Ministry of Higher Education and

Scientific Research University of

Babylon College of Science for Women

Department of Computer Science

Clarification of Ambiguity for the Simple Authentication and Security Layer

A Project

Submitted to the Council of College of Science-University of Babylon in Partial Fulfillment of the Requirements for the Degree of Higher Diploma in / Computer Science

By

Jinan Jassim Hussein Taeis

Supervised By

Dr. Farah Al-Shareefi

2021 A.D. **1443 A.H.**

بِسْمِ اللَّهِ الرَّحْمَنِ الرَّحِيمِ

وَأَنزَلَ اللَّهُ عَلَيْكَ الْكِتَابَ وَالْحِكْمَةَ وَعَلَّمَكَ مَا لَمْ تَكُنْ تَعْلَمُ وَكَانَ فَضْلُ اللَّهِ عَلَيْكَ عَظِيمًا ۝

صدق الله العظيم

سورة النساء آية

(113)

Supervisor Certification

I certify that this research titled "**Clarification of Ambiguity for the Simple Authentication and Security Layer**".

Was prepared at the Department of Computer Science/ College of Science for the Women/ University of Babylon, by (**Jinan Jassim Hussein)** as partial fulfillment of the requirements for the degree of Higher Diploma of Science in Computer Science .

Signature:

Name: Dr. Farah Al-Shareefi

Date: / / 2021

Address: University of Babylon/College of Science for Women

The Head of the Department Certification

In view of the available recommendations, I forward the research entitled ”**Clarification of the Ambiguity for the Simple Authentication and Security Layer**” for debate by the examination committee.

Signature:

Name: Dr. Farah Al-Shareefi

Date: / / 2021

Address: University of Babylon/College of Science for Women

Dedication

To: the fountain of giving that instilled in me the ambition and

perseverance my dear father.

To: the fountain of tenderness that is inexhaustible my precious mother.

To: those who carry in their eyes the memories of my childhood and youth

my brothers and sisters .

To: the person who stood with me and supported me in my life my dear husband.

To: my eyes, the fruit of my heart, and the hope in my world my a children's.

To: my professors at the College of Science for Women / University of Babylon specialy the head of Computer Science (Dr. Farah AL-Shareefi).

To: my gratitude is extended to my colleages.

I dedicate my research to them.

Jinan Jassim Hussein

Acknowledgments

I woud like to thank and appreciate Dr. Farah Al-Shareefi, for her support to me along my study bless her .All words of thanks and respect are unable to repay the great efforts that she made to carry out this research...

I woud also thank and appreciate all the distinguished staff, in the Department of Computer Science/The College of Sciences, for their valuable efforts for me.My thanks is extended to all those who helped me.

Jinan Jassim Hussein

Abstract

The Simple Authentication and Security Layer (SASL) , as its name suggests, is a framework for permitting Internet protocols to support security services, including authentication and optionally integrity and confidentiality checking. The SASL is first specified in RFC 2222, and subsequently upgraded in RFC 4422, in both of the natural language has been used.

However, natural language usually tends to improper interpretