists *cmac2* in *MAC* such that the *status2* and *cmac2* belong to *client2*; and *status2*, *cmac2*, *letters*, *t2*, and *ch11* belong to *sends*.

Line (31) for the predicate below in Appendix (F.11) is expressed in formula R2 below:

status2.client2.(status2.sends) ≠ indices->t”->status2.ch2

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((cmac MAC))(and (client2 status2 cmac)
(sends status2 cmac d11 t11 ch11)))
(and(= indices d11)(= t2 t11)(ch2 status2 ch11)))
(exists ((cmac2 MAC))(and (client2 status2 cmac2)
(not(sends status2 cmac2 indices t2 ch11))))))

Formula R2:

(∀ d11:Data,t11:Time,ch11:Channel (∃ cmac:MAC.(status2,cmac)
∈ client2 ∧ (status2,cmac,d11,t11,ch11) ∈ sends =>
(indices = d11) ∧ (t2 = t11) ∧ (status2,ch11) ∈ ch2) =>
(∃ cmac2:MAC.(status2,cmac2) ∈ client2 ∧
(status2,cmac2,indices,t2,ch11) ∉ sends)

The formula R2 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *cmac* in *MAC* such that *status2* and *cmac* belong to *client1*; and *status2*, *cmac*, *d11*, *t11*, and *ch11* belong to *sends* then *indices* equals *d11*, *t2* equals *t11*, and *status2*, *ch11* belong to *ch2*, follows that there exists *cmac2* in *MAC* such that *status2* and *cmac2* belong to *client1*; and *status2*, *cmac2*, *indices*, *t2*, and *ch11* do not belong to *sends*.

Line (32) for the predicate below in Appendix (F.11) is expressed in formula S2 below:

status2.server.(status2.receives)=letters'->t"'->status2.ch2

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status2 smac)
(receives status2 smac d11 t11 ch11)))
(and(= letters1 d11)(= t3 t11)(ch2 status2 ch11)))
(exists ((smac2 MAC))(and (server status2 smac2)
(receives status2 smac2 letters1 t3 ch11)))))

Formula S2:

(∀ d11:Data,t11:Time,ch11:Channel (∃ smac:MAC.(status2,smac)
∈ server ∧ (status2,smac,d11,t11,ch11) ∈ receives =>
(letters1 = d11) ∧ (t3 = t11) ∧ (status1,ch11) ∈ ch2) =>
(∃ smac2:MAC.(status2,smac2) ∈ server ∧
(status2,smac2,letters1,t3,ch11) ∈ receives)

The formula S2 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *smac* in *MAC* such that *status2* and *smac* belong to *server*; and *status2*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *letters1* equals *d11*, *t3* equals *t11*, and *status2*, *ch11* belong to *ch2*, follows that if there exists *smac2* in *MAC* such that *status2* and *smac2* belong to *server*; and *status2*, *smac2*, *letters1*, *t3*, and *ch11* belong to *receives*.

Line (33) for the predicate below in Appendix (F.11) is expressed in formula T2 below:

status2.server.(status2.receives) ≠ indices'->t"'->status2.ch2

(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status2 smac)
(receives status2 smac d11 t11 ch11)))
(and(= indices1 d11)(= t3 t11)(ch2 status2 ch11)))
(exists ((smac2 MAC))(and (server status2 smac2)
(not(receives status2 smac2 indices1 t3 ch11))))))

Formula T2:

(∀ d11:Data,t11:Time,ch11:Channel (∃ smac:MAC.(status2,smac)
∈ server ∧ (status2,smac,d11,t11,ch11) ∈ receives =>
(indices1 = d11) ∧ (t3 = t11) ∧ (status2,ch11) ∈ ch1) =>
(∃ smac2:MAC.(status2,smac2) ∈ server ∧
(status2,smac2,indices1,t3,ch11) ∉ receives)

The formula T2 specifies constraints that: for all *d11* in *Data*, *t11* in *Time*, and *ch11* in *Channel*, if there exists *smac* in *MAC* such that *status2* and *smac* belong to *server*; and *status2*, *smac*, *d11*, *t11*, and *ch11* belong to *receives* then *indices1* equals *d11*, *t3* equals *t11*, and *status2*, *ch11* belong to *ch1*, follows that if there exists *smac2* in *MAC* such that *status2* and *smac2* belong to *server*; and *status2*, *smac2*, *indices1*, *t3*, and *ch11* do not belong to *receives*.

Line (34) for the predicate below in Appendix (F.11) is expressed in formula U2 below:

no status2.server.(status2.sends) and

(forall ((ch Channel)(smac MAC))
(and(not(and(server status2 smac)(sends status2 smac indices t2 ch)))
(not(and(server status2 smac)(sends status2 smac indices1 t2 ch)))
(not(and(server status2 smac)(sends status2 smac letters t2 ch)))
(not(and(server status2 smac)(sends status2 smac letters1 t2 ch)))))

Formula U2:

$(\forall$ *ch:Channel ,smac:MAC.* $\neg$ *((status2,smac)* $\in$ *server* $\wedge$
(status2,smac,indices,t2,ch) in sends) $\wedge$
$\neg$ *((status2,smac)* $\in$ *server* $\wedge$ *(status2,smac,indices1,t2,ch) in sends)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *server* $\wedge$ *(status2,smac,letters,t2,ch) in sends)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *server* $\wedge$ *(status2,smac,letters1,t2,ch) in sends)* $\wedge$

The formula U2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that *status2* and *smac* do not belong to *server*; and *status2*, *smac*, *indices*, *t2*, and *ch* belong to *sends*; and *status2* and *smac* do not belong to *server*; and *status2*, *smac*, *indices1*, *t2*, and *ch* belong to *sends*; and *status2* and *smac* do not belong to *server*; and *status2*, *smac*, *letters*, *t2*, and *ch* belong to *sends*; and *status2* and *smac* do not belong to *server*; and *status2*, *smac*, *letters1*, *t2*, and *ch* belong to *sends*.

Line (35) for the predicate below in Appendix (F.11) is expressed in formula V2 below:

no status2.client1.(status2.receives) and

(forall ((ch Channel)(smac MAC))
(and(not(and(client1 status2 smac)(receives status2 smac indices t3 ch)))
(not(and(client1 status2 smac)(receives status2 smac indices1 t3 ch)))

(not(and(client1 status2 smac)(receives status2 smac letters t3 ch)))
(not(and(client1 status2 smac)(receives status2 smac letters1 t3 ch)))))

Formula V2:

$(\forall$ *ch:Channel ,smac:MAC.* $\neg$ *((status2,smac)* $\in$ *client1* $\wedge$
(status2,smac,indices,t3,ch) in receives) $\wedge$
$\neg$ *((status2,smac)* $\in$ *client1* $\wedge$ *(status2,smac,indices1,t3,ch) in receives)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *client1* $\wedge$ *(status2,smac,letters,t3,ch) in receives)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *client1* $\wedge$ *(status2,smac,letters1,t3,ch) in receives)* $\wedge$

The formula V2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *indices*, *t3*, and *ch* belong to *receives*; and *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *indices1*, *t3*, and *ch* belong to *receives*; and *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *letters*, *t3*, and *ch* belong to *receives*; and *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *letters1*, *t3*, and *ch* belong to *receives.*

Line (36) for the predicate below in Appendix (F.11) is expressed in formula W2 below:

---

no status2.client2.(status2.receives)

(forall ((ch Channel)(smac MAC))
(and(not(and(client2 status2 smac)(receives status2 smac indices t3 ch)))
(not(and(client2 status2 smac)(receives status2 smac indices1 t3 ch)))
(not(and(client2 status2 smac)(receives status2 smac letters t3 ch)))
(not(and(client2 status2 smac)(receives status2 smac letters1 t3 ch)))))

Formula W2:

$(\forall$ *ch:Channel ,smac:MAC.* $\neg$ *((status2,smac)* $\in$ *client2* $\wedge$
(status2,smac,indices,t3,ch) in receives) $\wedge$
$\neg$ *((status2,smac)* $\in$ *client2* $\wedge$ *(status2,smac,indices1,t3,ch) in receives)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *client2* $\wedge$ *(status2,smac,letters,t3,ch) in receives)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *client2* $\wedge$ *(status2,smac,letters1,t3,ch) in receives)* $\wedge$

The formula W2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that *status2* and *smac* do not belong to *client2*; and *status2*, *smac*, *indices*, *t3*, and *ch* belong to *receives*; and *status2* and *smac* do not belong to *client2*; and *status2*, *smac*, *indices1*, *t3*, and *ch* belong to *receives*; and *status2* and *smac* do not belong to *client2*; and *status2*, *smac*, *letters*, *t3*, and *ch* belong to *receives*; and *status2* and *smac* do not belong to *client2*; and *status2*, *smac*, *letters1*, *t3*, and *ch* belong to *receives*.

Line (37) for the predicate below in Appendix (F.11) is expressed in formula X2 below:

no status2.client1.(status2.sends)

```
 (forall ((ch Channel)(smac MAC))
(and(not(and(client1 status2 smac)(sends status2 smac indices t2 ch)))
(not(and(client1 status2 smac)(sends status2 smac indices1 t2 ch)))
(not(and(client1 status2 smac)(sends status2 smac letters t2 ch)))
(not(and(client1 status2 smac)(sends status2 smac letters1 t2 ch)))))
```

Formula X2:

$(\forall$ *ch:Channel ,smac:MAC.* $\neg$ *((status2,smac)* $\in$ *client1* $\wedge$
(status2,smac,indices,t2,ch) in sends) $\wedge$
$\neg$ *((status2,smac)* $\in$ *client1* $\wedge$ *(status2,smac,indices1,t2,ch) in sends)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *client1* $\wedge$ *(status2,smac,letters,t2,ch) in sends)* $\wedge$
$\neg$ *((status2,smac)* $\in$ *client1* $\wedge$ *(status2,smac,letters1,t2,ch) in sends)* $\wedge$

The formula X2 specifies constraints that: for all *ch* in *Channel*, and *smac* in *MAC*, such that *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *indices*, *t2*, and *ch* belong to *sends*; and *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *indices1*, *t2*, and *ch* belong to *sends*; and *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *letters*, *t2*, and *ch* belong to *sends*; and *status2* and *smac* do not belong to *client1*; and *status2*, *smac*, *letters1*, *t2*, and *ch* belong to *sends*.

Line (38) for the predicate below in Appendix (F.11) is expressed in formula Y2 below:

---

status1.client1 in status1.visitors and status1.client2
$\not\text{in}$ status1.visitors and
status2.client1 $\not\text{in}$ status2.visitors and status2.client2
in status2.visitors and
((status1.client1) in status1.interseptMacs and (status2.client2)
$\not\text{in}$ status2.interseptMacs
or (status1.client1) $\not\text{in}$ status1.interseptMacs and (status2.client2)
in status2.interseptMacs
or (status1.client1) $\not\text{in}$ status1.interseptMacs and (status2.client2)
$\not\text{in}$ status2.interseptMacs)

(forall ((cmac1 MAC)(cmac2 MAC))(and(and (=>
(client1 status1 cmac1)(visitors status1 cmac1))(=>(client2 status1 cmac2)
(not(visitors status1 cmac2)))(=>(client1 status2 cmac1)(not(visitors status2 cmac1)))
(=>(client2 status2 cmac2)(visitors status2 cmac2)))
(or (and (=>(client1 status1 cmac1)(interseptMacs status1 cmac1))
(=>(client2 status2 cmac2)(not(interseptMacs status2 cmac2))))
(and (=>(client1 status1 cmac1)(not(interseptMacs status1 cmac1)))
(=>(client2 status2 cmac2)(interseptMacs status2 cmac2)))
(and(=>(client1 status1 cmac1)(not(interseptMacs status1 cmac1)))
(=>(client2 status2 cmac2)(not(interseptMacs status2 cmac2)))))))

Formula Y1:

$\forall$ *cmac1, cmac2:MAC.((status1,cmac1)* $\in$ *client1*
=>(status1,cmac1) $\in$ *visitors* $\wedge$ *(status1,cmac2)* $\notin$ *client2*
=>(status1,cmac2) $\notin$ *visitors* $\wedge$ *(status2,cmac1)* $\notin$ *client1*
=>(status2,cmac1) $\notin$ *visitors* $\wedge$ *(status2,cmac2)* $\in$ *client2 =>(status2,cmac2)*
$\in$ *visitors* $\wedge$*((status1,cmac1)* $\in$ *client1 =>(status1,cmac1)* $\in$ *interseptMacs* $\wedge$
(status2,cmac2) $\in$ *client2 =>(status2,cmac2)* $\notin$ *interseptMacs* $\vee$
(status1,cmac1) $\in$ *client1 =>(status1,cmac1)* $\notin$ *interseptMacs* $\wedge$
(status2,cmac2) $\in$ *client2 =>(status2,cmac2)* $\in$ *interseptMacs* $\vee$
(status1,cmac1) $\in$ *client1 =>(status1,cmac1)* $\notin$ *interseptMacs* $\wedge$
(status2,cmac2) $\in$ *client2 =>(status2,cmac2)* $\notin$ *interseptMacs)*

The formula Y1 specifies constraints that: for all atoms *cmac1*, *cmac2* in *MAC*, such that if the first client belongs to the *client1* in *status1*, the first client belongs to *visitors* in *status1*, and if the second client does not belong to *client2* in *status1*, the second client does not belong to *visitors* in *status1*, and if the first client does not belong to *client1* in *status2*, the first client does not belong to *visitors* in *status2*, and if the second client belongs to *client2* in *status2*, the second client belongs to *visitors* in *status2*.

And, if the first client belongs to *client1* in *status1*, the first client belongs to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, the second client does not belong to *interseptMacs* in *status2*. Or, if the first client belongs to *client1* in *status1*, the first client does not belong to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, the second client belongs to the *interseptMacs* in *status2*. Or, if the first client belongs to *client1* in *status1*, the first client does not belong to *interseptMacs* in *status1*, and if the second client belongs to *client2* in *status2*, the second client does not belong to *interseptMacs* in *status2*.

7.1.2.8 Assertion

Lines (39) in Appendix (F.11) is expressed to formula Z2 below:

MultiChannel implies indices = indices1 and letters = letters1

(and(= indices indices1)(= letters letters1))

Formula Z2:

$indices = indices1 \land letters = letters1$

The formula Z2 specifies constraints that: sending and receiving indices should be equal and sending and receiving letters should be equal.

7.2 Results

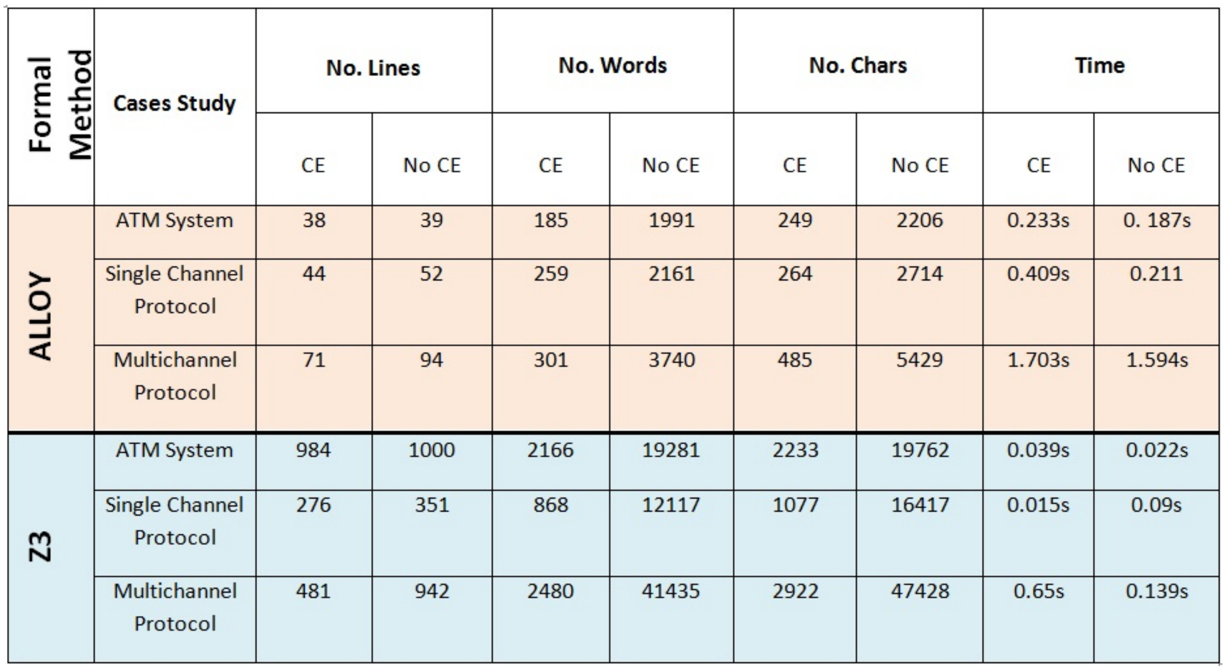
The negation of the assertion introduces that: the truth of the predicate and the achieved properties do not imply that data received equals data sent under all provided axioms, which conflicts our hypothesis.

We used *check-sat* as seen in Appendices (D.10,F.12) (Line 1) to ask the SMT solver to *check* whether the *negation* of the implication of the provided assertions is unsatisfiable or not. The SMT solver spent 0.09s and 0.039s to try to find a model that satisfied the negation of the set of formula but it did not find any.

However, without writing logical formulas as in Appendices (D.8, F.10) which restricted the properties of the first and the second protocol, the SMT solver provides SAT result in 0.015s for the first protocol and 0.65s for the second protocol. So that the instance found by the SMT solver is a counterexample to the assertion.

7.3 Discussion

Our goal from using Z3 SMT solver is to be more confident by proving the correctness of the results that we have achieved from the Alloy SAT solver. We will now compare the two approaches. First of all we compare the static size of specifications in Figure 7.1 in terms of the number of lines, the number of words, the number of characters for each component of our case studies in using both Alloy and Z3,and the time that SAT4J and SMT solvers spent to either generate a counterexample (CE), or no counterexample (no-CE).



Formal Method	Cases Study	No. Lines		No. Words		No. Chars		Time	
		CE	No CE	CE	No CE	CE	No CE	CE	No CE
ALLOY	ATM System	38	39	185	1991	249	2206	0.233s	0. 187s
	Single Channel Protocol	44	52	259	2161	264	2714	0.409s	0.211
	Multichannel Protocol	71	94	301	3740	485	5429	1.703s	1.594s
Z3	ATM System	984	1000	2166	19281	2233	19762	0.039s	0.022s
	Single Channel Protocol	276	351	868	12117	1077	16417	0.015s	0.09s
	Multichannel Protocol	481	942	2480	41435	2922	47428	0.65s	0.139s

Figure 7.1: Size and Time of Specifications

In general the number of lines, the number of words, and the number of characters for proving using Z3 is larger than using Alloy for all case studies whether there is a counterexample or not. The main reason for that is Alloy uses declarations to show the relations between entities and uses the multiplicity keywords to restrict the type of the relation, while Z3 uses functions and formulas to build the relations and specify them as functions.

In Alloy we see that the number of lines, the number of words, the number of characters, and and the time spent to generate a counterexample increase based on increasing the number of entities, the number of relations, the number of facts, the number of statuses in a predicate, and the number of properties in an assertion.

In Z3 we see that the number of lines, the number of words, and the number of characters increases based on increasing the number of: (1) extensions required to formulate how they are disjoint and to adjust return types (oneOf) as constants; (2) entities required to formulate how the abstract holds an extended entity. (3) multiplicity keywords. (4) relations, the number of facts, the number of statuses in a predicate, and the number of properties in an assertion.

Proving the ATM case study requires two abstracts and each abstract has at most six extensions. Also proving the multichannel protocol, requires one abstract and each abstract has at most two extensions increasing the number of lines, the number of words, and the number of characters.

As seen in Figure 6.2 Alloy and Z3 complement each other for several reasons:

- 1- Time for generating or solving a counterexample

Z3 is faster in searching for a counterexample as it spent 0.022s for proving the properties of the ATM model, while the SAT solver spent 0.187s for translation and checking the properties of the ATM model. Z3 spent 0.09s for solving the first protocol model and 0.139s for solving the second protocol model, while the SAT solver spent 0.211s for translation and checking the first protocol model and 1.594s for translation and solving the second protocol model.

In contrast, increasing the analysis scope to include more than one card, slows down the analysis in the Alloy Analyser. We find the same argument in [18] as when we model single channel and multichannel protocols which duplicates the scopes.

Moreover, when we increased the number of scopes from one channel to two channels, one ISP to two ISP, two datum to 4 datum, two times to four times, and one status to two statuses, the Alloy Analyser became slower and took longer in analysing the multichannel taking 1.703s compared to 0.409s for the single channel. To solve this disadvantage, [39] supposed that reducing the number of variables and increasing the number of constraints tends to reduce the solving time because it reduces the search space that the SAT solver must explore.

However, the Z3 SMT solver spent 0.015s for the first protocol and 0.65s for the second protocol. As seen in Figure 6.1, increasing the number of variables, the number of scopes, the number of properties, and the complexity lead to increasing the time that both the Alloy Analyser and Z3 spend to generate a counterexample.

- 2- Detecting, visualizing and understanding a counterexample

The common strength between Alloy and Z3 is providing a counterexample. However, in Alloy we could visualize the counterexample very easily. For example, in Figures 3.4, 6.2, 6.3, 6.5 when the SAT solver generated counterexamples which are instances that satisfy all the constraints, but violate the assertion. From this visualization we got the benefit to follow the flaws in the model and put more constrains until all instances satisfy the constraints, and do not violate the assertion as seen in Figures 3.5, 3.6, 3.7, 6.6, 6.7, 6.8. Visualising the model using Alloy may help us to find more constraints corresponding with the behaviour of the model that may not be taken in our account. For example, in the protocols model, we noticed that the model visualized that the client has an ability to receive and the server has an ability to send. However, in our model this is not allowed to be visualized. On the other hand, Z3 provides the counterexample as a formula and that requires expert understanding to extract and understand.

- 3- Limiting scopes of each type for checking the model

Alloy has the capability of limiting the state space that we need to look for a counterexample. For example, in the ATM system we needed to check the size of 1 atom for each signature except ATM 6 atoms. In the single channel protocol, we needed to check the size of 3 atoms for *MAC*, 2 atoms for *Time*, 2 atoms for *ISP*, 2 atoms for *Data*, 1 atom for *ConnectionStatus*, 1 atom for *Channel*, 1 atom for *Communication_Statu*. In the multichannel protocol, we needed to check the size of 3 atoms of *MAC*, 4 atoms of *Time*, 2 atoms of *ISP*, 2 atoms of *Channel*, 4 atoms of *Data*, 1 atom of *ConnectionStatus*, 2 atoms of *Communication_Status*. So, limiting scopes helps to detect any errors simply before developing the model to be complex with increasing the number of scopes gradually. Also it makes the Alloy analysis is noticeably faster. Moreover reducing the size of the model helps to accelerate and make more confident finding a counterexample.

- 4- Scanning all instances to get a counterexample

In Alloy we don't know where to stop increasing the scope to get a counterexample. So, we need to increase the size of scope and check each time. Consequently, we turned to Z3 as it checks the satisfiability for unbounded scope. For example, before restricting the properties of the ATM system, we checked the scopes from 1 to 4 and the analyser always said there was not a counterexample. When we turned to Z3, the counterexample appeared in one check. This advantage for Z3 also suggest to us that there is a counterexample in Alloy.

- 5- Confidence to begin with

Modelling a problem and then turning to prove it is useful because if the ease of using modelling with visualizing then turning to prove the result makes us feel more confident, especially when we got the same result in both formal methods.

- 6- Possibility of checking arbitrary relations

Z3 has an advantage in checking arbitrary sizes of relations. In Alloy when the universe contains more than 19 atoms as limited scope, the relations of arity 8 cannot be represented and the translation capacity will be exceeded.

- 7- Mistake percentage

In Alloy, the percentage of mistakes is higher because with limiting scope there may be an instance that causes a counterexample in a large scope. Also, with increasing the size of scope, there is an instance may be lost if the restriction in the relation between entities using the multiplicity keyword was not accurate.

- 8- An unrecognisable result

When running to check the satisfiability of a property, Z3 may return as a result the keyword `"unknown"`. So the property may or may not be valid so Z3 does not guarantee a complete analysis [63].

- 9- Syntax

Alloy provides a syntax which is easy to use directly without requiring one to write formulas to specify relations, extensions, abstracts, and multiplicity keywords.

- 10- Reason of model consistency

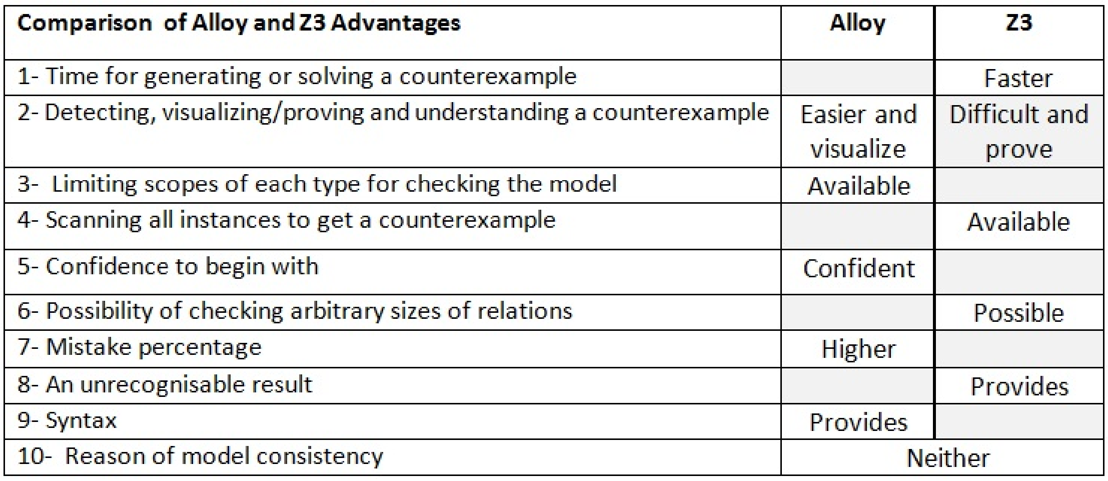
The common weakness, between Alloy and Z3 is that both do not tell why the model is inconsistent if it is inconsistent.

We conclude that, based on the results we achieved from analysing our protocol we cannot say that Alloy is the best choice for analysing a complex protocol compared to Z3 or vice versa. We noticed that both solvers complement each other.

When AA took a long time to generate a counterexample in limited scope, we found that Z3 is 11 times faster than AA. However, manually translating Alloy into a Z3 satisfiability-equivalent formulation took much more lines, words, and characters as seen in Figure 6.1. We tried to translate each line in Alloy into Z3 manually following the rules that will be provided that in next chapter covering Alloy syntax as seen in Figure 3.2 which Z3 does not have. We noticed that, after making the corresponding transformation, each part which causes a counterexample in Alloy also causes it in Z3 and if each restriction put in Alloy to solve the counterexample, is asserted in Z3 we got the same result. However, the manually translation takes long time and requires the user to be expert in relational logic to write accurate formulas corresponding Alloy formula to Z3 formulas.

Moreover, AA has a strength in visualizing the instances of the model and following the flows if there is a counterexample. This was a big advantage for Alloy as it gave us the ability to visualize mappings between variables and values to looking at counterexamples. In contrast, Z3 lacks that and presents the counterexample as a formula. Providing the syntax in Alloy and the advantage of visualizing the counterexample made us confident to begin with Alloy although the percentage of mistakes is higher.

In chapter 8 we show how a small portion of Alloy may be translated manually into a large amount of less readable Z3 including declaring types, subtypes, abstraction, extension, multiplicity constraints, relations, facts, assertions, formulas, and analysis.



Comparison of Alloy and Z3 Advantages	**Alloy**	**Z3**
1- Time for generating or solving a counterexample		Faster
2- Detecting, visualizing/proving and understanding a counterexample	Easier and visualize	Difficult and prove
3- Limiting scopes of each type for checking the model	Available	
4- Scanning all instances to get a counterexample		Available
5- Confidence to begin with	Confident	
6- Possibility of checking arbitrary sizes of relations		Possible
7- Mistake percentage	Higher	
8- An unrecognisable result		Provides
9- Syntax	Provides	
10- Reason of model consistency	Neither	

Figure 7.2: Comparison

Chapter 8

Systematic Translation Rules: A First Step Towards An Automated Translator

8.1 Introduction

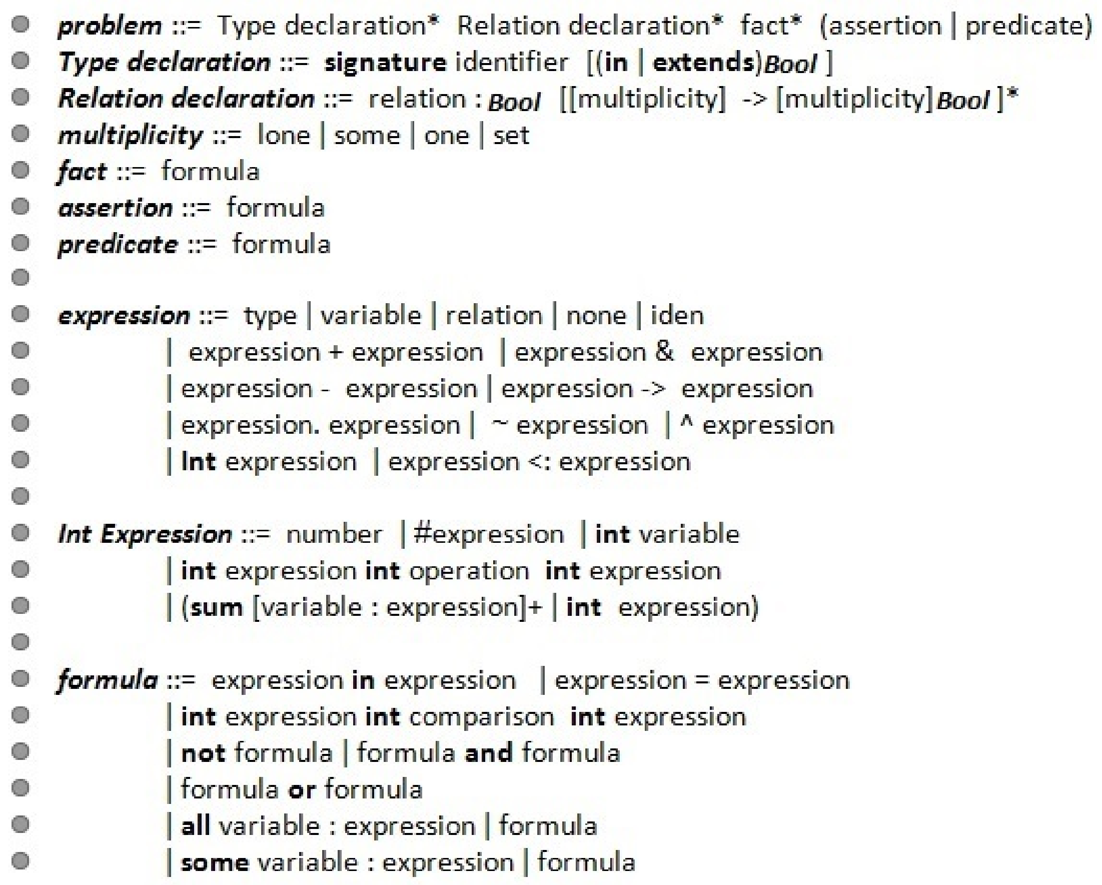
In this chapter we show how to translate a specification performed with Alloy into a satisfiability-equivalent SMT problem using Z3 SMT logic and solved by an SMT solver such that if there is a counterexample in Alloy in finite scopes, it supposed to be a counterexample in Z3 in infinite scopes and vice versa.

We clarify the translation from Alloy into Z3 including type declaration, relation declaration, multiplicity keywords, fact, assertion, predicate, expressions, and Formula.

This chapter has significance in:

- Helping researchers to understand how to express the same property in both Alloy and Z3.

- Guiding researchers to study the translation rules for each portion of the syntax of Alloy as seen in Figure 8.1 into Z3.
- Motivating researchers to build an automated tool to facilitate the translation operation from Alloy into Z3, saving time, effort and expert.
- Helping the interested researchers to combine model checkers and theorem provers and to harness their complementary strengths.



- ***problem*** ::= Type declaration* Relation declaration* fact* (assertion | predicate)
- ***Type declaration*** ::= **signature** identifier [(**in** | **extends**)*Bool*]
- ***Relation declaration*** ::= relation : *Bool* [[multiplicity] -> [multiplicity]*Bool*]*
- ***multiplicity*** ::= lone | some | one | set
- ***fact*** ::= formula
- ***assertion*** ::= formula
- ***predicate*** ::= formula
-
- ***expression*** ::= type | variable | relation | none | iden
- | expression + expression | expression & expression
- | expression - expression | expression -> expression
- | expression. expression | ~ expression | ^ expression
- | **Int** expression | expression <: expression
-
- ***Int Expression*** ::= number | #expression | **int** variable
- | **int** expression **int** operation **int** expression
- | (**sum** [variable : expression]+ | **int** expression)
-
- ***formula*** ::= expression **in** expression | expression = expression
- | **int** expression **int** comparison **int** expression
- | **not** formula | formula **and** formula
- | formula **or** formula
- | **all** variable : expression | formula
- | **some** variable : expression | formula

Figure 8.1: Abstract Syntax For The Core Alloy Logic [62]

8.2 The Alloy Syntax

A seen in Figure 8.1, the abstract syntax for the core Alloy logic contains three main parts: problem, expression, and formula.

- **Problem** is a collection of type declaration, relation declaration, fact, assertion, and predicate.

- **Type declaration:** declares all signatures of systems which represent sets of atoms. These signatures are for a top-level type and, a type subset of a type. For example, the signature declaration *sig X* declares a top-level type named X whereas the signature declaration

  $$sig\ Y\ extends\ X\{...\}$$

  $$sig\ Y\ in\ X\{...\}$$

  declares a type Y as a subtype (subset) of the type X.

- **Relation declaration and multiplicity keywords:** a relation is declared as a field of signatures. For example,

  $$sig\ A\{\},\ sig\ B\{\},\ sig\ C\ r : B\ m\ -> n\ A\}$$

which declares a ternary relation named

- **Fact, assertion, and predicate:** have been discussed in chapter 3 section 3.4.5.

- **Expressions:** Alloy expressions represent the fundamental buildings blocks of Alloy formula; they always evaluate to relations. There are two kinds of relational expressions, basic and complex. Basic Alloy expressions are constant relations; this includes all declared signatures and relations as well as the built-in constants: *sets* are unary relations, *scalars* are singleton unary relations, the built-in relation *none* denotes the empty set, and *none* for the unary empty set. Complex Alloy expressions are generated from basic expressions using Alloy's relational operators such as *r + s* (union), *r++s* (override), *r & s* (intersection), and *r - s* (difference) of same arity relations r and s. Also, *r ->s* for Cartesian product and *r.s* for relational join of arbitrary relations *r* and *s*.

Another expression provided in Alloy is integer expressions. An integer expression is different from relational expression. Integer value is not considered as in atom; however, utilizing an integer in relational expression, Alloy provides for every integer x value, Int contains exactly one atom that identifies that value [60]. They indicate rudimentary integers. The type Int represents the set of all atoms carrying rudimentary integers. The expression Int x denotes the atom carrying the integer denoted by the integer expression x, whereas int y denotes the integer value of the atom represented by the variable y. Integer expressions are obtained from an infinite set of Z numbers (. . . , -1, 0, 1, . . .), and combined using arithmetic operators (+ , -).

- **Formula:** fundamental Alloy formulas are formed from Alloy expressions utilizing the subset operator **in**, the equality operator =, the integer comparison operators less than < and greater than >, and the integer equality =. Fundamental formulas can be merged using logical connectivities including conjunction (**and** or &&), disjunction (**or** or |), implication (**implies** or =>), and negation (**not** or ¬).

  The quantified formula's form is *Q A:b | Formula*. *Q* denotes one of the *all*, *some*, *no*, *lone*, and *one*. The unary expression b bounds the quantification variable A, and Formula considered to be formula depends on A. However the expression b may not be begins with a multiplicity keyword. This is called a first order quantification, so A points to a single element of b. However, every Alloy expressions is considered relational, meaning that A is a singleton subset of b.

8.3 Z3 SMT solver

The Z3 SMT solver supports the SMT2 language which is the SMT-LIB standard version 2.0 [138]. Our formulas utilize the quantified theories of sorts, and uninterpreted functions with equality [138].

- **Declarations:** the logic of SMT2 language depends on a numerous sorted FOL with equality. It supports Integer, Real, and Boolean types, and enables users to declare new sorts (types) utilizing the command

  declare-sort

  Functions are the main structures of SMT formulas. SMT2 enables users to declare function utilizing the command

  (declare-fun f (arguments) Type)

  This command declares a function with a name *f*, that receives arguments $(A_1, ... , A_{n-1})$, and returns type (A_n).

  I.e,

  $$(declare - fun\ f\ (A_1, ..., A_{n-1})A_n)\ declares\ f : A_1 \times .... \times A_{n-1} - >A_n$$

  All functions are considered as *total* which means they are defined for every element of their domain.

  Constants are also considered as functions that do not take arguments. SMT2 enables users to declare function utilizing the command

  (declare-fun f () Type)

to declare a constant, where f is the name of the constant, and *Type* is the constant type.

- **Assertions:** the command *(assert f)* asserts a formula f in the present logical context. The main formulas are functions and can be connected utilizing the boolean operators *and* (conjunction), *or* (disjunction), *not* (negation), and $=>$ (implies).

  The universal quantifier is indicated by

  $$(forall\ (a_1\ A_1)...(a_n\ A_n)f)$$

  whereas the existential quantifier is indicated by

  $$(exists\ (a_1\ A_1)...(a_n\ A_n)f)$$

  as where a_1 is a variable and A_1 is the type of this variable.

- **Analysis:** we utilize the *check-sat* command to ask the SMT solver to check if the conjunction of the provided assertions is satisfiable or unsatisfiable.

8.4 Tool Integration and Methodology

Our framework provides seven stages for checking a property of an Alloy specification as seen in the section below. Checking Alloy specification within a bounded scope means finding counterexamples for the model in this bounded scope. However, no existence of a counterexample does not mean proof; it only means that non counterexample exists within the bounded scope.

Our framework also provides an SMT solver. If Z3 outputs *unsat*, the property has been proven correct, and if it outputs *sat*, a valid counterexample has been found. However, Z3 does not guarantee a complete analysis: it may output *unknown*, implying that the property may or may not be valid, or time out.

Section 8.5, describes the translation rules using a running example. It focuses on the main ideas in the translation involved in each stage to clarify their differences.

Our translation includes Alloy type declaration, relation declaration, fact, assertion, multiplicity keywords (one, lone, set), extension, abstract, and expressions such as conjunction, Cartesian product and relational join, and formulas.

8.4.1 Constructing Alloy Models

The Alloy modelling stages are:

- The first stage is determining the properties to be achieved in the system.

  second stage is to identify the main entities that interact in the system as signatures.

- The third stage is determining and constraining how the entities are related to each other. i.e, how many atoms on one side are related with an atom on the other side by constraining the sizes of the sets.

- The fourth stage involves constructing a predicate that describes the dynamic behaviour of the system.

- The fifth stage is to restrict the specification using facts. Facts are assumed to be true.

- The sixth stage is to build a formula as an assertion to check the validity of the model.

- The seventh stage is limiting the scope to gain more confidence about the correctness of a property.

8.4.2 Constructing Z3 Models

We have build our Z3 model by closely following the same stages as for Alloy except the seventh stage as Z3 is unbounded. Each stage in Alloy has been translated into its equivalent in Z3 to achieve the scale of the model as seen below. As the Z3 tool set has no editor, we begin by writing logical formulas representing the three properties of an Alloy model in the set of FOL formulas and declarations, and a sequence of commands in the Z3 stack.

8.5 Systematic Translation Rules

8.5.1 Type Declarations

This section gives the translation rules for Alloy type, subtype/extension, and abstraction which are represented in Figure 8.2. D in the figure defines Alloy type declarations, AT Alloy type, and SV SMT variables.

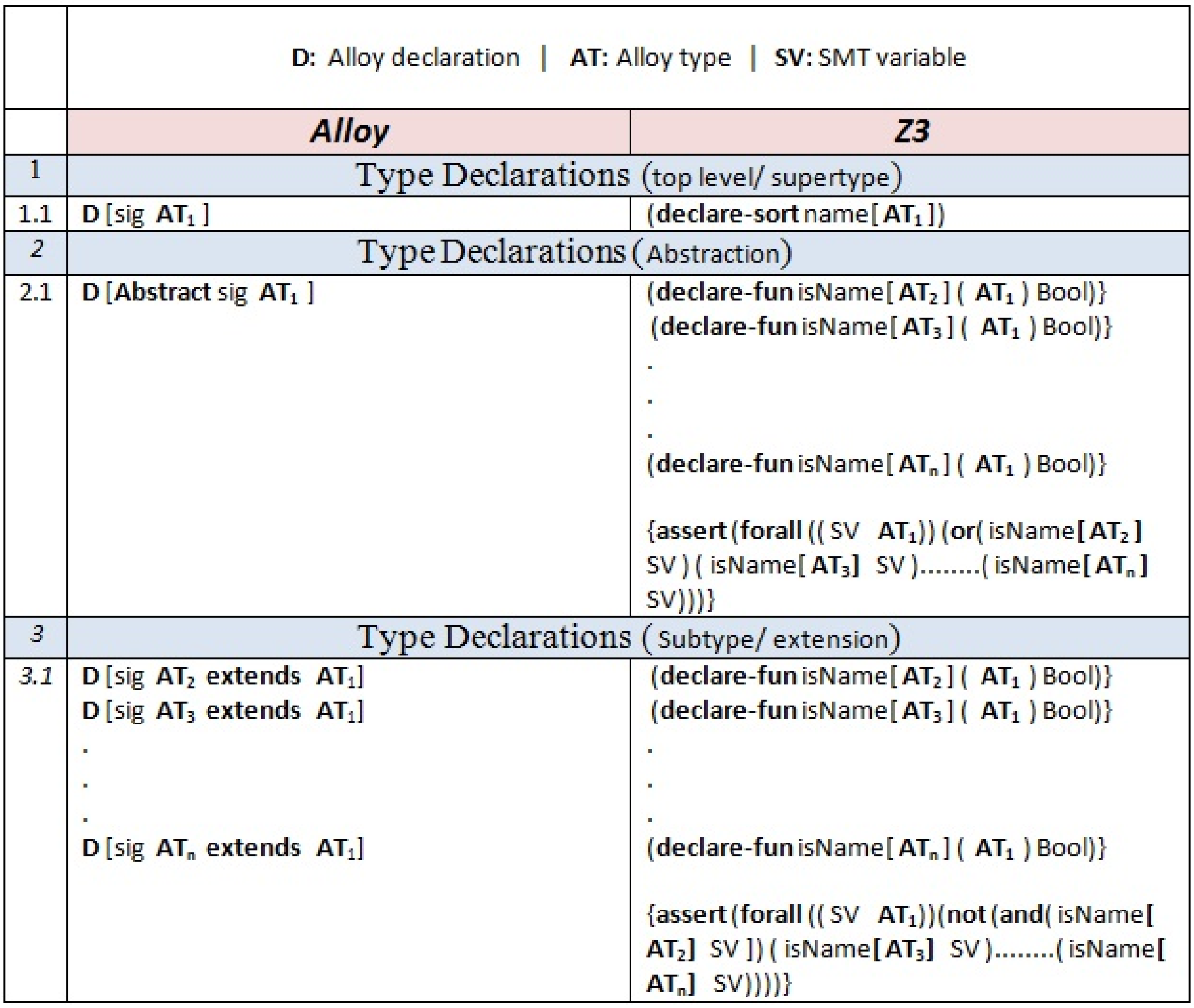
8.5.1.1 Signature Identifier

SMT2 enables users to declare a type declaration utilizing the command

declare-sort type

Each Top-level types in Alloy is translated to uninterpreted SMT2 sorts to identify atomic entities with no parameters as seen in Figure 8.2, line number 1.1. For example: the top-level type (Card):

$$sig\ Card\{\}$$



	D: Alloy declaration | AT: Alloy type | SV: SMT variable	
	Alloy	**Z3**
1	Type Declarations (top level/ supertype)	
1.1	**D** [sig **AT_1**]	(**declare-sort** name[**AT_1**])
2	Type Declarations (Abstraction)	

2.1	**D** [**Abstract** sig **AT_1**]	(**declare-fun** isName[**AT_2**] (**AT_1**) Bool)}
		(**declare-fun** isName[**AT_3**] (**AT_1**) Bool)}
		.
		.
		.
		(**declare-fun** isName[**AT_n**] (**AT_1**) Bool)}
		
		{**assert** (**forall** ((SV **AT_1**)) (**or**(isName[**AT_2**] SV) (isName[**AT_3**] SV)........(isName[**AT_n**] SV)))}

3	Type Declarations (Subtype/ extension)	

3.1	**D** [sig **AT_2 extends AT_1**]
	D [sig **AT_3 extends AT_1**]
	.
	.
	.
	D [sig **AT_n extends AT_1**]
		(**declare-fun** isName[**AT_2**] (**AT_1**) Bool)}
		(**declare-fun** isName[**AT_3**] (**AT_1**) Bool)}
		.
		.
		.
		(**declare-fun** isName[**AT_n**] (**AT_1**) Bool)}
		
		{**assert** (**forall** ((SV **AT_1**))(**not** (**and**(isName[**AT_2**] SV]) (isName[**AT_3**] SV)........(isName[**AT_n**] SV))))}

Figure 8.2: Translation Rules for Alloy Type Declarations

in Alloy is translated into Z3 to be declared as an uninterpreted sort:

$$(declare - sortCard)$$

8.5.1.2 Abstraction

Abstraction is not supported by SMT2. Thus, abstract signatures are required to be translated into SMT2 only through axioms as seen in Figure 8.2, line number 2.1, formulating that the abstract type has no elements except those which are extended AT_1 which constraints each element SV of this type to belong to one of its extending subtypes $\mathrm{AT}_2, AT_3, ..., AT_n$ using the general form

$$(forall((this\ SuperType))(or(isSubtype_1\ SV)(isSubtype_2\ SV)...(isSubtype_n\ SV)))$$

For example the abstract signature:

$$abstract\ sig\ Operations\{\}$$

$$one\ sig\ EnterCard, TypePin, RequistCash,$$

$$ReceiveCashAndCard, ReceiveCard\ extends\ Operations\{\}$$

in Alloy is translated into Z3 using an assertion that all sub sorts in the sort *Operation* should be one of them.

$$(forall((o\ Operations))(or(isEnterCard\ o)(isTypePin\ o)$$

$$(isRequistCash\ o)(isReceiveCashAndCard\ o)(isReceiveCard\ o)))$$

8.5.1.3 Subtype/Extension

Subtype declarations are not supported by SMT2. Thus, we translate Alloys hierarchical type system implicitly through axioms. To do that we need to use a *Boolean valued function* to declare an uninterpreted membership function called *is* to identify all sub sorts as seen in Figure 8.2, line number 3.1. An uninterpreted membership function is important to determine the semantics of subtypes for each Alloy type to represent the subtypes making all elements of the subtype belong to their supertype. A membership function takes the form of a command:

(declare-fun isSub-type (supertype) Bool)

The membership function takes only one parameter which is the extended sort AT_1 (superType) and returns the result as a Boolean to expresses a subset between two sorts AT_2, AT_3,...., AT_n (subTypes) and AT_1 (superType).

For example: the subtype *EnterCard* $\subseteq$ its supertype (*Operations*):

$$sig\ Operations\{\}$$

$$sig\ EnterCard\ extends\ Operations\{\}$$

in Alloy is translated into Z3 as a membership function using the command:

(declare-fun isEnterCard (Operations) Bool)

Extension between two types to make them disjoint is not supported by SMT2. Thus, extension signatures are required to be translated into SMT2 through axioms formulating that the extending subtypes AT$_2$, AT$_3$,...., AT$_n$ are disjoint as seen in Figure 8.2, line number 3.1 by using *not* and *and*, using the general form:

$$(forall((SV\ superType))(not(and(subType_1\ SV)(subType_2\ SV)...(subType_n\ SV)))$$

For example the extension signatures:

$$abstract\ sig\ Operations\{\}$$

$$one\ sig\ EnterCard, TypePin, RequistCash,$$

$$ReceiveCashAndCard, ReceiveCard\ extends\ Operations\{\}$$

in Alloy is translated into Z3 using an assertion that all sub sorts in the sort *Operation* are disjoint:

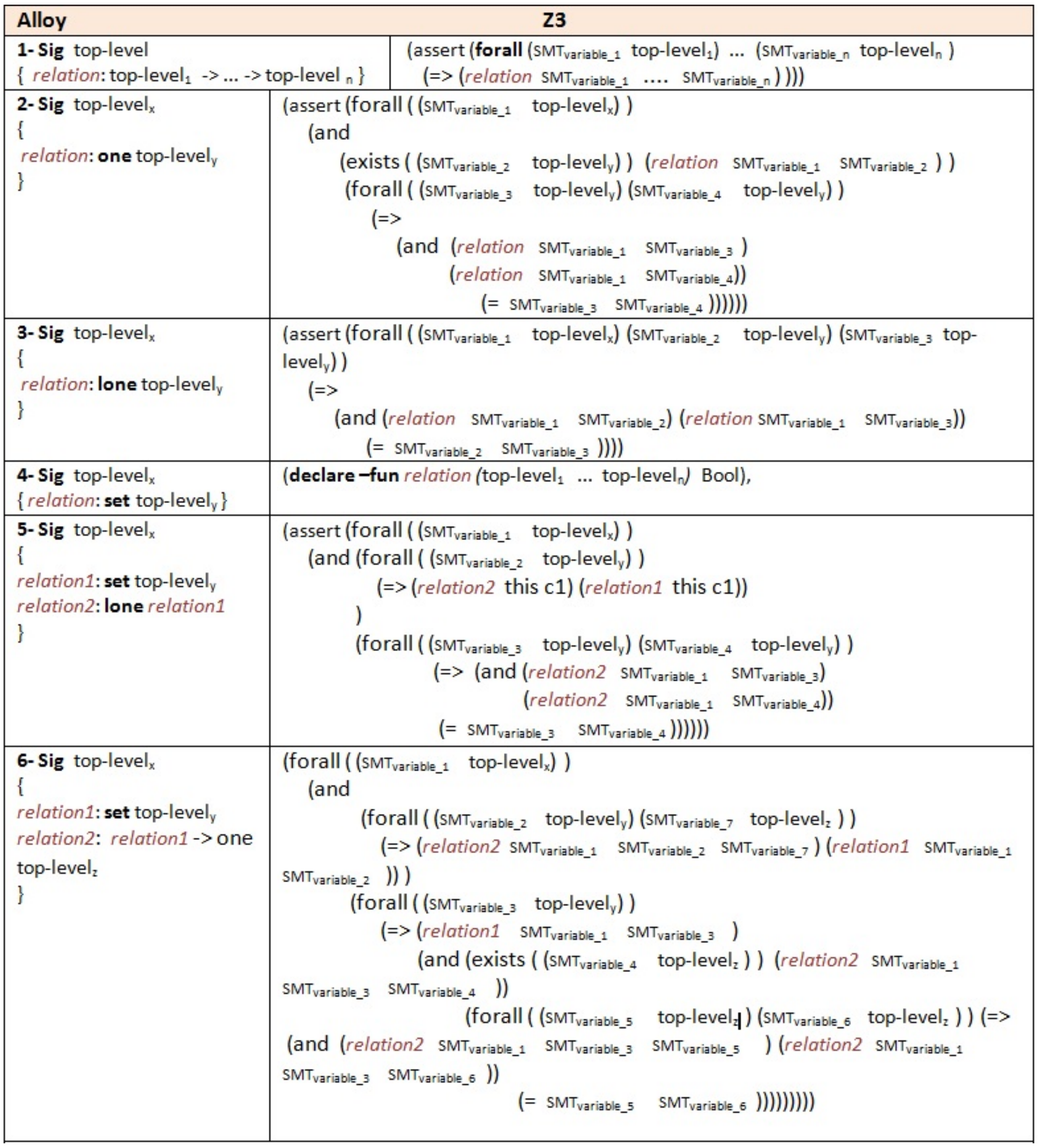
$$(forall((o\ Operations))(not(and(isEnterCard\ o)(isTypePin\ o))))$$

and that means all element *o* belongs to the super type *Operations*, *o* either belongs to subtype *isEnterCard* or subtype *isTypePin*.

8.5.2 Relation Declarations

8.5.2.1 Relation

An Alloy relation is translated to a Boolean-valued SMT2 function. This function is declared over top-level types because only top-level types are declared as sorts. As seen in Figure 8.3, a relation can be constraint in different domain.



Alloy	Z3
1- Sig top-level { *relation*: top-level$_1$ -> ... -> top-level $_n$ }	(assert (**forall** ($SMT_{variable\_1}$ top-level$_1$) ... ($SMT_{variable\_n}$ top-level$_n$) (=> (*relation* $SMT_{variable\_1}$ .... $SMT_{variable\_n}$))))
2- Sig top-level$_x$ { *relation*: **one** top-level$_y$ }	(assert (forall (($SMT_{variable\_1}$ top-level$_x$)) (and (exists (($SMT_{variable\_2}$ top-level$_y$)) (*relation* $SMT_{variable\_1}$ $SMT_{variable\_2}$)) (forall (($SMT_{variable\_3}$ top-level$_y$) ($SMT_{variable\_4}$ top-level$_y$)) (=> (and (*relation* $SMT_{variable\_1}$ $SMT_{variable\_3}$) (*relation* $SMT_{variable\_1}$ $SMT_{variable\_4}$)) (= $SMT_{variable\_3}$ $SMT_{variable\_4}$))))))
3- Sig top-level$_x$ { *relation*: **lone** top-level$_y$ }	(assert (forall (($SMT_{variable\_1}$ top-level$_x$) ($SMT_{variable\_2}$ top-level$_y$) ($SMT_{variable\_3}$ top-level$_y$)) (=> (and (*relation* $SMT_{variable\_1}$ $SMT_{variable\_2}$) (*relation* $SMT_{variable\_1}$ $SMT_{variable\_3}$)) (= $SMT_{variable\_2}$ $SMT_{variable\_3}$))))
4- Sig top-level$_x$ { *relation*: **set** top-level$_y$ }	(**declare –fun** *relation* (top-level$_1$ ... top-level$_n$) Bool),
5- Sig top-level$_x$ { *relation1*: **set** top-level$_y$ *relation2*: **lone** *relation1* }	(assert (forall (($SMT_{variable\_1}$ top-level$_x$)) (and (forall (($SMT_{variable\_2}$ top-level$_y$)) (=> (*relation2* this c1) (*relation1* this c1))) (forall (($SMT_{variable\_3}$ top-level$_y$) ($SMT_{variable\_4}$ top-level$_y$)) (=> (and (*relation2* $SMT_{variable\_1}$ $SMT_{variable\_3}$) (*relation2* $SMT_{variable\_1}$ $SMT_{variable\_4}$)) (= $SMT_{variable\_3}$ $SMT_{variable\_4}$))))))
6- Sig top-level$_x$ { *relation1*: **set** top-level$_y$ *relation2*: *relation1* -> one top-level$_z$ }	(forall (($SMT_{variable\_1}$ top-level$_x$)) (and (forall (($SMT_{variable\_2}$ top-level$_y$) ($SMT_{variable\_7}$ top-level$_z$)) (=> (*relation2* $SMT_{variable\_1}$ $SMT_{variable\_2}$ $SMT_{variable\_7}$) (*relation1* $SMT_{variable\_1}$ $SMT_{variable\_2}$))) (forall (($SMT_{variable\_3}$ top-level$_y$)) (=> (*relation1* $SMT_{variable\_1}$ $SMT_{variable\_3}$) (and (exists (($SMT_{variable\_4}$ top-level$_z$)) (*relation2* $SMT_{variable\_1}$ $SMT_{variable\_3}$ $SMT_{variable\_4}$)) (forall (($SMT_{variable\_5}$ top-level$_z$) ($SMT_{variable\_6}$ top-level$_z$)) (=> (and (*relation2* $SMT_{variable\_1}$ $SMT_{variable\_3}$ $SMT_{variable\_5}$) (*relation2* $SMT_{variable\_1}$ $SMT_{variable\_3}$ $SMT_{variable\_6}$)) (= $SMT_{variable\_5}$ $SMT_{variable\_6}$))))))))

Figure 8.3: Translation Rules for Alloy Relation Declarations

In the first relation (1) in Figure 8.3, a relation has no multiplicity constraint and in this case the constraints by default is *set*. In this relation, we need to assert that all variables belong to the top-levels, every tuple of a top-level is mapped to set of tuples of the related top-level.

8.5.2.2 Multiplicity

The general form of the function for multiplicities is

$$(declare-fun|oneOfsubType|()\ supertype)$$

For example the multiplicity keyword *one* before the sub signature *EnterCard*:

$$one\ sig\ EnterCard\ extends\ Operations\{\}$$

is translated into Z3 as:

$$(declare-fun|oneOfEnterCard|()\ Operations)$$

After declaring the function we need to adjust return types of the "oneOf" functions to match each element is declared in function *oneOf* with its corresponding *Operation* and to avoid making Z3 returns incorrect operation when it is called. So the adjustment return types of the "oneOf" functions are declared as:

$$(isSubType|oneOfsubType|)$$

meaning that any element returned by *oneOf* is always one, and its type is *subType*.

Multiplicity keyword *lone* forces the existence of two elements of a *supertype* to be equal. The general form of the function is

$$(forall\ ((this1\ supertype)(this2\ supertype))$$

$$(=>(and(issubType\ this1)(issubType\ this2))(=\ this1\ this2)))$$

For example:

$$(forall\ ((o1\ Operations)(o2\ Operations))$$

$$(=>(and(isEnterCard\ o1)(isEnterCard\ o2))(=\ o1\ o2)))$$

In the second relation (2) in Figure 8.3, has a relation has *one* multiplicity constraint. In this relation, we need to assert that for all variables belonging to the top-level.

Every tuple is mapped to exactly one tuple of the related top-level using the there exists quantifier.

In the third relation (3) in Figure 8.3, a relation has a *lone* multiplicity constraint. In this relation, we need to assert that for all variables belonging to the top-level, every tuple is mapped to at most one tuple of the related top-level restricting that if two different variables belong to the relation, they should be equal.

In the fourth relation (4) in Figure 8.3, a relation has *set* multiplicity constraint. In this relation, we only need to declare a relation with no need to write axioms as *set* may be empty.

In the fifth and sixth relations (5,6) in Figure 8.3, a relation has multiple relations. In this relation, we need to assert that the existence relation maps to an already existing relation first and then applying the constraints (*set, one, and lone*) as mentioned above.

For example in modelling the ATM system the relation *money* in Alloy:

$$sig\ ATM\{money : cards\ \ - > lone\ Int\}$$

is translated into Z3 as:

$$(declare - fun\ money(ATM\ Card\ Int)Bool)$$

In Alloy we placed the restriction that requiring the money for a card does not take place in the second status but in the fourth status as seen in the predicate below:

$$atm4.inCard.(atm4.money) = mon$$

This is translated into Z3 to be asserted as:

$$(and(inCard\ atm4\ c9)(money\ atm4\ c9\ i3))\ (1)$$

and:

$$no\ atm2.money$$

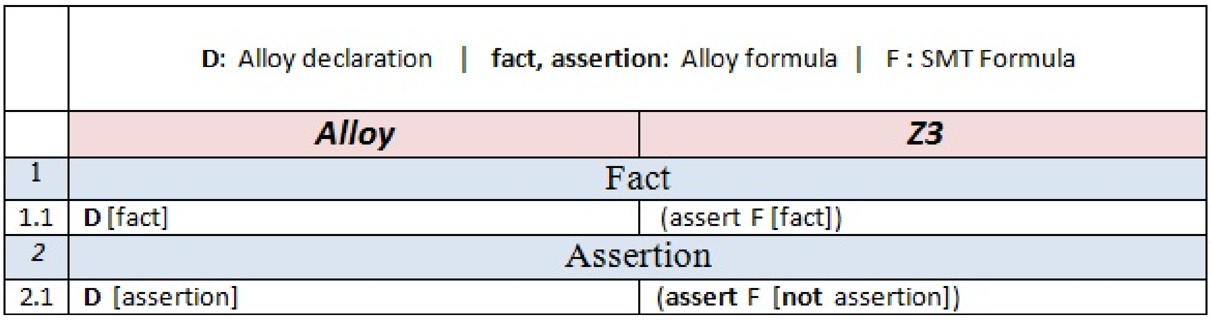
is translated into Z3 as:

$$(not(money\ atm2\ c9\ m))\ (2)$$

Then, to get the expressive power of the relation, the function returning Boolean controls the existence or non existence of a tuple in a relation *money*, by returning *true* making the tuple (atm4, c9, i3) exist in the relation *money* as seen in (1) above, and *false* making the tuple (atm2, c9, m) not exist in the relation *money* as seen in (2) above. The Boolean valued function method is applied to all relations.

8.5.3 Facts and Assertions

Figure 8.4 gives the translation rules for Alloy facts that are assumed to be true, and assertions that are intended to be checked. As seen in Figure 8.4, line 2.1, the assertion is negated, so when the SMT solver found any instance, this instance will be a counterexample to the assertion. Thus, if the solver does not find any instances, the assertion is correctly proven.



	D: Alloy declaration | **fact, assertion:** Alloy formula | F : SMT Formula	
	Alloy	***Z3***
1	Fact	
1.1	**D** [fact]	(assert F [fact])
2	Assertion	
2.1	**D** [assertion]	(assert F [**not** assertion])

Figure 8.4: Translation Rules for Fact and Assertion Formulas

8.5.4 Expression:

8.5.4.1 Cartesian Product

An expression *expression1* ->*expression2* contains a tuple

$$(t_1, .., t_n, .., t_{n+m})$$

iff *expression1* contains

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

and *expression2* contains $(\mathrm{t}_{n+1}, .., t_{n+m})$ where n is the arity of *expression1* and m is the arity of *expression2* as seen in line 5 Figure 8.5.

8.5.4.2 Relational Join

An expression *expression1* ...... *expression2* contains a tuple

$$\{(expression1, .., expression1_{m-1}, expression2, .., expression2_n)$$

$$|(expression1, .., expression_m) \in expression1 \wedge$$

$$(expression2_1, .., expression2_n) \in expression2 \wedge expression1_m = expression2_1\}$$

Relational join needs a quantified variable for the combined column of the two relations as seen in line 4 Figure 8.5.

8.5.5 Formulas

In Alloy formulas are formed using the subset operator *in* and the integer comparison operators such as greater than >, less than <, and equalities =, and combined using logical operators such as negation *not*, conjunction *and*, and disjunction *or*, relational join, and Cartesian product which are mapped to those in SMT2. As seen in Figure 8.5, an Alloy formula operates is for translated relational logic to correspond with Z3.

8.5.5.1 Subset

The Alloy formula (*expression1* in *expression2*) is well-formed only when the arity of *expression1* equals the arity of *expression2* and is translated by specifying that each element of *expression1* is included in *expression2* as seen in line 3 Figure 8.5.

8.5.5.2 Negation

The Alloy formula *(not expression1)* is translated into Z3 by negating expressions making the new formula empty as seen in line 6 Figure 8.5.

8.5.5.3 Conjunction

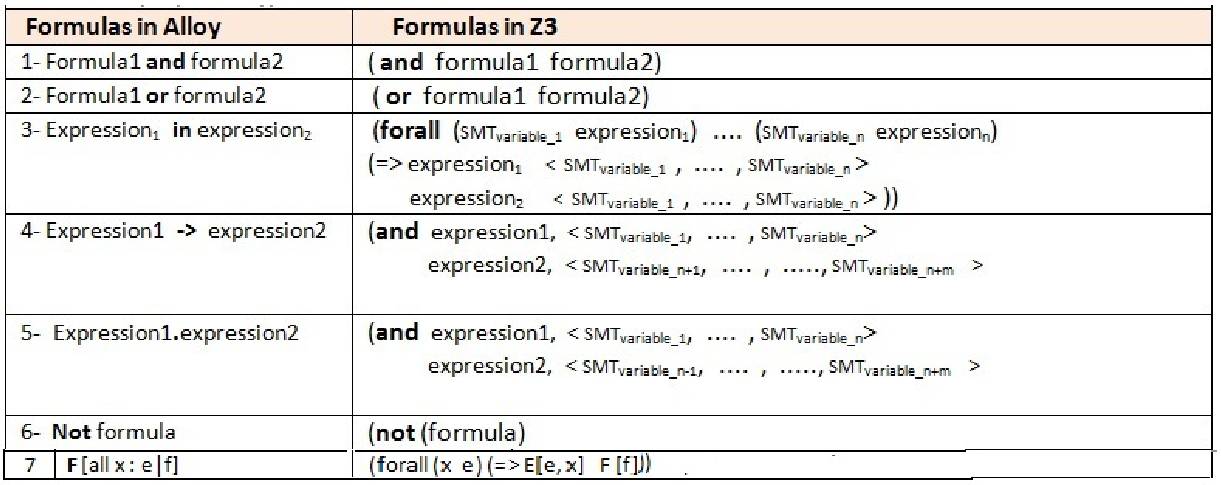
The Alloy formula *(expression1 and expression2)* is translated into Z3 by joining *expression1* and *expression2* making the new formula includes all variables of the two expressions together as seen in line 1 Figure 8.5.

8.5.5.4 Disjunction

The Alloy formula *(expression1 or expression2)* is translated into Z3 by joining *expression1* or *expression2* making the new formula includes the variables of one of the two expressions which one achieved the condition as seen in line 2 Figure 8.5.

8.5.5.5 All

As seen in line 7 Figure 8.5, for all x e, if the expression for *e* achieved for all *x*, then the function *f* is true.



Formulas in Alloy	Formulas in Z3
1- Formula1 **and** formula2	(**and** formula1 formula2)
2- Formula1 **or** formula2	(**or** formula1 formula2)
3- Expression$_1$ **in** expression$_2$	(**forall** (SMT$_{variable\_1}$ expression$_1$) .... (SMT$_{variable\_n}$ expression$_n$) (=> expression$_1$ < SMT$_{variable\_1}$, ... , SMT$_{variable\_n}$ > expression$_2$ < SMT$_{variable\_1}$, ... , SMT$_{variable\_n}$ >))
4- Expression1 **->** expression2	(**and** expression1, < SMT$_{variable\_1}$, ... , SMT$_{variable\_n}$> expression2, < SMT$_{variable\_n+1}$, ... , ..., SMT$_{variable\_n+m}$ >
5- Expression1**.**expression2	(**and** expression1, < SMT$_{variable\_1}$, ... , SMT$_{variable\_n}$> expression2, < SMT$_{variable\_n-1}$, ... , ..., SMT$_{variable\_n+m}$ >
6- **Not** formula	(**not** (formula)
7 F [all x : e | f]	(forall (x e) (=> E[e, x] F [f]))

Figure 8.5: Formulas Constraints

8.5.6 The general form for translation

The general form for translating Alloy problem into Z3 is expressed as:

$$(not\ (\,=\,>\,(A\ B)))$$

A includes all declared sorts; sub sorts; constraints; facts; and relations, and predicates, while B includes the assertions. So, if the SMT solver found any instance that achieved the validity of the general form that we used, it will be a counterexample to the assertion. However, if the SMT solver did not find any instance, the negation of the assertion is proven correct.

Chapter 9

Conclusion and Future Work

9.1 Conclusion

We have presented an approach for analysing single and multichannel protocols expressed in Alloy and Z3. We presented an approach to translate specifications of the protocols that were performed with Alloy into a satisfiability-equivalent SMT problem using Z3. We presented seven stages that we have used to model the protocols using Alloy. We showed how these seven stages have been translated into Z3.

The contributions of this thesis have been as follows:

- Modelling three case studies using Alloy.
- Checking the validity of their properties to see if they satisfy their requirements using the SAT solver.
- Gaining confidence in the correctness of the properties by modelling the case studies using Z3.
- Proving the correctness of the properties of the case studies using the SMT solver.
- Comparing the results that were achieved through Alloy and Z.

- Determining and comparing the strengths, limitations, and advantages of using these formal methods.

These case studies are an ATM system, which uses as an example to introduce Alloy and Z3, followed by two security protocols of transmitting data over a single channel and multichannel.

The thesis also offers an approach to manually building satisfiability-equivalent SMT problem of the specification of the case studies that were performed first with Alloy by proving the same model properties as in Alloy and searching for a counterexample using Z3 in infinite scopes which is supposed to be equivalent to Alloy in finite scopes and vice versa.

Throughout this thesis we have tried to identify the strengths, weaknesses and the limitations of Alloy and Z3 to decide which method is strong enough to depend on to achieve confidence in results. However, during the work we noticed that the formal methods complemented each other: Alloy for modelling properties and Z3 for proving the properties and we turned from one to another to get the benefit from one we did not find it in the other.

Since AA could find small a counterexamples well, we suggest that the user begins by using Alloy to check the validity of the assertion and then turns to Z3 to prove the validity of the assertion.

We found that Alloy facilitated building the restricted relations between interacting entities using constraints already provided such as multiplicities keyword, abstract, and extends. Also, after building the models, the Alloy Analyser has an advantage in determining if the model is consistent or not if it finds instances of a model automatically by search within scope.

However, this advantage turned into a disadvantage if the Alloy Analyser returns that the model is inconstant because it did not show or say why. When we turn to Z3, it has no advantage in saying why the model is inconsistent as well. The consistency of a model is very important before checking the assertions because if a model is inconsistent, the Alloy Analyser could not find instances to visualize. As a result, the SAT solver could not generate a counterexample.

The SAT solver in Alloy also has an advantage in requiring a limited scope for each type of search for a counterexample to accelerate detecting flaws and generating a counterexample if one exists. However, this advantage in Alloy turned into a disadvantage as we could not recognize how many scopes we had to inspect to see if there is a counterexample. It may be found in a larger scopes, and the absence of an instance does not include checking of satisfiability. In addition, the more the number of scopes is increased, the slower becomes detecting flaws and generating a counterexample. So, we needed to turn to the unbounded SMT solver to be more confident in covering all instances and providing a counterexample more quickly than with the SAT solver.

Alloy also has an advantage in visualized a counterexample to detect and follow the flaws easily and thus, add more restrictions, and properties. However, the SAT solver could not prove why the counterexample existed. So, we needed to turn to the unbounded SMT solver to prove that. As a result we conclude that both Alloy and Z3 are required to model and prove the properties of a problem as we noticed that any counterexample provided in Alloy, is provided in Z3 under the same restrictions.

When we used Alloy and Z3 to model an ATM system we have noticed that Alloy has advantage in helping non experts understand the problem well through visualizing a counterexample, saving time and effort. Moreover, Alloy visualizes why the system is incorrect and from that the user can learn what caused the problem in the specification and how to fix the problem.

Also we noticed that Z3 increased our confidence about getting results from Alloy when it gave the same results. Furthermore, it could provide the results faster than Alloy regardless of the number of scopes or if the model is becoming more complicated.

The results we achieved from modelling the ATM system using Alloy and Z3, encouraged us to use them to model our protocols. As a result we did not see any difference in the protocol results compared to the ATM system results. Alloy helped in understanding the flaws in the protocols and expressed why the first protocol was insecure using a single channel and how to develop it; from that the user can learn what caused an insecurity in the specification and how to fix it.

9.2 Future Work

In the future work we planned to:

- Translate Alloy specifications into satisfiability-equivalent SMT problems using SMT-based Bounded Verification.

We found that the bounded SAT solver which is decidable and the SMT solver which is undecidable complemented each other as the first is for modelling the satisfiability problem for bounded scope and the other is for proving the satisfiability problem for unbounded scopes. However, we wish to study whether using a bounded SAT solver which is decidable and a bounded SMT solver which lies within QBVF (quantified bit-vector formula), and thus is decidable [155] complement each other as well as an unbounded SMT solver. Do the the top-level types of an Alloy problems that are translated into SMT bitvectors, according to the scope information, provide the same results as the bounded SAT solver does?. Or do we really need an unbounded SMT solver to get the confidence?.

- Add more properties to the protocol and study if Alloy and Z3 still have the same power and still complement each other. Such properties might include:
  - Developing the multichannel protocol to be more complicated by encrypting data.
  - Increasing the number of MitM and their abilities to intercept both channels.

We showed that for Alloy, as the number of scopes increased more as increasing , the more time as is needed to get the results. Also, as the complexity of the model increased, the relations become more complicated. Therefore with complicated relations and increasing the number of scopes the translation capacity may be exceeded because the universe contains excessive atoms, and relations of a huge arity cannot be represented. So, could Z3 solve the state space explosion problem that is caused by Alloy when a model becomes bigger?

- Developing a new application using the same method

In future, if we develop a new security protocol we will follow as same methodology, using same the formal methods we have used in our studied protocol to see if they still keep their power in working together.

- Tool support for working between Alloy and Z3

Alloy and Z3 have their own advantages. So, designing an automated translator tool to translate the relational logic from Alloy into Z3 using our systematic translation rules may combine their advantages. The translator works as bridge which converts a problem from Alloy into Z3 easily.

The tool would save time, effort and expertise of the user.

As Z3 SMT provides the counterexample as formulas and AA provides it as visualization, we suggest that the tool will be more valuable if it could convert the counterexample from formula into visualization the same as AA does. We did not offer a methodology for this suggestion in our thesis.

As a result, we will have a tool that includes all the advantages of Alloy and Z3 which may help in achieving stronger results combining between modelling and proving.

Appendix A

ATM Model Using Alloy

Listing A.1: Signatures

```
module examples/systems/ATM_System
abstract sig Operations{}
one sig EnterCard,TypePin,RequistCash,ReceiveCashAndCard,
ReceiveCard extends Operations{}
abstract sig ATM_Status {}
one sig WaitingCard,WaitingPin,WaitingMoney,
WaitingReceiveCashAndCard,WaitingReceiveCard,Update
extends ATM_Status{}
sig Card{}
sig ATM {
```

Listing A.2: Relations Declaration

```
cards: set Card,
inCard : lone cards ,
pin :  cards -> one Int ,
balance: cards -> one Int,
money:cards -> lone Int,
atmStatuse:  one  ATM_Status,
op: lone Operations}
```

Listing A.3: Facts

```
fact{all atm1,atm2: ATM|atm2.cards=atm1.cards and atm2.pin =
atm1.pin}
fact{all atm1:ATM,card:atm1.cards|card.(atm1.balance)>=0 and
card.(atm1.pin)>0}
```

Listing A.4: Predicates

```
pred ATMTransaction [atm1,atm2,atm3,atm4,atm5,atm6:ATM,
crd:Card, pn, mon:Int]{
(atm1.atmStatuse)= WaitingCard and
crd in atm1.cards and
no atm1.inCard and no atm1.op and no atm1.money
and atm2.op= EnterCard and
atm2.inCard = crd and
atm2.balance = atm1.balance and
(atm2.atmStatuse) = WaitingPin  and no atm2.money
and atm3.op= TypePin and
atm3.inCard = atm2.inCard and
atm3.inCard.(atm3.pin)=pn and
atm3.balance = atm2.balance and
(atm3.atmStatuse) = WaitingMoney and no atm3.money
and atm4.op= RequistCash and
atm4.balance = atm3.balance and
atm4.inCard = atm3.inCard and atm4.inCard .(atm4.money)=mon
and atm5.inCard = atm4.inCard and
((( mon > atm4.inCard.(atm4.balance) or  mon <0) and
(atm5.inCard.(atm5.balance) = atm4.inCard.(atm4.balance) and
atm5.op= ReceiveCard and (atm4.atmStatuse)= WaitingReceiveCard))
or
(( mon <= atm4.inCard.(atm4.balance) and  mon >0) and
(atm5.inCard.(atm5.balance)=atm4.inCard.(atm4.balance).minus[mon]
and atm5.op= ReceiveCashAndCard and
(atm4.atmStatuse)=WaitingReceiveCashAndCard))) and
(atm5.atmStatuse) = Update and
no atm6.inCard and (atm6.atmStatuse)=WaitingCard and no atm6.op}
```

Listing A.5: Assertion

```
assert prop1 {
all atm1,atm2,atm3,atm4,atm5,atm6: ATM,pn,mon:Int, crd:Card|
mon <= crd.(atm1.balance)  and   mon >0 and
ATMTransaction[atm1,atm2,atm3,atm4,atm5,atm6,crd,pn,mon]
implies crd.(atm5.balance) = crd.(atm1.balance).minus[mon] }
```

Listing A.6: Commands

```
check prop1  for  1 but 6 ATM
run ATMTransaction  for  1 but 6 ATM
```

Appendix B

ATM Proving Using Z3 Theorem Prover

Listing B.1: Sorts

```
(declare-sort Operations )
(declare-sort ATM_Status )
(declare-sort Card )
(declare-sort ATM )
```

Listing B.2: Functions

```
(declare-fun cards (ATM Card ) Bool)
(declare-fun inCard (ATM Card ) Bool)
(declare-fun pin (ATM Card Int ) Bool)
(declare-fun balance (ATM Card Int ) Bool)
(declare-fun money (ATM Card Int ) Bool)
(declare-fun atmStatuse (ATM ATM_Status ) Bool)
(declare-fun op (ATM Operations ) Bool)
(declare-fun isEnterCard (Operations ) Bool)
(declare-fun isTypePin (Operations ) Bool)
(declare-fun isRequistCash (Operations ) Bool)
(declare-fun isReceiveCashAndCard (Operations ) Bool)
(declare-fun isReceiveCard (Operations ) Bool)
(declare-fun isWaitingCard (ATM_Status ) Bool)
(declare-fun isWaitingPin (ATM_Status ) Bool)
(declare-fun isWaitingMoney (ATM_Status ) Bool)
(declare-fun isWaitingReceiveCashAndCard (ATM_Status ) Bool)
(declare-fun isWaitingReceiveCard (ATM_Status ) Bool)
(declare-fun isUpdate (ATM_Status ) Bool)
```

Listing B.3: Some Property

```
;; the some property of the signatures <EnterCard>, <TypePin>,
;<RequistCash>, <ReceiveCashAndCard>, <ReceiveCard>
(declare-fun oneOf_EnterCard () Operations)
(declare-fun oneOf_TypePin () Operations)
(declare-fun oneOf_RequistCash () Operations)
(declare-fun oneOf_ReceiveCashAndCard () Operations)
(declare-fun oneOf_ReceiveCard () Operations)
;; the some property of the signatures <WaitingCard>, <WaitingPin>,
;<WaitingMoney>, <WaitingReceiveCashAndCard>,<WaitingReceiveCard>,
;<Update>
(declare-fun oneOf_WaitingCard () ATM_Status)
(declare-fun oneOf_WaitingPin () ATM_Status)
(declare-fun oneOf_WaitingMoney () ATM_Status)
(declare-fun oneOf_WaitingReceiveCashAndCard () ATM_Status)
```

```
(declare-fun oneOf_WaitingReceiveCard () ATM_Status)
(declare-fun oneOf_Update () ATM_Status)
```

Listing B.4: Negation Of The Implication Of The Assertion

```
(assert
(not
(=>
(and
```

Listing B.5: Return Types Of The "oneOf" Functions/Constants

```
(isEnterCard |oneOf_EnterCard|)
(isTypePin |oneOf_TypePin|)
(isRequistCash |oneOf_RequistCash|)
(isReceiveCashAndCard |oneOf_ReceiveCashAndCard|)
(isReceiveCard |oneOf_ReceiveCard|)
(isWaitingCard |oneOf_WaitingCard|)
(isWaitingPin |oneOf_WaitingPin|)
(isWaitingMoney |oneOf_WaitingMoney|)
(isWaitingReceiveCashAndCard |oneOf_WaitingReceiveCashAndCard|)
(isWaitingReceiveCard |oneOf_WaitingReceiveCard|)
(isUpdate |oneOf_Update|)
```

Listing B.6: The Lone Property Of The Signatures

```
;; the lone property of the signatures: <EnterCard>, <TypePin>,
;<RequistCash>, <ReceiveCashAndCard>, <ReceiveCard>
(forall ((o1 Operations)(o2 Operations))(=>
(and(isEnterCard o1)(isEnterCard o2))(= o1 o2)))
(forall ((o1 Operations)(o2 Operations))(=>
(and(isTypePin o1)(isTypePin o2))(= o1 o2)))
(forall ((o1 Operations)(o2 Operations))(=>
(and(isRequistCash o1)(isRequistCash o2)) (= o1 o2)))
(forall ((o1 Operations)(o2 Operations))(=>
(and(isReceiveCashAndCard o1)(isReceiveCashAndCard o2))
(= o1 o2)))
(forall ((o1 Operations)(o2 Operations))(=>
(and(isReceiveCard o1)(isReceiveCard o2))(= o1 o2)))

;; The lone property of the signatures: <WaitingCard>,
<WaitingPin>,<WaitingMoney>,<WaitingReceiveCashAndCard>,
;<WaitingReceiveCard>,<Update>
(forall ((a1 ATM_Status)(a2 ATM_Status))(=>
(and(isWaitingCard a1)(isWaitingCard a2)) (= a1 a2)))
(forall ((a1 ATM_Status)(a2 ATM_Status))(=>
(and(isWaitingPin a1)(isWaitingPin a2)) (= a1 a2)))
(forall ((a1 ATM_Status)(a2 ATM_Status))(=>
(and (isWaitingMoney a1)(isWaitingMoney a2))(= a1 a2)))
(forall ((a1 ATM_Status)(a2 ATM_Status))(=>
(and(isWaitingReceiveCashAndCard a1)
(isWaitingReceiveCashAndCard a2))(= a1 a2)))
(forall ((a1 ATM_Status)(a2 ATM_Status))(=>
(and(isWaitingReceiveCard a1)
(isWaitingReceiveCard a2)) (= a1 a2)))
(forall ((a1 ATM_Status)(a2 ATM_Status))(=>
(and(isUpdate a1)(isUpdate a2))(= a1 a2)))
```

Listing B.7: Abstract and Extension property

```
;; abstract property of signature <Operations>
(forall ((o Operations))(or(isEnterCard o)(isTypePin o)
(isRequistCash o)(isReceiveCashAndCard o)
(isReceiveCard o)))

;; the extends property of the signatures <EnterCard>,
;<TypePin>, <RequistCash>, <ReceiveCashAndCard>,
;<ReceiveCard> to the signature <Operations>
(forall ((o Operations))(not(and
(isEnterCard o)(isTypePin o))))
(forall ((o Operations))(not(and
(isEnterCard o)(isRequistCash o))))
(forall ((o Operations))(not(and
(isEnterCard o)(isReceiveCashAndCard o))))
(forall ((o Operations))(not(and
(isEnterCard o)(isReceiveCard o))))
(forall ((o Operations))(not(and
(isTypePin o)(isRequistCash o))))
(forall ((o Operations))(not(and
(isTypePin o)(isReceiveCashAndCard o))))
(forall ((o Operations))(not(and
(isTypePin o)(isReceiveCard o))))
(forall ((o Operations))(not(and
(isRequistCash o)(isReceiveCashAndCard o))))
(forall ((o Operations))(not(and
(isRequistCash o)(isReceiveCard o))))
(forall ((o Operations))(not(and
(isReceiveCashAndCard o)(isReceiveCard o))))

;; the abstract property of signature <ATM_Status>
(forall ((a ATM_Status))(or(isWaitingCard a)
(isWaitingPin a)(isWaitingMoney a)
(isWaitingReceiveCashAndCard a)
(isWaitingReceiveCard a) (isUpdate a)))

;; the extends property of the signatures <WaitingCard>,
;<WaitingPin>, <WaitingMoney>,<WaitingReceiveCashAndCard>,
;<WaitingReceiveCard>,<Update> to the signature<ATM_Status>
(forall ((a ATM_Status))(not(and
(isWaitingCard a)(isWaitingPin a))))
(forall ((a ATM_Status))(not(and
(isWaitingCard a)(isWaitingMoney a))))
(forall ((a ATM_Status))(not(and
(isWaitingCard a)(isWaitingReceiveCashAndCard a))))
(forall ((a ATM_Status))(not(and
(isWaitingCard a)(isWaitingReceiveCard a))))
(forall ((a ATM_Status))(not(and
(isWaitingCard a)(isUpdate a))))
(forall ((a ATM_Status))(not(and
(isWaitingPin a)(isWaitingMoney a))))
(forall ((a ATM_Status))(not(and
(isWaitingPin a)(isWaitingReceiveCashAndCard a))))
(forall ((a ATM_Status))(not(and
(isWaitingPin a)(isWaitingReceiveCard a))))
(forall ((a ATM_Status))(not(and
(isWaitingPin a)(isUpdate a))))
(forall ((a ATM_Status))(not(and
(isWaitingMoney a)(isWaitingReceiveCashAndCard a))))
```

```
(forall ((a ATM_Status))(not(and
(isWaitingMoney a)(isWaitingReceiveCard a))))
(forall ((a ATM_Status))(not(and
(isWaitingMoney a)(isUpdate a))))
(forall ((a ATM_Status))(not(and
(isWaitingReceiveCashAndCard a)(isWaitingReceiveCard a))))
(forall ((a ATM_Status))(not(and
(isWaitingReceiveCashAndCard a)(isUpdate a))))
(forall ((a ATM_Status))(not(and
(isWaitingReceiveCard a)(isUpdate a))))
```

Listing B.8: Relations

```
;; relations of the signature <ATM>
;;inCard : lone cards ,
(forall ((this ATM))(and(forall ((c1 Card))(=>
(inCard this c1)(cards this c1)))
(forall ((c3 Card)(c2 Card)(=>(and(inCard this c2)
(inCard this c3)) (= c2 c3)))))
;; pin : cards -> one Int ,
(forall ((this ATM))(and (forall ((c1 Card) (i Int))
(=> (pin this c1 i)(cards this c1)))
(forall ((a1 Card))(=> (cards this a1)(and
(exists ((i1 Int))(pin this a1 i1))
(forall ((i3 Int)(i2 Int))(=>(and (pin this a1 i2)
(pin this a1 i3))(= i2 i3))))))))
;; balance: cards -> one Int,
(forall ((this ATM))(and (forall ((c1 Card)(i Int))(=>
(balance this c1 i)(cards this c1)))
(forall ((a1 Card)) (=>(cards this a1)(and
(exists ((i1 Int))(balance this a1 i1))
(forall ((i3 Int)(i2 Int)) (=> (and (balance this a1 i2)
(balance this a1 i3))(= i2 i3))))))))
;; money:cards -> lone Int,
(forall ((this ATM))(and (forall ((c1 Card)(i Int))(=>
(money this c1 i)(cards this c1)))(forall ((a1 Card)
(i3 Int)(i2 Int))(=>(and(money this a1 i2)(money this a1 i3))
(= i2  i3)))))))
;; atmStatuse:  one  ATM_Status,
(forall ((this ATM))(and(exists ((a1 ATM_Status))
(atmStatuse this a1))
(forall ((a3 ATM_Status) (a2 ATM_Status)) (=> (and
(atmStatuse this a2)(atmStatuse this a3))(= a2 a3)))))
;; op: lone Operations
(forall ((this ATM) (o2 Operations) (o1 Operations))(=>
(and(op this o1)(op this o2))(= o1 o2)))
```

Listing B.9: Facts

```
;; translation of the first fact
(forall ((atm1 ATM) (atm2 ATM))(and(forall ((c1 Card))
(=> (cards atm2 c1)(cards atm1 c1)))
(forall ((c2 Card))(=> (cards atm1 c2)(cards atm2 c2)))
(forall ((c3 Card)(i Int))(=>
(pin atm2 c3 i) (pin atm1 c3 i)))
(forall ((c4 Card)(i1 Int))(=>
(pin atm1 c4 i1)(pin atm2 c4 i1)))))
;; translation of the second fact
(forall ((atm1 ATM)(card Card)) (=> (cards atm1 card)
(and
;; card.(atm1.balance)>=0
(forall ((i Int))(=>(balance atm1 card i)(>=  i 0)))
;; card.(atm1.pin)> 0
(forall ((i1 Int))(=>(pin atm1 card i1)(> i1 0)))))))
```

Listing B.10: Predicate and Assertion

```
;; "inlined" translation of the predicate <ATMTransaction>.
;Inlined means: without explicit declartion

;; translation of the assertion <prop1>
(forall ((atm1 ATM)(atm2 ATM)(atm3 ATM)(atm4 ATM)
(atm5 ATM)(atm6 ATM)(pn Int)(mon Int)(crd Card))
(=> (and
;; translation of: mon <= crd.(atm1.balance)  and  0 < mon
;; This assumption is mandatory of the correction of the assertion
<prop1>
;; If it is out-commented the SMT solver should return
;; a counterexample (i.e., sat),
;; otherwise it should prove the assertion correct (i.e., unsat)
(forall ((i Int))(=>(balance atm1 crd i)
(<= mon i)))(< 0 mon)
;; translation of: (atm1.atmStatuse)= WaitingCard
(forall ((a1 ATM_Status))(=>
(atmStatuse atm1 a1)(isWaitingCard a1)))
(forall ((w ATM_Status))(=>
(isWaitingCard w)(atmStatuse atm1 w)))
;; translation of: crd in atm1.cards
(cards atm1 crd)
;; translation of: no atm1.inCard and no atm1.op
(forall ((c2 Card))(not(inCard atm1 c2)))
(forall ((o Operations)) (not (op atm1 o)))
;; translation of: no atm1.money
(forall ((m Int)(c9 Card))(not (money atm1 c9 m)))
;; translation of: no atm2.money
(forall ((m Int(c9 Card)))(not (money atm2 c9 m)))
;; translation of: atm2.op= EnterCard
(forall ((o1 Operations))(=>
(op atm2 o1)(isEnterCard o1)))
(forall ((e Operations))(=>
(isEnterCard e)(op atm2 e)))
;; translation of: atm2.inCard = crd
(forall ((c3 Card)) (=> (inCard atm2 c3)
(= crd c3)))(inCard atm2 crd)
;; translation of: atm2.balance = atm1.balance
(forall ((c5 Card)(i1 Int))(=>
(balance atm2 c5 i1)(balance atm1 c5 i1)))
(forall ((c6 Card)(i2 Int))(=>
(balance atm1 c6 i2)(balance atm2 c6 i2)))
;; translation of: (atm2.atmStatuse) = WaitingPin
(forall ((a15 ATM_Status))(=>
(atmStatuse atm2 a15)(isWaitingPin a15)))
(forall ((w1 ATM_Status))(=>
(isWaitingPin w1)(atmStatuse atm2 w1)))
;; translation of: atm3.op = TypePin
(forall ((o2 Operations))(=>(op atm3 o2)(isTypePin o2)))
(forall ((t Operations))(=>(isTypePin t)(op atm3 t)))
;; translation of: atm3.inCard = atm2.inCard
(forall ((c7 Card))(=>(inCard atm3 c7)(inCard atm2 c7)))
(forall ((c8 Card))(=>(inCard atm2 c8)(inCard atm3 c8)))
```

```
16  ;; translation of: atm3.inCard.(atm3.pin) = pn
    (forall ((i3 Int))(=>(exists ((c9 Card))
    (and(inCard atm3 c9)(pin atm3 c9 i3)))(= pn i3)))
    (exists ((c10 Card))(and(inCard atm3 c10)(pin atm3 c10 pn)))
17  ;; translation of: atm3.balance = atm2.balance
    (forall ((c11 Card)(i5 Int))(=>
    (balance atm3 c11 i5)(balance atm2 c11 i5)))
    (forall ((c12 Card)(i6 Int))(=>
    (balance atm2 c12 i6)(balance atm3 c12 i6)))
18  ;; translation of: (atm3.atmStatuse) = WaitingMoney
    (forall ((a32 ATM_Status))(=>
    (atmStatuse atm3 a32)(isWaitingMoney a32)))
    (forall ((w2 ATM_Status))(=>
    (isWaitingMoney w2)(atmStatuse atm3 w2)))
19   ;; translation of: no atm3.money
    (forall ((m Int)(c9 Card))(not(money atm3 c9 m)))
20   ;; translation of: atm4.op= RequistCash
    (forall ((o3 Operations))(=>(op atm4 o3)(isRequistCash o3)))
    (forall ((r Operations))(=>(isRequistCash r)(op atm4 r)))
21  ;; translation of: atm4.balance = atm3.balance
    (forall ((c13 Card)(i7 Int))(=>(balance atm4 c13 i7)
    (balance atm3 c13 i7)))(forall ((c14 Card)(i8 Int))(=>
    (balance atm3 c14 i8)(balance atm4 c14 i8)))
22  ;; translation of: atm4.inCard = atm3.inCard
    (forall ((c15 Card))(=>(inCard atm4 c15)(inCard atm3 c15)))
    (forall ((c16 Card))(=>(inCard atm3 c16)(inCard atm4 c16)))
23  ;; translation of: atm4.inCard.(atm4.money) = mon
    (forall ((i3 Int))(=>(forall ((c9 Card))
    (and(inCard atm4 c9)(money atm4 c9 i3)))(= mon i3)))
    (forall ((c10 Card))(and(inCard atm4 c10)(money atm4 c10 mon)))
24  ;; translation of: atm5.inCard = atm4.inCard
    (forall ((c17 Card))(=> (inCard atm5 c17)(inCard atm4 c17)))
    (forall ((c18 Card))(=>(inCard atm4 c18)(inCard atm5 c18)))
25  ;;translation of:( mon<=atm4.inCard.(atm4.balance)and 0<mon)
    ; and
    ;(atm5.inCard.(atm5.balance)=atm4.inCard.(atm4.balance)- mon
    ;and atm5.op= ReceiveCashAndCard and (atm4.atmStatuse) =
    ;WaitingReceiveCashAndCard)
    (or (and (forall ((i9 Int))(=>
    (exists ((c19 Card))(and(inCard atm4 c19)
    (balance atm4 c19 i9)))(<= mon i9)))
    (< 0 mon)(forall ((i10 Int)(i11 Int))
    (=(and (exists ((c20 Card))(and(inCard atm5 c20)
    (balance atm5 c20 i10)))(exists ((c21 Card))
    (and (inCard atm4 c21)(balance atm4 c21 i11))))
    (= i10 (- i11 mon))))
    (forall ((o4 Operations))(=>
    (op atm5 o4)(isReceiveCashAndCard o4)))
    (forall ((r1 Operations)) (=>
    (isReceiveCashAndCard r1)(op atm5 r1)))
    (forall ( (a57 ATM_Status))
    (=> (atmStatuse atm4 a57)(isWaitingReceiveCashAndCard a57)))
    (forall ((w3 ATM_Status))(=>
    (isWaitingReceiveCashAndCard w3)(atmStatuse atm4 w3))))
26  ;;translation of:(mon>atm4.inCard.(atm4.balance) or mon<0)
    ;and
    ;;;(atm5.inCard.(atm5.balance) = atm4.inCard.(atm4.balance) and
    ;atm5.op= ReceiveCard and(atm4.atmStatuse)=WaitingReceiveCard)
    (and(or (forall ((i12 Int))(=> (exists ((c22 Card))(and
    (inCard atm4 c22)(balance atm4 c22 i12)))
    (< i12 mon)))(< mon 0))
    (forall ((i13 Int))(=> (exists ((c23 Card))
    (and (inCard atm5 c23)(balance atm5 c23 i13)))
    (exists ((c24 Card))(and (inCard atm4 c24)
    (balance atm4 c24 i13)))))
    (forall ((i14 Int)) (=> (exists ((c25 Card))(and
    (inCard atm4 c25)(balance atm4 c25 i14)))
    (exists ((c26 Card))(and(inCard atm5 c26)
    (balance atm5 c26 i14)))))
    (forall ((o5 Operations))(=>
    (op atm5 o5)(isReceiveCard o5)))
    (forall ((r2 Operations))(=>
    (isReceiveCard r2)(op atm5 r2)))
    (forall ((a72 ATM_Status))(=>
    (atmStatuse atm4 a72)(isWaitingReceiveCard a72)))
    (forall ((w4 ATM_Status)) (=>
    (isWaitingReceiveCard w4)(atmStatuse atm4 w4)))))
    ;; translation of: (atm5.atmStatuse) =  Update
    (forall ((a75 ATM_Status)) (=>
    (atmStatuse atm5 a75)(isUpdate a75)))
    (forall ((u ATM_Status))(=>
    (isUpdate u)(atmStatuse atm5 u)))
    ;; translation of: no atm6.inCard and (atm6.atmStatuse) =
    ;;WaitingCard and no atm6.op
    (forall ((c27 Card))(not(inCard atm6 c27)))
    (forall ((a79 ATM_Status))(=>
    (atmStatuse atm6 a79)(isWaitingCard a79)))
    (forall ((w5 ATM_Status)) (=>
    (isWaitingCard w5)(atmStatuse atm6 w5)))
    (forall ((o6 Operations))(not(op atm6 o6))))
    ;; translation of:crd.(atm5.balance)=crd.(atm1.balance)- mon
    (forall ((i15 Int) (i16 Int))
    (=> (and (balance atm5 crd i15)(balance atm1 crd i16))
    (= i15 (- i16 mon)))
    ))))))

```

Listing B.11: Commands

```
(check-sat)
;(get-model)
(exit)
```

Appendix C

Transmitting Data Over Single Channel Model Using Alloy (Second Protocol) in Secure/ Insecure Scope

Listing C.1: Signatures

```
sig Time{}
sig Channel{}
sig ISP{}
sig Data {}
abstract sig ConnectionStatus{}
one sig Connection_And_Exchanging_Data extends ConnectionStatus{}
sig MAC{}
sig Communication_Status {
```

Listing C.2: Relations Declaration

```
serviceProvider:  set ISP,
visitors:set MAC,
client: one MAC,
server: one MAC,
mitmIntercepts: lone MAC - server,
connection: MAC -> lone serviceProvider  ,
opens :MAC->  serviceProvider  ->   Channel,
status:MAC -> one ConnectionStatus,
sends: MAC ->lone (Data -> Time)-> Channel,
receives: MAC ->lone (Data -> Time)-> Channel}
```

Listing C.3: Facts

```
:FIRST FACT
fact {
all s: Communication_Status |
(s.mitmIntercepts !in s.visitors or
(s.mitmIntercepts in s.visitors and s.client !in s.mitmIntercepts))
implies (all t, t': Time, d, d': Data,ch:Channel |
s.client.(s.sends) =d ->t->ch and
s.server.(s.receives) =d' ->t'->ch
implies  t= t' )}
SECOND FACT
fact {
all t,t':Time,d,d':Data,s:Communication_Status,ch:Channel |
(s.client.(s.sends) =d ->t->ch and
s.server.(s.receives) =d' ->t'->ch and t =t')
```

```
 implies ((d = d') and no s.mitmIntercepts.(s.sends) and
no s.mitmIntercepts.(s.receives))}
;THIRD FACT
fact {
all s: Communication_Status | s.client != s.server}
```

Listing C.4: Predecates

```
pred  SingleChannel [t,t':Time, d,d':Data,isp,isp':ISP,
status1:Communication_Status ,ch:Channel]{
status1.client.(status1.status) =Connection_And_Exchanging_Data and
status1.server.(status1.status) =Connection_And_Exchanging_Data and
status1.client in status1.visitors and
status1.server in status1.visitors and
status1.client.(status1.connection)= isp  and
status1.server.(status1.connection)= isp' and
status1.client.(status1.opens)=
status1.client.(status1.connection)->ch and
status1.server.(status1.opens)=
status1.server.(status1.connection)->ch and
status1.client.(status1.sends)=d->t->ch and
status1.server.(status1.receives)=d'->t'->ch and
no status1.client.(status1.receives)  and
no status1.server.(status1.sends)  and
// No CE secure scope (no Mitm at all)
//status1.mitmIntercepts !in status1.visitors
//or
//MITM INTERCEPTION (NO CE)  Ststic and dynamic mitm
(((status1.mitmIntercepts in status1.visitors) and
(status1.client ) !in status1.mitmIntercepts))}
```

Listing C.5: Assertion

```
assert DataSecure {
all  isp,isp' :ISP, d,d':Data, t,t':Time,ch:Channel,
status1: Communication_Status |
( SingleChannel [t,t',d,d',isp,isp',status1,ch] )
implies (d=d')}
```

Listing C.6: Commands

```
check DataSecure  for 3  MAC, 2 Time, 1 ISP, 2 Data,
1 ConnectionStatus, 1 Channel, 1 Communication_Status
run SingleChannel  for  3 MAC, 2 Time, 1 ISP, 2 Data,
1 ConnectionStatus, 1 Channel, 1 Communication_Status
```

Appendix D

Single Channel Proving Using Z3 Theorem Prover

Listing D.1: Sorts

```
(declare-sort Time)
(declare-sort Channel)
(declare-sort ISP)
(declare-sort Data)
(declare-sort ConnectionStatus)
(declare-sort MAC)
(declare-sort CommunicationStatus)
```

Listing D.2: Functions

```
(declare-fun serviceProvider (CommunicationStatus ISP) Bool)
(declare-fun visitors (CommunicationStatus MAC) Bool)
(declare-fun client1 (CommunicationStatus MAC ) Bool)
(declare-fun server (CommunicationStatus MAC) Bool)
(declare-fun mitmIntercepts (CommunicationStatus MAC) Bool)
(declare-fun connection (CommunicationStatus MAC ISP) Bool)
(declare-fun opens (CommunicationStatus MAC ISP Channel) Bool)
(declare-fun status (CommunicationStatus MAC ConnectionStatus) Bool)
(declare-fun sends (CommunicationStatus MAC Data Time Channel) Bool)
(declare-fun receives (CommunicationStatus MAC Data Time Channel)
Bool)
(declare-fun isConnection_And_Exchanging_Data (ConnectionStatus)
Bool)
```

Listing D.3: Some Property

```
;; the some property of the signatures
;Connection_And_Exchanging_Data
(declare-fun |oneOf_Connection_And_Exchanging_Data| ()
ConnectionStatus)
```

Listing D.4: Negation Of The Implication Of The Assertion

```
(assert
(not
(=>
(and
```

Listing D.5: Return Types Of The "oneOf" Functions/Constants

```
(isConnection_And_Exchanging_Data |
oneOf_Connection_And_Exchanging_Data|)
```

Listing D.6: The Lone Property Of The Signatures

```
 ;; the lone property of the signatures:
;Connection_And_Exchanging_Data
(forall ((o1 ConnectionStatus)(o2 ConnectionStatus))(=>
(and (isConnection_And_Exchanging_Data o1)
(isConnection_And_Exchanging_Data o2))(= o1 o2)))
```

Listing D.7: Relations

```
;client1: one MAC,
(forall ((cs CommunicationStatus))
(and (exists ((mac1 MAC))(client1 cs mac1))
(forall ((mac3 MAC)(mac2 MAC))
(=> (and(client1 cs mac2)(client1 cs mac3))
(= mac2 mac3)))))
;server: one MAC,
(forall ((cs CommunicationStatus))
(and(exists ((mac1 MAC))(server cs mac1))
(forall ((mac3 MAC)(mac2 MAC))
(=>(and(server cs mac2)(server cs mac3))(= mac2 mac3)))))
;mitmIntercepts: lone MAC - server,
(forall ((cs CommunicationStatus)(mac3 MAC)(mac2 MAC))(=>
(and(mitmIntercepts cs mac3)(mitmIntercepts cs mac2))(= mac2 mac3)))
(forall ((cs CommunicationStatus))(exists ((mac1 MAC))
(not(mitmIntercepts cs mac1))))
;connection: MAC -> lone serviceProvider ,
(forall ((cs CommunicationStatus)(mac MAC))(and
(forall ((isp11 ISP))(=>(connection cs mac isp11)
(serviceProvider cs isp11)))
(forall ((isp12 ISP)(isp13 ISP))(=> (and(connection cs mac isp12)
(connection cs mac isp13))(= isp12 isp13)))))
;status:MAC -> one ConnectionStatus,
(forall ((cs CommunicationStatus)(mac MAC))
(and (exists ((co ConnectionStatus))(status cs mac co))
(forall ((a3 ConnectionStatus)(a2 ConnectionStatus))
(=> (and (status cs mac a2)(status cs mac a3))(= a2 a3)))))
;opens :MAC-> serviceProvider -> Channel,
(forall ((cs CommunicationStatus)(mac MAC) (isp ISP)
(ch Channel))(=>(opens cs mac isp ch)(serviceProvider cs isp)))
;sends: MAC ->lone (Data -> Time)-> Channel,
(forall ((cs CommunicationStatus)(mac MAC)(ch Channel)
(d1 Data)(d2 Data)(t Time)(t1 Time))
(=>(and(sends cs mac d1 t ch)(sends cs mac d2 t1 ch))
(and(= d1 d2)(= t t1))))
;receives: MAC ->lone (Data -> Time)-> Channel,
(forall ((cs CommunicationStatus)(mac MAC)(ch Channel)
(d1 Data)(d2 Data)(t Time)(t1 Time))
(=>(and(receives cs mac d1 t ch)
(receives cs mac d2 t1 ch))(and(= d1 d2)(= t t1))))
```

Listing D.8: Facts

```
;FIRST FACT
(forall ((status1 CommunicationStatus))(implies
(forall ((cmac MAC)(mmac MAC))(or(=>(mitmIntercepts status1 mmac)
(not(visitors status1 mmac)))
(and(=>(mitmIntercepts status1 mmac)(visitors status1 mmac))
(=>(client1 status1 cmac)
(not(mitmIntercepts status1 cmac))))))
(forall ((t Time)(t1 Time)(d Data)(d1 Data)(ch1 Channel))
(implies (and (forall ((d11 Data)(t11 Time)(ch11 Channel))
(=> (exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= d d11)(= t t11)(= ch1 ch11))))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 d t ch1)))
(forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= d1 d11)(= t1 t11) (= ch1 ch11))))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 d1 t1 ch1))))(= t t1)))))
;SECOND FACT
(forall ((status1 CommunicationStatus)(t Time)(t1 Time)
(d Data)(d1 Data)(ch1 Channel))(implies(and
(forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= d d11)(= t t11)(= ch1 ch11))))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 d t ch1)))
(forall ((d11 Data)(t11 Time)(ch11 Channel))
(=>(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= d1 d11)(= t1 t11)(= ch1 ch11))))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 d1 t1 ch1)))(= t t1))(= d d1)))
;THIRD FACT
(forall ((s CommunicationStatus))
(forall ((mac1 MAC) (mac2 MAC))(=> (and(client1 s mac1)
(server s mac2))not(= mac1  mac2)))))
```

Listing D.9: Predicate and Assertion

```
(forall ((isp ISP)(isp1 ISP)(d Data)(d1 Data)(t Time)(t1 Time)
(ch1 Channel)(status1 CommunicationStatus))(=>(and
;status1.client.(status1.status)=isConnection_And_Exchanging_Data
(=>(forall ((cs1 ConnectionStatus)(cmac MAC))
(=> (and (client1 status1 cmac)(status status1 cmac cs1))
(isConnection_And_Exchanging_Data cs1)))
(forall ((cs2 ConnectionStatus)(cmac MAC))
(=> (isConnection_And_Exchanging_Data cs2)
(and (client1 status1 cmac)(status status1 cmac cs2)))))
;status1.server.(status1.status)=isConnection_And_Exchanging_Data
(=>(forall ((cs1 ConnectionStatus)(smac MAC))
(=> (and (server status1 smac)(status status1 smac cs1))
(isConnection_And_Exchanging_Data cs1)))
(forall ((cs2 ConnectionStatus)(smac MAC))
(=> (isConnection_And_Exchanging_Data cs2)
(and (server status1 smac)(status status1 smac cs2)))))
;status1.client1 in status1.visitors
(forall ((cmac MAC))(=> (client1 status1 cmac)
(visitors status1 cmac)))
;status1.server in status1.visitors
(forall ((smac MAC))(=> (server status1 smac)
(visitors status1 smac)))
;no status1.client1.(status1.receives)
(forall ((cmac MAC))(and(not(and(client1 status1 cmac)
(receives status1 cmac d1 t1 ch1)))(not(and(client1 status1 cmac)
(receives status1 cmac d t1 ch1)))))
;no status1.server.(status1.sends)
(forall ((smac MAC))(and(not(and(server status1 smac)
(sends status1 smac d t ch1)))(not(and(server status1 smac)
(sends status1 smac d1 t ch1)))))
;status1.client1.(status1.connection)= isp
(forall ((isp11 ISP))(=>(=> (exists ((cmac MAC))
(and (client1 status1 cmac)(connection status1 cmac isp11)))
(= isp isp11))(exists ((cmac1 MAC))
(and (client1 status1 cmac1)(connection status1 cmac1 isp)))))
;status1.server.(status1.connection)= isp`
(forall ((isp11 ISP))(=>(=> (exists ((smac MAC))
(and (server status1 smac)(connection status1 smac isp11)))
(= isp1 isp11))(exists ((smac1 MAC))
(and (server status1 smac1)(connection status1 smac1 isp1)))))
;status1.client1.(status1.opens)=
;status1.client1.(status1.connection)->ch1
(forall ((isp11 ISP)(ch Channel))(=>(forall ((cmac MAC))(=>
(and (client1 status1 cmac)(connection status1 cmac isp11))
(and (client1 status1 cmac)(opens status1 cmac isp11 ch ))))
(and(= isp isp11)(= ch1 ch))))(forall ((cmac1 MAC))(=>
(and (client1 status1 cmac1)(connection status1 cmac1 isp))
(and (client1 status1 cmac1)(opens status1 cmac1 isp ch1))))
;status1.server.(status1.opens)=
;status1.server.(status1.connection)->ch1
(forall ((isp11 ISP)(ch Channel))(=>(forall ((smac MAC))(=>
(and (server status1 smac)(connection status1 smac isp11))
(and (server status1 smac)(opens status1 smac isp11 ch))))
(and(= isp1 isp11)(= ch1 ch))))(forall ((smac1 MAC))(=>
(and (server status1 smac1)(connection status1 smac1 isp1))
(and (server status1 smac1)(opens status1 smac1 isp1 ch1 ))))
;status1.client1.(status1.sends)=d->t->ch1
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>
(exists ((cmac MAC))(and (client1 status1 cmac)
(sends status1 cmac d11 t11 ch11)))
(and(= d d11)(= t t11)(= ch1 ch11))))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 d t ch1)))
;status1.server.(status1.receives)=d'->t'->ch1
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>
(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= d1 d11)(= t1 t11)(= ch1 ch11))))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 d1 t1 ch1)))
;status1.mitmIntercepts !in status1.visitors  or
;(((status1.mitmIntercepts in status1.visitors) and
;(status1.client ) !in status1.mitmIntercepts))
(forall ((cmac MAC)(mmac MAC))(or(=>(mitmIntercepts status1 mmac)
(not(visitors status1 mmac)))
(and(=>(mitmIntercepts status1 mmac)(visitors status1 mmac))
(=>(client1 status1 cmac)(not(mitmIntercepts status1 cmac))))))
;ASSERTION    (= d d1))))))
```

Listing D.10: Commands

```
(check-sat);(get-model)(exit)
```

Appendix E

Transmitting Data Over Multichannel Model Using Alloy

Listing E.1: Signatures

```
sig Time{}
sig Channel{}
sig ISP{}
sig Data{}
abstract sig ConnectionStatus{}
one sig  First_Communication_And_Exchanging_Indices ,
Second_Communication_And_Exchanging_Letters extends ConnectionStatus
{}
sig MAC{}
sig Communication_Status {
```

Listing E.2: Relation Declaration

```
visitors:set MAC,
client1:one MAC, //each visitor (client) has one MAC
client2:one MAC ,
server:one MAC,
interseptMacs:lone MAC - server ,  // no need to intercept server
serviceProvider:set ISP,
ispA:one ISP,
ispB:one ISP,
connection: MAC -> lone serviceProvider  ,
status:MAC -> one ConnectionStatus,
channel:set Channel,
ch1:one Channel,
ch2:one Channel,
sends:MAC ->lone (Data -> Time)-> Channel,
receives:MAC ->lone (Data -> Time)-> Channel}
```

Listing E.3: Facts

```
;FIRST FACT
fact {all s, s': Communication_Status |
/** In case the first channel has been intercepted */
((s.client1 in s.visitors and s.client2 !in s.visitors) and
(s'.client1 !in s'.visitors and s'.client2 in s'.visitors)
and ((s.client1) in s.interseptMacs  and (s'.client2) !in
s'.interseptMacs or (s.client1) !in s.interseptMacs  and
(s'.client2) in s'.interseptMacs or (s.client1) !in
s.interseptMacs  and (s'.client2) !in s'.interseptMacs ) implies
(all s, s': Communication_Status, t, t', t'', t''':Time,
indices, indices', letters, letters': Data  |
(s.client1.(s.sends) = indices->t->s.ch1 and
s.server.(s.receives) = indices'->t'->s.ch1 and
s'.client2.(s'.sends)=letters->t''->s'.ch2 and
s'.server.(s'.receives)=letters'->t'''->s'.ch2)
implies ((t''!= t''' and  t=t') and (s'.client2) in
s'.interseptMacs and (s.client1) !in s.interseptMacs or
(t''= t''' and  t!=t') and (s.client1) in s.interseptMacs and
(s'.client2) !in s'.interseptMacs or (t''= t''' and  t=t')
and (s.client1) !in s.interseptMacs  and (s'.client2) !in
s'.interseptMacs)) }
;SECOND FACT
fact { all s,s':Communication_Status,t,t',t'',t''': Time,indices,
indices',   letters,letters':Data | ((s.client1.(s.sends) =
indices->t->s.ch1
and s.server.(s.receives) = indices'->t'->s.ch1) and
(s'.client2.(s'.sends)= letters->t''->s'.ch2
and s'.server.(s'.receives)=letters'->t'''->s'.ch2) and
((t''!= t''' and t=t') or (t''= t''' and  t=t') or
(t''= t''' and t!=t')))
implies (letters=letters' and indices=indices')}
;THIRD FACT
fact {all s',s: Communication_Status | s.client1 != s.server
and s'.client2 != s.server and s.client1 != s'.client2
and s.client1=s'.client1 and s.server = s'.server}
```

Listing E.4: Predicates

```
pred MultiChannel [t,t',t'',t''':Time,indices,indices',
letters,letters':Data,isp:ISP,status1,status2:Communication_Status]
{status1.client1.(status1.status) =
First_Communication_And_Exchanging_Indices and
status1.server.(status1.status) =
First_Communication_And_Exchanging_Indices and
status1.server in status1.visitors and
status1.ispA in status1.serviceProvider and
status1.ispB !in status1.serviceProvider and
no status1.client2.(status1.connection)
status1.client1.(status1.connection) =status1.ispA  and
status1.server.(status1.connection)= isp and
status1.ch1 in status1.channel and
status1.ch2 !in status1.channel and
status1.client1.(status1.sends)= indices->t->status1.ch1 and
status1.client1.(status1.sends)!= letters->t->status1.ch1 and
status1.server.(status1.receives)=indices'->t'->status1.ch1 and
status1.server.(status1.receives)!=letters'->t'->status1.ch1 and
no status1.server.(status1.sends) and
no status1.client1.(status1.receives) and
no status1.client2.(status1.receives)  and
no status1.client2.(status1.sends)
status2.client2.(status2.status) =
Second_Communication_And_Exchanging_Letters and
status2.server.(status2.status) =
Second_Communication_And_Exchanging_Letters and
status2.server in status2.visitors and
status2.ispA !in status2.serviceProvider and
status2.ispB in status2.serviceProvider and
no status2.client1.(status2.connection)and
status2.client2.(status2.connection)=status2.ispB and
status2.server.(status2.connection)=
status1.server.(status1.connection) and
status2.ch1 !in status2.channel and
status2.ch2 in status2.channel and
status2.client2.(status2.sends)=letters->t''->status2.ch2 and
status2.client2.(status2.sends)!=indices->t''->status2.ch2 and
status2.server.(status2.receives)=letters'->t'''->status2.ch2 and
status2.server.(status2.receives)!=indices'->t'''->status2.ch2 and
no status2.server.(status2.sends) and
no status2.client2.(status2.receives) and
no status2.client1.(status2.sends) and
no status2.client1.(status2.receives) and
status1.client1 in status1.visitors and
status1.client2 !in status1.visitors and
status2.client1 !in status2.visitors and
status2.client2 in status2.visitors and(
;NO CE in case the first channel is intercepted OR
;in case the second channel is intercepted
;OR neither
//(status1.client1) in status1.interseptMacs and
(status2.client2) !in status2.interseptMacs
// or (status1.client1) !in status1.interseptMacs and
(status2.client2) in status2.interseptMacs
// or (status1.client1) !in status1.interseptMacs and
(status2.client2) !in status2.interseptMacs)}
```

Listing E.5: Assertion

```
assert DataSecure {
all isp :ISP,status1, status2:Communication_Status,
indices,indices',letters,letters':Data, t,t',t'',t''':Time |
MultiChannel [t,t',t'',t''', indices,indices',
letters,letters',isp ,status1,status2]
implies indices = indices' and letters = letters'}
```

Listing E.6: Commands

```
check DataSecure for 3 MAC, 4 Time, 2 ISP, 2 Channel,
4 Data, 1 ConnectionStatus, 2 Communication_Status
run MultiChannel for 3 MAC, 4 Time, 2 ISP, 2 Channel,
4 Data, 1 ConnectionStatus, 2 Communication_Status
```

Appendix F

Multichannel Proving Using Z3 SMT Solver

Listing F.1: Sorts

```
(declare-sort Time)
(declare-sort Channel)
(declare-sort ISP)
(declare-sort Data)
(declare-sort ConnectionStatus)
(declare-sort MAC)
(declare-sort CommunicationStatus)
```

Listing F.2: Functions

```
(declare-fun serviceProvider (CommunicationStatus ISP) Bool)
(declare-fun visitors (CommunicationStatus MAC) Bool)
(declare-fun client1 (CommunicationStatus MAC ) Bool)
(declare-fun client2 (CommunicationStatus MAC ) Bool)
(declare-fun server (CommunicationStatus MAC) Bool)
(declare-fun interseptMacs (CommunicationStatus MAC) Bool)
(declare-fun ispA (CommunicationStatus ISP) Bool)
(declare-fun ispB (CommunicationStatus ISP) Bool)
(declare-fun connection (CommunicationStatus MAC ISP) Bool)
(declare-fun status (CommunicationStatus MAC ConnectionStatus) Bool)
(declare-fun channel (CommunicationStatus Channel) Bool)
(declare-fun ch1 (CommunicationStatus Channel) Bool)
(declare-fun ch2 (CommunicationStatus Channel) Bool)
(declare-fun sends (CommunicationStatus MAC Data Time Channel) Bool)
(declare-fun receives (CommunicationStatus MAC Data Time Channel)
Bool)
(declare-fun isFirst_Communication_And_Exchanging_Indices
(ConnectionStatus) Bool)
(declare-fun isSecond_Communication_And_Exchanging_Letters
(ConnectionStatus) Bool)
```

Listing F.3: Some Property

```
;; the some property of the signature
;<First_Communication_And_Exchanging_Indices>
(declare-fun |oneOf_First_Communication_And_Exchanging_Indices|()
ConnectionStatus)
;; the some property of the signature
;<Second_Communication_And_Exchanging_Letters>
(declare-fun |oneOf_Second_Communication_And_Exchanging_Letters|()
ConnectionStatus)
```

Listing F.4: Negation Of The Implication Of The Assertion

```
(assert
(not
(=>
(and
```

Listing F.5: Return Types Of The "oneOf" Functions/Constants

```
(isFirst_Communication_And_Exchanging_Indices|
oneOf_First_Communication_And_Exchanging_Indices|)
(isSecond_Communication_And_Exchanging_Letters|
oneOf_Second_Communication_And_Exchanging_Letters|)
```

Listing F.6: The Lone Property Of The Signatures

```
;the lone property of the signature:
;<First_Communication_And_Exchanging_Indices>
(forall((f1 ConnectionStatus)(f2 ConnectionStatus))(=>(and
(isFirst_Communication_And_Exchanging_Indices f1)
(isFirst_Communication_And_Exchanging_Indices f2))(= f1 f2)))
;the lone property of the signature:
;<Second_Communication_And_Exchanging_Letters>
(forall ((s1 ConnectionStatus)(s2 ConnectionStatus))(=>(and
(isSecond_Communication_And_Exchanging_Letters s1)
(isSecond_Communication_And_Exchanging_Letters s2))(= s1 s2)))
```

Listing F.7: Abstract Property Of Signature ConnectionStatus

```
(forall ((co ConnectionStatus))(or
(isFirst_Communication_And_Exchanging_Indices co)
(Second_Communication_And_Exchanging_Letters co)))
```

Listing F.8: The Extends Property Of The Signatures

```
;signatures <First_Communication_And_Exchanging_Indices>,
;<Second_Communication_And_Exchanging_Letters>
(forall ((co ConnectionStatus))(not(and
(isFirst_Communication_And_Exchanging_Indices co)
(isSecond_Communication_And_Exchanging_Letters co))))
```

Listing F.9: Relations

```
;client1: one MAC,
(forall ((cs CommunicationStatus ))(and
(exists ((mac1 MAC))(client1 cs mac1))
(forall ((mac3 MAC)(mac2 MAC))(=>(and
(client1 cs mac2)(client1 cs mac3))(= mac2 mac3)))))
;client2: one MAC,
(forall ((cs CommunicationStatus ))(and
(exists ((mac1 MAC))(client2 cs mac1))
(forall ((mac3 MAC)(mac2 MAC))(=>(and
(client2 cs mac2)(client2 cs mac3))(= mac2 mac3)))))
;server: one MAC,
(forall ((cs CommunicationStatus))(and
(exists ((mac1 MAC))(server cs mac1))
(forall ((mac3 MAC)(mac2 MAC))(=>(and
(server cs mac2)(server cs mac3))(= mac2 mac3)))))
```

```
;mitmIntercepts: lone MAC - server,
(forall ((cs CommunicationStatus)(mac3 MAC)(mac2 MAC))
(=>(and (mitmIntercepts cs mac3)(mitmIntercepts cs mac2))
(= mac2 mac3)))
(forall ((cs CommunicationStatus))(exists ((mac1 MAC))
(not(mitmIntercepts cs mac1))))
;ispA:one ISP,
(forall ((cs CommunicationStatus))(and
(exists ((isp1 ISP))(ispA cs isp1))
(forall ((isp2 ISP)(isp3 ISP))(=>(and
(ispA cs isp2)(ispA cs isp3))(= isp2 isp3)))))
;ispB :one ISP,
(forall ((cs CommunicationStatus))(and
(exists ((isp1 ISP))(ispB cs isp1))
(forall ((isp2 ISP)(isp3 ISP))(=>(and
(ispB cs isp2)(ispB cs isp3))(= isp2 isp3)))))
;connection: MAC -> lone serviceProvider ,
(forall ((cs CommunicationStatus)(mac MAC))(and
(forall ((isp11 ISP))(=>(connection cs mac isp11)
(serviceProvider cs isp11)))
(forall ((isp12 ISP)(isp13 ISP))(=> (and(connection cs mac isp12)
(connection cs mac isp13))(= isp12 isp13)))))
;status:MAC -> one ConnectionStatus,
(forall ((cs CommunicationStatus)(mac MAC))
(and (exists ((co ConnectionStatus))(status cs mac co))
(forall ((a3 ConnectionStatus)(a2 ConnectionStatus))
(=> (and (status cs mac a2)(status cs mac a3))
(= a2 a3)))))
;ch1: one Channel,
(forall ((cs CommunicationStatus))(and(exists
((ch11 Channel))(ch1 cs ch11))
(forall ((ch22 Channel)(ch33 Channel))(=>
(and (ch1 cs ch22)(ch1 cs ch33))(= ch22 ch33)))))
;ch2: one Channel,
(forall ((cs CommunicationStatus))(and(exists
((ch11 Channel))(ch2 cs ch11))
(forall ((ch22 Channel)(ch33 Channel))(=>(and
(ch2 cs ch22)(ch2 cs ch33))(= ch22 ch33)))))
;sends: MAC ->lone (Data -> Time)-> Channel,
(forall ((cs CommunicationStatus)(mac MAC)(ch Channel)
(d1 Data)(d2 Data)(t Time)(t1 Time))
(=>(and(sends cs mac d1 t ch)(sends cs mac d2 t1 ch))
(and(= d1 d2)(= t t1))))
;receives: MAC ->lone (Data -> Time)-> Channel,
(forall ((cs CommunicationStatus)(mac MAC)(ch Channel)
(d1 Data)(d2 Data)(t Time)(t1 Time))
(=>(and(receives cs mac d1 t ch)
(receives cs mac d2 t1 ch))(and(= d1 d2)(= t t1))))
```

Listing F.10: Facts

```
;FIRST FACT
(forall ((status1 CommunicationStatus)
(status2 CommunicationStatus))(implies
(forall ((cmac1 MAC)(cmac2 MAC))(and
(and (=>(client1 status1 cmac1)(visitors status1 cmac1))
(=> (client2 status1 cmac2)(not(visitors status1 cmac2)))
(=> (client1 status2 cmac1)(not(visitors status2 cmac1)))
(=>(client2 status2 cmac2)(visitors status2 cmac2)))
```

```
(or (and (=>(client1 status1 cmac1)(interseptMacs status1 cmac1))
(=> (client2 status2 cmac2)
(not(interseptMacs status2 cmac2))))(and (=>(client1 status1 cmac1)
(not(interseptMacs status1 cmac1)))(=>
(client2 status2 cmac2)(interseptMacs status2 cmac2))
(and(=>(client1 status1 cmac1)
(not(interseptMacs status1 cmac1)))(=> (client2 status2 cmac2)
(not(interseptMacs status2 cmac2)))))))
(forall ((status1 CommunicationStatus)(status2 CommunicationStatus)
(t Time)(t1 Time)(t2 Time)(t3 Time)(indices Data)
(indices1 Data)(letters Data)(letters1 Data))
(implies (and (forall ((d11 Data)(t11 Time)
(ch11 Channel))(=>(=> (exists ((cmac MAC))(and
(client1 status1 cmac)(sends status1 cmac d11 t11 ch11)))
(and(= indices d11)(= t t11)(ch1 status1 ch11)))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 indices t ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= indices1 d11)(= t1 t11)(ch1 status1 ch11)))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 indices1 t1 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((cmac MAC))(and (client2 status2 cmac)
(sends status2 cmac d11 t11 ch11)))
(and(= letters d11)(= t2 t11)(ch2 status2 ch11)))
(exists ((cmac2 MAC))(and (client2 status2 cmac2)
(sends status2 cmac2 letters t2 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status2 smac)
(receives status2 smac d11 t11 ch11)))
(and(= letters1 d11)(= t3 t11)(ch2 status2 ch11)))
(exists ((smac2 MAC))(and (server status2 smac2)
(receives status2 smac2 letters1 t3 ch11))))))
(forall ((cmac1 MAC)(cmac2 MAC))
(or(and(and(= t2 t3)(not(= t t1)))(and (=>(client1 status1 cmac1)
(interseptMacs status1 cmac1))
(=> (client2 status2 cmac2)(not(interseptMacs status2 cmac2)))))
(and(and(not(= t2 t3))(= t t1))
(and (=>(client1 status1 cmac1)(not(interseptMacs status1 cmac1)))
(=>(client2 status2 cmac2)
(interseptMacs status2 cmac2))))(and(and(= t t1)(= t2 t3))
(and(=>(client1 status1 cmac1)
(not(interseptMacs status1 cmac1))) (=>(client2 status2 cmac2)
(not(interseptMacs status2 cmac2))))))))))
;SECOND FACT
(forall ((status1 CommunicationStatus)
(status2 CommunicationStatus)(t Time)(t1 Time)(t2 Time)(t3 Time)
(indices Data)(indices1 Data)(letters Data)(letters1 Data))
(implies (and(forall ((d11 Data)(t11 Time)
(ch11 Channel))(=>(=> (exists ((cmac MAC))(and
(client1 status1 cmac)(sends status1 cmac d11 t11 ch11)))
(and(= indices d11)(= t t11)(ch1 status1 ch11)))
(exists ((cmac1 MAC))(and (client1 status1 cmac1)
(sends status1 cmac1 indices t ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status1 smac)
(receives status1 smac d11 t11 ch11)))
(and(= indices1 d11)(= t1 t11)(ch1 status1 ch11)))
(exists ((smac1 MAC))(and (server status1 smac1)
(receives status1 smac1 indices1 t1 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((cmac MAC))(and (client2 status2 cmac)
(sends status2 cmac d11 t11 ch11)))
(and(= letters d11)(= t2 t11)(ch2 status2 ch11)))
(exists ((cmac2 MAC))(and (client2 status2 cmac2)
(sends status2 cmac2 letters t2 ch11)))))
(forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
(exists ((smac MAC))(and (server status2 smac)
(receives status2 smac d11 t11 ch11)))
(and(= letters1 d11)(= t3 t11)(ch2 status2 ch11)))
(exists ((smac2 MAC))(and (server status2 smac2)
(receives status2 smac2 letters1 t3 ch11)))))
(or (and(= t t1)(not(= t2 t3)))
(and(= t2 t3) (not(= t t1)))(and(= t2 t3)(= t t1))))
(and(= indices indices1)(= letters letters1))))
;THIRD FACT
(forall ((s CommunicationStatus)(s1 CommunicationStatus)
( mac1 MAC) (mac2 MAC)(mac3 MAC)(mac4 MAC))(and(=>
(and(client1 s mac1)(server s mac2))(not(= mac1  mac2)))
( =>(and(client2 s mac1)(server s mac2))(not(= mac1  mac2)))(=>
(and(client1 s mac1)(client2 s1 mac2))(not(= mac1  mac2)))
( => (and(client1 s mac1)(client1 s1 mac2))(= mac1 mac2
(=> (and(server s mac3)(server s1 mac4))(= mac3 mac4))
);ends of and )
```

Listing F.11: Predicate and Assertion

```
(forall ((status1 CommunicationStatus)(isp ISP)
(status2 CommunicationStatus)(t Time)(t1 Time)
(t2 Time)(t3 Time)(indices Data)(indices1 Data)
(letters Data)(letters1 Data))(=>(and
;status1.client1.(status1.status) =
;isFirst_Communication_And_Exchanging_Indices
(=>(forall ((cs1 ConnectionStatus)(cmac MAC))(=>
(and (client1 status1 cmac)(status status1 cmac cs1))
(isFirst_Communication_And_Exchanging_Indices  cs1))))
(forall ((cs2 ConnectionStatus)(cmac MAC))
(=>(isFirst_Communication_And_Exchanging_Indices cs2)(and
(client1 status1 cmac)(status status1 cmac cs2)))))
;status1.server.(status1.status) =
;First_Communication_And_Exchanging_Indices
(=> (forall ((cs1 ConnectionStatus)(smac MAC))(=>
(and (server status1 smac)(status status1 smac cs1))
(isFirst_Communication_And_Exchanging_Indices  cs1))))
(forall ((cs2 ConnectionStatus)(smac MAC))
(=> (isFirst_Communication_And_Exchanging_Indices cs2)(and
(server status1 smac)(status status1 smac cs2)))))
;status1.server in status1.visitors
(forall ((smac MAC))(=> (server status1 smac)
(visitors status1 smac)))
;status1.client1.(status1.connection) =status1.ispA
(forall ((isp11 ISP))(=> (=>
(exists ((cmac MAC))(and (client1 status1 cmac)
(connection status1 cmac isp11)))
(ispA status1 isp11))(exists ((cmac1 MAC))(and
   (client1 status1 cmac1)(connection status1 cmac1 isp11)))))
 6 ;status1.ispA in status1.serviceProvider
   (forall ((ispa ISP))(=> (ispA status1 ispa)
   (serviceProvider status1 ispa)))
 7 ;status1.ispB !in status1.serviceProvider
   (forall ((ispb ISP))(=>(not (ispB status1 ispb))
   (serviceProvider status1 ispb)))
 8 ;status1.server.(status1.connection)= isp
   (forall ((isp11 ISP))(=>(=> (exists ((smac MAC))
   (and (server status1 smac)(connection status1 smac isp11)))
   (= isp isp11))(exists ((smac1 MAC))
   (and (server status1 smac1)(connection status1 smac1 isp)))))
 9 ;status1.ch1 in status1.channel
   (forall ((ch Channel))(=> (ch1 status1 ch)
   (channel status1 ch)))
10 ;status1.ch2 !in status1.channel
   (forall ((ch Channel))(=>(not (ch2 status1 ch))
   (channel status1 ch)))
11 ;no status1.client2.(status1.connection)
   (forall ((cmac MAC))(not(and(client2 status1 cmac)
   (connection status1 cmac isp))))
12 ;status1.client1.(status1.sends)= indices->t->status1.ch1
   (forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
   (exists ((cmac MAC))(and (client1 status1 cmac)
   (sends status1 cmac d11 t11 ch11)))
   (and(= indices d11)(= t t11)(ch1 status1 ch11)))
   (exists ((cmac1 MAC))(and (client1 status1 cmac1)
   (sends status1 cmac1 indices t ch11)))))
13 ;status1.client1.(status1.sends)!= letters->t->status1.ch1
   (forall ((d11 Data)(t11 Time)(ch11 Channel))
   (=> (=>(exists ((cmac MAC))(and (client1 status1 cmac)
   (sends status1 cmac d11 t11 ch11)))
   (and(= letters d11) (= t t11)(ch1 status1 ch11)))
   (exists ((cmac1 MAC))(and (client1 status1 cmac1)
   (not(sends status1 cmac1 letters t ch11))))))
14 ;status1.server.(status1.receives)!=letters'->t'->status1.ch1
   (forall ((d11 Data)(t11 Time)(ch11 Channel))
   (=> (=>(exists ((smac MAC))(and (server status1 smac)
   (receives status1 smac d11 t11 ch11)))
   (and(= letters1 d11)(= t1 t11)(ch1 status1 ch11)))
   (exists ((smac1 MAC))(and (server status1 smac1)
   (not(receives status1 smac1 letters1 t1 ch11))))))
15 ;status1.server.(status1.receives)=indices'->t'->status1.ch1
   (forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
   (exists ((smac MAC))(and (server status1 smac)
   (receives status1 smac d11 t11 ch11)))
   (and(= indices1 d11)(= t1 t11)(ch1 status1 ch11)))
   (exists ((smac1 MAC))(and (server status1 smac1)
   (receives status1 smac1 indices1 t1 ch11)))))
16 ;no status1.server.(status1.sends)
   (forall ((ch Channel)(smac MAC))
   (and(not(and(server status1 smac)
   (sends status1 smac indices t ch)))
   (not(and(server status1 smac)
   (sends status1 smac indices1 t ch)))
   (not(and(server status1 smac)
   (sends status1 smac letters t ch)))
   (not(and(server status1 smac)
   (sends status1 smac letters1 t ch)))))
17 ;no status1.client1.(status1.receives)
   (forall ((ch Channel)(cmac MAC))
   (and(not(and(client1 status1 cmac)
   (receives status1 cmac indices t1 ch)))
   (not(and(client1 status1 cmac)
   (receives status1 cmac indices1 t1 ch)))
   (not(and(client1 status1 cmac)
   (receives status1 cmac letters t1 ch)))
   (not(and(client1 status1 cmac)
   (receives status1 cmac letters1 t1 ch)))))
18 ;no status1.client2.(status1.receives)
   (forall ((ch Channel)(cmac MAC))
   (and(not(and(client2 status1 cmac)
   (receives status1 cmac indices t1 ch)))
   (not(and(client2 status1 cmac)
   (receives status1 cmac indices1 t1 ch)))
   (not(and(client2 status1 cmac)
   (receives status1 cmac letters t1 ch)))
   (not(and(client2 status1 cmac)
   (receives status1 cmac letters1 t1 ch)))))
19 ;no status1.client2.(status1.sends)
   (forall ((ch Channel)(cmac MAC))
   (and(not(and(client2 status1 cmac)
   (sends status1 cmac indices t ch)))
   (not(and(client2 status1 cmac)
   (sends status1 cmac indices1 t ch)))
   (not(and(client2 status1 cmac)
   (sends status1 cmac letters t ch)))
   (not(and(client2 status1 cmac)
   (sends status1 cmac letters1 t ch)))))
20 ;status2.client2.(status2.status) =
   ;Second_Communication_And_Exchanging_Letters
   (=>(forall ((cs1 ConnectionStatus)(cmac MAC))(=>
   (and(client2 status2 cmac)(status status2 cmac cs1))
   (isSecond_Communication_And_Exchanging_Letters cs1))))
   (forall ((cs2 ConnectionStatus)(cmac MAC))
   (=> (isSecond_Communication_And_Exchanging_Letters  cs2)(and
   (client2 status2 cmac)(status status2 cmac cs2))))))
21 ;status2.server.(status2.status) =
   ;Second_Communication_And_Exchanging_Letters
   (=> (forall((cs1 ConnectionStatus)(smac MAC))
   (=> (and (server status2 smac)(status status2 smac cs1))
   (isSecond_Communication_And_Exchanging_Letters  cs1))))
   (forall ((cs2 ConnectionStatus)(smac MAC))
   (=> (isSecond_Communication_And_Exchanging_Letters cs2)
   (and (server status2 smac)(status status2 smac cs2))))))
22 ;status2.server in status2.visitors
   (forall ((smac MAC))(=> (server status2 smac)
   (visitors status2 smac)))
23 ;status2.ispA !in status2.serviceProvider
   (forall ((ispa ISP))(=>(not(ispA status2 ispa))
   (serviceProvider status2 ispa)))
24 ;status2.ispB in status2.serviceProvider
   (forall ((ispb ISP))(=>(ispB status2 ispb)
   (serviceProvider status2 ispb)))
25 ;status2.ch1 !in status2.channel
   (forall ((ch22 Channel))(=>(not(ch1 status2 ch22))
   (channel status2 ch22)))
```

```
26  ;status2.ch2 in status2.channel
    (forall ((ch22 Channel)) (=>(ch2 status2 ch22)
    (channel status2 ch22)))
27  ;no status2.client1.(status2.connection)
    (forall ((cmac MAC))(not(and(client1 status2 cmac)
    (connection status2 cmac isp))))
28  ;status2.client2.(status2.connection)=status2.ispB
    (forall ((isp11 ISP))(=>(=>
    (exists ((cmac MAC))(and (client2 status2 cmac)
    (connection status2 cmac isp11)))(ispB status2 isp11))
    (exists ((cmac2 MAC))(and (client2 status2 cmac2)
    (connection status2 cmac2 isp11)))))
29  ;status2.server.(status2.connection)=
    ;status1.server.(status1.connection)
    (forall ((smac MAC)(isp11 ISP))(=> (and (server status2 smac)
    (connection status2 smac isp11))(and (server status1 smac)
    (connection status1 smac isp11))))
    (forall ((smac1 MAC)(isp12 ISP))(=> (and
    (server status1 smac1)(connection status1 smac1 isp12))(and
    (server status2 smac1)(connection status2 smac1 isp12))))
30  ;status2.client2.(status2.sends)=letters->t''->status2.ch2
    (forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
    (exists ((cmac MAC))(and (client2 status2 cmac)
    (sends status2 cmac d11 t11 ch11)))
    (and(= letters d11)(= t2 t11)(ch2 status2 ch11)))
    (exists ((cmac2 MAC))(and (client2 status2 cmac2)
    (sends status2 cmac2 letters t2 ch11)))))
31  ;status2.client2.(status2.sends)!=indices->t''->status2.ch2
    (forall ((d11 Data)(t11 Time)(ch11 Channel))
   (=> (=>(exists ((cmac MAC))(and (client2 status2 cmac)
    (sends status2 cmac d11 t11 ch11)))
    (and(= indices d11)(= t2 t11)(ch2 status2 ch11)))
    (exists ((cmac2 MAC))(and (client2 status2 cmac2)
    (not(sends status2 cmac2 indices t2 ch11))))))
32  ;status2.server.(status2.receives)=letters'->t'''->status2.ch2
    (forall ((d11 Data)(t11 Time)(ch11 Channel))(=>(=>
    (exists ((smac MAC))(and (server status2 smac)
    (receives status2 smac d11 t11 ch11)))
    (and(= letters1 d11)(= t3 t11)(ch2 status2 ch11)))
    (exists ((smac2 MAC))(and (server status2 smac2)
    (receives status2 smac2 letters1 t3 ch11))))))
33  ;status2.server.(status2.receives)!=indices'->t'''->status2.ch2
    (forall ((d11 Data)(t11 Time)(ch11 Channel))(=> (=>
    (exists ((smac MAC))(and (server status2 smac)
    (receives status2 smac d11 t11 ch11)))
    (and(= indices1 d11)(= t3 t11)(ch2 status2 ch11)))
    (exists ((smac2 MAC))(and (server status2 smac2)
    (not(receives status2 smac2 indices1 t3 ch11))))))
34  ;no status2.server.(status2.sends)
    (forall ((ch Channel)(smac MAC))(and(not(and
    (server status2 smac)(sends status2 smac indices t2 ch)))
    (not(and(server status2 smac)
    (sends status2 smac indices1 t2 ch)))
    (not(and(server status2 smac)
    (sends status2 smac letters t2 ch)))(not(and
    (server status2 smac)(sends status2 smac letters1 t2 ch)))))
35  ;no status2.client1.(status2.receives)
    (forall ((ch Channel)(cmac MAC))(and(not(and
    (client1 status2 cmac)(receives status2 cmac indices t3 ch)))
    (not(and(client1 status2 cmac)
    (receives status2 cmac indices1 t3 ch)))
    (not(and(client1 status2 cmac)
    (receives status2 cmac letters t3 ch)))(not(and
    (client1 status2 cmac)(receives status2 cmac letters1 t3 ch)))))
    ;no status2.client2.(status2.receives)
    (forall ((ch Channel)(cmac MAC))(and(not(and
    (client2 status2 cmac)(receives status2 cmac indices t3 ch)))
    (not(and(client2 status2 cmac)
    (receives status2 cmac indices1 t3 ch)))(not(and
    (client2 status2 cmac)(receives status2 cmac letters t3 ch)))
    (not(and(client2 status2 cmac)
    (receives status2 cmac letters1 t3 ch)))))
    ;no status2.client1.(status2.sends)
    (forall ((ch Channel)(cmac MAC))(and(not(and
    (client1 status2 cmac)(sends status2 cmac indices t2 ch)))
    (not(and(client1 status2 cmac)
    (sends status2 cmac indices1 t2 ch)))(not(and
    (client1 status2 cmac)(sends status2 cmac letters t2 ch)))
    (not(and(client1 status2 cmac)
    (sends status2 cmac letters1 t2 ch)))))
    ;status1.client1 in status1.visitors and status1.client2 !in
    ; status1.visitors and
    ;status2.client1 !in status2.visitors and status2.client2 in
    ; status2.visitors  and (
    ;(status1.client1) in status1.interseptMacs  and (status2.client2)
    ;!in status2.interseptMacs
    ; or (status1.client1) !in status1.interseptMacs  and
    ;(status2.client2) in status2.interseptMacs
    ; or (status1.client1) !in status1.interseptMacs  and
    ;(status2.client2) !in status2.interseptMacs)
    (forall ((cmac1 MAC)(cmac2 MAC))(and (and (=>
    (client1 status1 cmac1)(visitors status1 cmac1))
    (=> (client2 status1 cmac2)(not(visitors status1 cmac2)))
    (=> (client1 status2 cmac1)(not(visitors status2 cmac1)))
    (=>(client2 status2 cmac2)(visitors status2 cmac2)))
    (or (and (=>(client1 status1 cmac1)(interseptMacs status1 cmac1))
    (=> (client2 status2 cmac2)
    (not(interseptMacs status2 cmac2))))(and (=>
    (client1 status1 cmac1)(not(interseptMacs status1 cmac1)))
    (=>(client2 status2 cmac2)(interseptMacs status2 cmac2)))
    (and(=>(client1 status1 cmac1)
    (not(interseptMacs status1 cmac1)))(=> (client2 status2 cmac2)
    (not(interseptMacs status2 cmac2)))))))
    ;Assertion
    (and(= indices indices1)(= letters letters1))
    ) ;ends of => );ends of forall )))
```

Listing F.12: Commands

```
(check-sat)
;(get-model)
(exit)
```

Listing F.13: Commands

```
(check-sat);(get-model)(exit)
```

Bibliography

[1] 24 Days of Hackage: sbv. https://ocharles.org.uk/blog/guest-posts/2013-12-09-24-days-of-hackage-sbv.html. Accessed December 1, 2015.

[2] Emerging Challenges For Security, Privacy And Trust. In D. Gritzalis and J. Lopez, editors, *24th IFIP TC 11 International Information Security Conference, SEC 2009, Pafos, Cyprus, May 18-20, 2009. Proceedings*, volume 297 of *IFIP Advances in Information and Communication Technology*, pages 76–86. Springer, 2009.

[3] Railway applications - communication, signalling and processing systems - software for railway control and protection systems. Brussels, Belgium, 2011. Standard EN 50128:2011, European Committee for Standardization.

[4] W. Adjie-Winoto, E. Schwartz, H. Balakrishnan, and J. Lilley. The design and implementation of an intentional naming system. In *17th ACM Symposium on Operating Systems Principles*, Charleston, SC, December 1999.

[5] A. M. Alabdali, L. Georgieva, and G. Michaelson. Modelling of Secure Data Transmission over a Multichannel Wireless Network in Alloy. In *11th IEEE International Conference on Trust, Security and Privacy in Computing and Communications, TrustCom 2012, Liverpool, United Kingdom, June 25-27, 2012*, pages 785–792, 2012.

[6] R. M. Amadio and D. Lugiez. On the Reachability Problem in Cryptographic Protocols. In *CONCUR 2000 - Concurrency Theory, 11th International Conference, University Park, PA, USA, August 22-25, 2000, Proceedings*, pages 380–394, 2000.

[7] H. Amjad. Combining model checking and theorem proving. Technical report, Univercity of Cambridge, London, 2004.

[8] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. UML2Alloy: A Challenging Model Transformation. In *In: ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS*, pages 436–450. Springer, 2007.

[9] K. Anastasakis, B. Bordbar, G. Georg, and I. Ray. On Challenges Of Model Transformation From UML To Alloy. *Software and Systems Modeling*, 9(1):69–86, Dec. 2009.

[10] F. Barbier, B. Henderson-Sellers, A. Le Parc-Lacayrelle, and J. Bruel. Formalization of the Whole-Part relationship in the Unified Modeling Language. *Software Engineering, IEEE Transactions on*, 29(5):459–470, May 2003.

[11] A. Bauer, M. Pister, and M. Tautschnig. Tool-Support For The Analysis Of Hybrid Systems And Mmodels. In *In Design, Automation and Test in Europe (DATE*, pages 924–929, 2007.

[12] P. Bendersky, J. P. Galeotti, and D. Garbervetsky. The dynalloy visualizer. volume 139 of *Electronic Proceedings in Theoretical Computer Science*, pages 59–64. Open Publishing Association, Jan. 2014.

[13] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement Types for Secure Implementations. In *21st IEEE Computer Security Foundations Symposium (CSF-21)*, pages 17–32, 2008.

[14] T. Benzel. Analysis Of A Kernel Verification. In *IEEE Symposium on Security and Privacy'84*, pages 125–133, 1984.

[15] J. Berdine, B. Cook, and S. Ishtiaq. SLAyer: Memory Safety for Systems-level Code. In *CAV*, 2011.

[16] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant Synthesis for Combined Theories. In B. Cook and A. Podelski, editors, *Proceedings of 8th Verification, Model Checking, and Abstract Interpretation In-*

ternational Conference (VMCAI 2007), volume 4349/2007 of *Lecture Notes in Computer Science*, pages 78–394, Nice, France, January 2007. Springer.

[17] N. Bjørner and L. M. de Moura. Tapas: Theory Combinations and Practical Applications. In *FORMATS*, pages 1–6, 2009.

[18] K. Bk. Optimized Translation of Clafer Models to Alloy. Technical report, university of waterloo, July 2011.

[19] B. Blanchet. Security Protocol Verification: Symbolic and Computational Models. In *Principles of Security and Trust - First International Conference, POST 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012, Proceedings*, pages 3–29, 2012.

[20] B. M. Boreale M, editor. *A framework for the analysis of security protocols. Proceedings*, volume 2421 of *Lecture Notes in Computer Science*. Springer, Berlin Heidelberg New York, (2002).

[21] R. Boyatt and J. Sinclair. Investigating post-completion errors with the Alloy Analyzer. Technical Report CS-RR-433, University of Warwick, Coventry, UK, July 2007.

[22] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. Mathsat: Tight integration of sat and mathematical decision procedures. *J. Autom. Reason.*, 35(1-3):265–293, Oct. 2005.

[23] S. Brands and D. Chaum. Distance-Bounding Protocols. In *EUROCRYPT93, Lecture Notes in Computer Science*, volume 765, pages 344–359. Springer-Verlag, 1993.

[24] C. Brock and J. W.A. Hunt. Formally Specifying And Mechanically Verifying Programs For The Motorola Complex Arithmetic Processor DSP. In *Proceedings of the 1997 International Conference on Computer Design (ICCD '97)*, ICCD '97, pages 31–36, Washington, DC, USA, 1997. IEEE Computer Society.

[25] J. Brunel, L. Rioux, S. Paul, A. Faucogney, and F. Vallée. Formal Safety and Security Assessment of an Avionic Architecture with Alloy. In *Proceedings Third International Workshop on Engineering Safety and Security Systems, ESSS 2014, Singapore, Singapore, 13 May 2014.*, pages 8–19, 2014.

[26] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic Model Checking For Sequential Circuit Verification. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 13(4):401–424, 1994.

[27] L. Buttyn and J.-P. Hubaux. *Security And Cooperation In Wireless Networks : Thwarting Malicious And Selfish Behavior In The Age Of Ubiquitous Computing.* Cambridge University Press, Cambridge, UK, New York, 2008.

[28] D. Calegari, C. Luna, N. Szasz, and A. Tasistro. A Type-Theoretic Framework for Certified Model Transformations. In *Formal Methods: Foundations and Applications - 13th Brazilian Symposium on Formal Methods, SBMF 2010, Natal, Brazil, November 8-11, 2010, Revised Selected Papers*, pages 112–127, 2010.

[29] R. Carbone. *LTL Model-Checking for Security Protocols.* PhD thesis, UNIVERSITY OF GENOVA, 2009.

[30] B. Christianson and J. Li. Multi-channel key Agreement Using Encrypted Public Key Exchange. In *Proceedings of the 15th international conference on Security protocols*, pages 133–138, Berlin, Heidelberg, 2007. Springer-Verlag.

[31] A. Clark and R. Poovendran. A metric for quantifying key exposure vulnerability in wireless sensor networks. In *WCNC*, pages 1–6. IEEE, 2010.

[32] E. M. Clarke. 25 Years Of Model Checking. chapter The Birth of Model Checking, pages 1–26. Springer-Verlag, Berlin, Heidelberg, 2008.

[33] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Progress on the State Explosion Problem in Model Checking. In *Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194, London, UK, UK, 2001. Springer-Verlag.

[34] E. M. Clarke, A. Gupta, J. H. Kukula, and O. Strichman. SAT Based Abstraction-Refinement Using ILP and Machine Learning Techniques. In E. Brinksma and K. G. Larsen, editors, *CAV*, volume 2404 of *Lecture Notes in Computer Science*, pages 265–279. Springer, 2002.

[35] T. Coe, T. Mathisen, C. Moler, and V. Pratt. Computational Aspects Of The Pentium Affair. *IEEE Comput. Sci. Eng.*, 2(1):18–31, Mar. 1995.

[36] R. L. Constable, S. F. Allen, H. M. Bromley, W. R. Cleaveland, J. F. Cremer, R. W. Harper, D. J. Howe, T. B. Knoblock, N. P. Mendler, P. Panangaden, J. T. Sasaki, and S. F. Smith. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1986.

[37] M. Conti, N. Dragoni, and V. Lesyk. A Survey of Man In The Middle Attacks. *IEEE Communications Surveys Tutorials*, 18(3):2027–2051, thirdquarter 2016.

[38] Crisys. Formal Verification - Theorem Proving. http://crisys.cs.umn.edu/theorem-proving.html. Accessed Aug 7, 2012.

[39] S. B. L. Z. D. Marinov, S. Khurshid and M. Rinard. Optimizations for Compiling Declarative Models into Boolean Formulas. In *8th International Conference on Theory and Applications of Satisfiability Testing (SAT 2005)*, Jun 2005.

[40] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[41] L. De Moura and N. Bjørner. Z3: An Efficient SMT Solver. In *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, volume 4963 of *TACAS'08/ETAPS'08*, pages 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[42] R. Demolombe, L. Fariñas Del Cerro, and N. Obeid. Automated Reasoning in Metabolic Networks with Inhibition. In *Proceeding of the XIIIth International Conference on AI*IA 2013: Advances in Artificial Intelligence - Volume 8249*, pages 37–47, New York, NY, USA, 2013. Springer-Verlag New York, Inc.

[43] G. Dennis. TSAFE: Building a trusted computing base for air traffic control software. Masters thesis. 2003.

[44] Y. Desmedt, C. Goutier, and S. Bengio. Special Uses And Abuses Of The Fiat-Shamir Passport Protocol. In C. Pomerance, editor, *CRYPTO*, volume 293 of *Lecture Notes in Computer Science*, pages 21–39. Springer, 1987.

[45] C. Devine. Aircrack-2.41. http://aircrack-ng.org/. Accessed June 17, 2014.

[46] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, Internet Engineering Task Force, April 2006.

[47] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.1. RFC 4346, April 2006.

[48] W. Diffie and M. E. Hellman. New Directions In Cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[49] J. Dingel and T. Filkorn. Model Checking for Infinite State Systems Using Data Abstraction, Assumption-Commitment Style reasoning and Theorem Proving. In P. Wolper, editor, *CAV*, volume 939 of *Lecture Notes in Computer Science*, pages 54–69. Springer, 1995.

[50] D. O. DISTRIBUTED, D. S. F. O. MATHEMATICS, and P. C. U. I. PRAGUE. Alloy. http://d3s.mff.cuni.cz/research/seminar/download/2010-11-10-Bures-Alloy.pdf. Accessed Apr 3, 2014.

[51] J. W. D.Kindred. Theory Generation for Security Protocols. *ACM TOPLAS*, 1999.

[52] D. Dolev and A. C. Yao. On the Security of Public Key Protocols. pages 198–208, Washington, DC, USA, 1983. IEEE Computer Society.

[53] B. Donovan, P. Norris, and G. Lowe. Analyzing a Library of Security Protocols using Casper and FDR. In *In Workshop on Formal Methods and Security Protocols*, 1999.

[54] A. Dwivedi and S. Rath. Formalization of Web Security Patterns. *INFOCOMP Journal of Computer Science*, 14(1):14–25, 2015.

[55] N. Eén and N. Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, pages 502–518, 2003.

[56] A. A. El Ghazi and M. Taghdiri. Analyzing Alloy Constraints using an SMT Solver: A Case Study. In *5th International Workshop on Automated Formal Methods (AFM)*, Edinburgh, United Kingdom, 2010.

[57] J.-C. Fillitre, H. Herbelin, B. Barras, B. Barras, S. Boutin, E. Gimnez, S. Boutin, G. Huet, C. Muoz, C. Cornes, C. Cornes, J. Courant, J. Courant, C. Murthy, C. Murthy, C. Parent, C. Parent, C. Paulin-mohring, C. Paulin-mohring, A. Saibi, A. Saibi, B. Werner, and B. Werner. The Coq Proof Assistant - Reference Manual Version 6.1. Technical report, 1997.

[58] F.Wong and F.Stajano. Multi-Channel Protocols: Strong Authentication Using Camera-Equipped Wireless Devices. *Security Protocols 13, LNCS*, 4631:112–132, 2007.

[59] D. Gabbay, A. Kurucz, F. Wolter, and M. Zakharyaschev. MANY-DIMENSIONAL MODAL LOGICS: THEORY AND APPLICATIONS. Unpublished book January 2003, Department of Computer Science, King's College, 2003.

[60] U. Geilman. *Verifying Alloy Models Using Key.* PhD thesis, Institute of Informatics, Instatute for Theoretical Computer Science, 2011.

[61] G. Georg, J. Bieman, and R. B. France. Using Alloy And UML/OCL to Specify Run-Time Configuration Management: A Case Study. In A. Evans,

R. B. France, A. M. D. Moreira, and B. Rumpe, editors, *pUML*, volume 7 of *LNI*, pages 128–141. GI, 2001.

[62] A. A. E. Ghazi. *Relational Reasoning Constraint Solving, Deduction, and Program Verification.* PhD thesis, von der Fakultat fur Informatik des Karlsruher Instituts fur Technologie (KIT), 2015.

[63] A. A. E. Ghazi, U. Geilmann, M. Ulbrich, and M. Taghdiri. A Dual-Engine for Early Analysis of Critical Systems. *CoRR*, abs/1408.0707, 2014.

[64] A. A. E. Ghazi and M. Taghdiri. Relational reasoning via smt solving. In M. Butler and W. Schulte, editors, *FM*, volume 6664 of *Lecture Notes in Computer Science*, pages 133–148. Springer, 2011.

[65] E. Goldberg and Y. Novikov. BerkMin: A fast and robust Sat-solver. *Discrete Applied Mathematics*, 155(12):1549–1561, 2007.

[66] M. J. C. Gordon and T. F. Melham, editors. *Introduction to HOL: A Theorem proving Environment for Higher Order Logic.* Cambridge University Press, 1993.

[67] S. Graf and H. Saidi. Construction of Abstract State Graphs with PVS. In *CAV'97*, volume 1254 of *LNCS*, pages 72–83, 1997.

[68] K. Grover, A. Lim, and Q. Yang. Jamming and Anti-jamming Techniques in Wireless Networks: A Survey. *Int. J. Ad Hoc Ubiquitous Comput.*, 17(4):197–215, Dec. 2014.

[69] O. Hak5Darren. Hacking Wireless Networks With Man In The Middle Attacks . http://www.youtube.com/watch?v=N86xJpna9Js, 2008. Accessed Jul 19, 2011.

[70] C. He. *Analysis Of Security Protocols For Wireless Networks.* PhD thesis, Electrical Engineering And The Committee On Graduate Studies Of Stanford University, 2005.

[71] C. Heitmeyer, M. Archer, E. Leonard, and J. McLean. Applying Formal Methods To A Certifiably Secure Software System. *IEEE Transactions on Software Engineering*, 34:82–98, 2008.

[72] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. "Using Formal Specifications to Support Testing". *ACM Computing Surveys*, 41(2):1–76, Feb. 2009.

[73] J. Holmström, J. Rajamäki, and T. Hult. DSiP Distributed Systems Intercommunication Protocol: A Traffic Engineering Solution For Secure Multichannel Communication. In *Proceedings of the 10th WSEAS international conference on communications, electrical &#38; computer engineering, and 9th WSEAS international conference on Applied electromagnetics, wireless and optical communications*, ACELAE'11, pages 57–60, Stevens Point, Wisconsin, USA, 2011. World Scientific and Engineering Academy and Society (WSEAS).

[74] G. J. Holzmann. Trends In Software Verification. In *In: Proceedings of the Formal Methods Europe Conference*, 2003.

[75] G. J. Holzmann. *The SPIN Model Checker - Primer and Reference Manual.* Addison-Wesley, 2004.

[76] H. Hu. *Assurance Management Framework for Access Control Systems.* PhD thesis, ARIZONA STATE UNIVERSITY, 2012.

[77] D. Jackson. Automating First-order Relational Logic. In *Proceedings of the 8th ACM SIGSOFT International Symposium on Foundations of Software Engineering: Twenty-first Century Applications*, SIGSOFT '00/FSE-8, pages 130–139, New York, NY, USA, 2000. ACM.

[78] D. Jackson. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, Apr. 2002.

[79] D. Jackson. Micromodels Of Software: Lightweight Modelling And Analysis With Alloy. Technical report, Software Design Group. MIT Lab for Computer Science, 2002.

[80] D. Jackson. Alloy 3.0 Reference Manual. Technical report, 2004. Available at `http://homepage.divms.uiowa.edu/~pgaroche/181/Papers/Jack04.pdf`.

[81] D. Jackson. *Software Abstractions: Logic, Language, And Analysis*. The MIT Press, 2006.

[82] D. Jackson. The Alloy Analyzer Layout . http://alloy.mit.edu/alloy/documentation/quickguide/gui.html, 2012. Accessed Apr 17, 2015.

[83] D. Jackson. Alloy: Language and Tool for Relational Models. http://alloy.mit.edu/alloy/faq.html, 2012. Accessed January 17, 2014.

[84] D. Jackson and C. Damon. Semi-executable Specifications. Technical report cmucs-95-216, school of computer science, carnegie mellon university, pittsburgh, pa,, November 1995.

[85] D. Jackson and M. Jackson. Separating Concerns in Requirements Analysis: An Example, chapter Rigorous development of complex fault tolerant systems. In *RODIN Book*, pages 210–225. Springer, 2006.

[86] D. Jackson, I. Schechter, and I. Shlyakhter. Alcoa: the Alloy constraint analyzer. In *Software Engineering, 2000. Proceedings of the 2000 International Conference on*, pages 730–733, 2000.

[87] A. Joshi, S. P. Miller, and M. P. E. Heimdahl. Mode Confusion Analysis Of A Flight Guidance System Using Formal Methods. In *Proceedings of the 22nd Digital Avionics Systems Conference*, volume 1, page 2D.12112, Piscataway, Oct. 2003. IEEE.

[88] E. M. C. Jr., O. Grumberg, and D. A. Peled. *Model Checking*. The MIT Press, 1999.

[89] V. L. Juncheng Wu, Gang Liu. Formal Verification. http://people.cis.ksu.edu/˜hankley/d841/Fa99/chap4.html. Accessed Nov 15, 2015.

[90] M. Kaufmann, P. Manolios, and J. S. Moore. *Computer-aided reasoning : an approach*. Advances in formal methods. Kluwer Academic Publishers, Boston, 2000.

[91] M. J. Kaufmann, M. Some Key Research Problems In Automated Theorem Proving For Hardware And Software Verification. *In: Spanish Royal Academy of Science (RAMSAC)*, 98:181–196, 2004.

[92] S. A. Khalek and S. Khurshid. Systematic Testing Of Database Engines Using A Relational Constraint Solver. In *ICST*, pages 50–59, 2011.

[93] I.-G. Kim and J.-Y. Choi. Formal Verification Of PAP And EAP-MD5 Protocols In Wireless Networks: FDR Model Checking. In *18th International Conference on Advanced Information Networking and Applications (AINA 2004), 29-31 March 2004, Fukuoka, Japan*, pages 264–269. IEEE Computer Society, 2004.

[94] M. Kumar and S. Goel. Specifying safety and critical real-time systems in z. In *Computer and Communication Technology (ICCCT), 2010 International Conference on*, pages 596–602, Sept 2010.

[95] R. P. Kurshan and L. Lamport. Verification of a Multiplier: 64 Bits and Beyond. In C. Courcoubetis, editor, *Computer Aided Verification: Proc. of the 5th International Conference CAV'93*, pages 166–179. Springer, Berlin, Heidelberg, 1993.

[96] S. Kyungah. Cryptanalysis Of Mutual Authentication And Key Exchange For Low Power Wireless Communications[J]. *IEEE Communications Letters*, 7(5):248–250, 2003.

[97] S. Lal, M. Jain, and V. Chaplot. Approaches to Formal Verification of Security Protocols. *CoRR*, abs/1101.1815, 2011.

[98] L. Lensink. *Applying Formal Methods in Software Development.* PhD thesis, radboud university nijmegen, 2013.

[99] F. Lerda. LTL Model Checking [PowerPoint slides]. https://www.cs.cmu.edu/ẽmc/15817-s05/ltlmc.ppt. Accessed May 17, 2015.

[100] J. Li, B. Christianson, and M. Loomes. Fair Authentication In Pervasive Computing. In M. Burmester and A. Yasinsac, editors, *Secure Mobile Ad-hoc Networks and Sensors, First International Workshop, MADNES 2005, Singapore, September 20-22, 2005, Revised Selected Papers*, volume 4074 of *Lecture Notes in Computer Science*, pages 132–143. Springer, 2005.

[101] A. Lin, M. Bond, and J. Clulow. Modeling Partial Attacks with ALLOY. In *Proceedings of the 15th international conference on Security protocols*, pages 20–33, Berlin, Heidelberg, 2010. Springer-Verlag.

[102] L. Lockefeer. *Formal specification and verification of TCP extended with the Window Scale Option.* PhD thesis, Vrije Universiteit Amsterdam, 2013.

[103] G. Lowe. Breaking and fixing the Needham-Schroeder Public-Key Protocol using FDR. In T. Margaria and B. Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 1055 of *Lecture Notes in Computer Science*, pages 147–166. Springer Berlin / Heidelberg, 1996. 10.1007/3-540-61042-$1_4$3.

[104] G. Lowe. Breaking and Fixing the Needham-Schroeder Public-Key Protocol Using FDR. *Proc. 2nd Int'l Workshop on Tools and Algorithms for Construction and Analysis of Systems (TACA 96)*, page 147166., 1996.

[105] A. D. Lucia, F. Ferrucci, G. Tortora, and M. Tucci. *Emerging Methods, Technologies and Process Management in Software Engineering.* Wiley-IEEE Computer Society Pr, 2008.

[106] W. Marrero, E. Clarke, and S. Jha. A Model Checker For Authentication Protocols. In *Rutgers University*, pages 134–141, 1997.

[107] W. Marrero, E. Clarke, and S. Jha. Model Checking For Security Protocols. Technical report, CARNEGIE MELLON UNIVERSITY, 1997.

[108] J. Mclean. Security Models. In *Encyclopedia of Software Engineering*, pages 1136–1145. Wiley Sons, 1994.

[109] C. Meadows. Formal methods for cryptographic protocol analysis: emerging issues and trends. *IEEE Journal on Selected Areas in Communications*, 21(1):44–54, 2003.

[110] S. V. Millen JK. Constraint solving for bounded-process cryptographic protocol analysis. In *In: Proceedings of CCS01*, page 166175. ACM Press, 2001.

[111] D. L. Mitchell, N. A. Durgin, P. D. Lincoln, J. C. Mitchell, and A. Scedrov. Undecidability of Bounded Security Protocols. 1999.

[112] S. U. Mitchell JC, Mitchell M. Symbolic protocol analysis with products and Diffie-Hellman exponentiation. In *In: Proceedings of CSFW03*, page 4761. IEEE Press, 2003.

[113] L. Momtahan. Towards a small model theorem for data independent systems in alloy. *Electronic Notes in Theoretical Computer Science*, 128(6):37 – 52, 2005. Proceedings of the Fouth International Workshop on Automated Verification of Critical Systems (AVoCS 2004)Automated Verification of Critical Systems 2004.

[114] MontyNewborn. Automated Theorem Proving: Theory and Practice. Springer-Verlag, 2000.

[115] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, pages 530–535, New York, NY, USA, 2001. ACM.

[116] F. Mostefaoui and J. Vachon. Verification Of Aspect-UML Models Using Alloy. In *Proceedings of the 10th international workshop on Aspect-oriented modeling*, AOM '07, pages 41–48, New York, NY, USA, 2007. ACM.

[117] C. Nachreiner. Foundations: What Are NIC, MAC and ARP? . http://www.watchguard.com/infocenter/editorial/135250.asp. Accessed Aug 5, 2012.

[118] R. M. Needham and M. D. Schroeder. Using Encryption for Authentication in Large Networks of Computers. *Commun. ACM*, 21(12):993–999, Dec. 1978.

[119] D. P. Nigam and A. Ojha. An Aspect Oriented Model Mf Efficient And Secure Card-Based Payment System. In S. K. Jena, R. Kumar, A. K. Turuk, and M. Dash, editors, *ICCCS*, pages 559–564. ACM, 2011.

[120] Owasp. Man-In-The-Middle Attack . http://www.owasp.org/index.php/Man-in-the-middle_attack, 2009. Accessed Mar 23, 2011.

[121] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. pages 411–414. Springer-Verlag, 1996.

[122] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof Checking, and Model Checking. pages 411–414. Springer-Verlag, 1996.

[123] G. K. Palshikar. An Introduction To model Checking, 2004. http://webdocs.cs.ualberta.ca/p̃aullu/C605/EMS-2004-02-12.pdf.

[124] A.-S. K. Pathan, T. T. Dai, and C. S. Hong. An Efficient LU Decomposition-Based Key Pre-Distribution Scheme For Ensuring Security In Wireless Sensor Networks. In *Proceedings of the Sixth IEEE International Conference on Computer and Information Technology*, CIT '06, page 227, Washington, DC, USA, 2006. IEEE Computer Society.

[125] S. K. Paul S Grisham, Charles L. Chen and D. E. Perry. Validation of a Security Model with the Alloy Analyzer. 2007.

[126] L. C. Paulson. Inductive analysis of the internet protocol tls.

[127] L. C. Paulson. The Foundation of a Generic Theorem Prover. *J. Autom. Reasoning*, 5(3):363–397, 1989.

[128] L. C. Paulson. The Inductive Approach To Verifying Cryptographic Protocols. *Journal of Computer Security*, 6:85–128, 1998.

[129] A. Perrig, R. Szewczyk, J. D. Tygar, V. Wen, and D. E. Culler. SPINS: Security Protocols for Sensor Networks. *Wirel. Netw.*, 8(5):521–534, Sept. 2002.

[130] A. Pnueli. The Temporal Logic Of Programs. In *Proceedings of the 18th Annual Symposium on Foundations of Computer Science*, SFCS '77, pages 46–57, Washington, DC, USA, 1977. IEEE Computer Society.

[131] R. Podorozhny, S. Khurshid, D. Perry, and X. Zhang. Verification of Multi-agent Negotiations Using the Alloy Analyzer. In *Proceedings of the 6th International Conference on Integrated Formal Methods*, IFM'07, pages 501–517, Berlin, Heidelberg, 2007. Springer-Verlag.

[132] Policy. Information Security Policy. http://policy.nd.edu/policy_files/ InformationSecurity Policy.pdf, 2009. Accessed March 1, 2012.

[133] Policy. Man-In-The-Middle Attack (MITM). http://policy.nd.edu/policy_files/ InformationSecurity Policy.pdf, 2009. Accessed July 12, 2011.

[134] J. Qadir and O. Hasan. Applying Formal Methods to Networking: Theory, Techniques and Applications. *CoRR*, abs/1311.4303, 2013.

[135] D. R. Quinta. *Application of Formal Methods in The ITASAT project.* PhD thesis, Informatic, 2013.

[136] S. M. M. Rahman, N. Nasser, A. Inomata, T. Okamoto, M. Mambo, and E. Okamoto. Anonymous authentication and secure communication protocol for wireless mobile *ad hoc* networks. *Security and Communication Networks*, 1(2):179–189, 2008.

[137] S. Rajan, N. Shankar, and M. Srivas. An Integration of Model-Checking with Automated Proof Checking. In P. Wolper, editor, *Computer-Aided Verification, CAV '95*, volume 939 of *Lecture Notes in Computer Science*, pages 84–97, Liege, Belgium, June 1995. Springer-Verlag.

[138] Ranise And Cesare Tinelli. The Satisfiability Modulo Theories Library (SMT-LIB). www.SMT-LIB.org, 2006.

[139] S. Ray. *Scalable Techniques for Formal Verification.* Springer, Dordrecht, 2010.

[140] M. Reynolds. Lightweight modeling of java virtual machine security constraints. In *Abstract State Machines, Alloy, B and Z, volume 5977 of Lecture Notes in Computer Science,*, page 146159. Springer Berlin Heidelberg,, 2010.

[141] A. Richard, J.-P. Comet, and G. Bernot. *Modern Formal Methods and Applications*, pages 83–122. Springer, ISBN: 1-4020-4222-1, 2006.

[142] E. Romanowicz. Verification Of Programs With Z3, Open Access Dissertations And Theses. Available:http://digitalcommons.mcmaster.ca/opendissertations/4339, 2010. Accessed Jul 5, 2012.

[143] J. Rushby. Formal Specification and Verification for Critical Systems: Tools, Achievements, and Prospects . *IEEE Transactions on Software Engineering*, 21(2):107–125, 1995.

[144] G. M. L. G. R. B. Ryan P, Schneider S. *Modelling and analysis of security protocols.* Pearson Education Ltd, 2000.

[145] G. A. U. A. R. B. C. B. P. A. B. F. B. W. D. N. E. N. E. F. T. G. N. H. J. M. M. M. A. M. S. C. M. H. J. O. S. S. M. S. K. U. S. U. S. S. Abdennadher, J. Alves Alferes and G. Wagner. Automated Reasoning on the Web. *Communications of Applied Logic*, 9, 2004.

[146] H. Saïdi. Model Checking Guided Abstraction and Analysis. In *Proceedings of the 7th International Symposium on Static Analysis*, SAS '00, pages 377–396, London, UK, UK, 2000. Springer-Verlag.

[147] J. L. M. Silva. GRASP - A New Search Algorithm For Satisfiability. In *in Proceedings of the International Conference on Computer-Aided Design*, pages 220–227, 1996.

[148] P. A. Song D, Berezin S. Athena: a novel approach to efficient automatic security protocol analysis. 9:4774, 2001.

[149] F. Stajano, F.-L. Wong, and B. Christianson. Multichannel Protocols To Prevent Relay Attacks. In *Proceedings of the 14th international conference on Financial Cryptography and Data Security*, FC'10, pages 4–19, Berlin, Heidelberg, 2010. Springer-Verlag.

[150] M. Taghdiri and D. Jackson. A Lightweight Formal Analysis Of A Multicast Key Management Scheme. In H. Knig, M. Heiner, and A. Wolisz, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2003, 23rd IFIP WG 6.1 International Conference, Berlin, Germany, September 29 - October 2, 2003, Proceedings*, volume 2767 of *Lecture Notes in Computer Science*, pages 240–256. Springer, 2003.

[151] M. Tanaka. "Using Formal Specifications to Support Testing". *Verifying Security Protocols Using Theorem Provers*, pages 79–86, 2007.

[152] M. Toahchoodee and I. Ray. Using alloy to analyse a spatio-temporal access control model supporting delegation. *Information Security*, 3(3):75–113, Sept. 2009.

[153] S. Vakilinia, M. H. Alvandi, M. R. K. Shoja, and I. Vakilinia. Multi-path multi-channel protocol design for secure qos-aware VOIP in wireless ad-hoc networks. In *WMNC*, pages 1–6, 2013.

[154] H. C. A. van Tilborg and S. Jajodia, editors. *Encyclopedia Of Cryptography And Security, 2nd Ed.* Springer, 2011.

[155] C. M. Wintersteiger, Y. Hamadi, and L. M. de Moura. Efficiently Solving Quantified Bit-Vector Formulas. In *Proceedings of 10th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2010, Lugano, Switzerland, October 20-23*, pages 239–246, 2010.

[156] F. L. Wong. Protocols And Technologies For Security In Pervasive Computing And Communications. Technical Report UCAM-CL-TR-709, University of Cambridge, Computer Laboratory, Jan. 2008.

[157] S.-L. Wu, C.-Y. Lin, Y.-C. Tseng, and J.-P. Sheu. A New Multi-Channel MAC Protocol With On-Demand Channel Assignment For Multi-Hop Mobile Ad Hoc Networks. In *Proceedings of the 2000 International Symposium on Parallel Architectures, Algorithms and Networks*, ISPAN '00, pages 232–237, Washington, DC, USA, 2000. IEEE Computer Society.

[158] T. Ye, D. Veitch, and J. C. Bolot. Improving wireless security through network diversity. *Computer Communication Review*, 39(1), January 2009.

[159] C. H. Z. Htike. Cognitive Radio Based Jamming Resilient Multi-channel MAC Protocol For Wireless Network. Technical report, 2009.

[160] Y. Zhao, Z. Yang, J. Xie, and Q. Liu. Formal Model and Analysis of Sliding Window Protocol Based on NuSMV. *JCP*, 4(6):519–526, 2009.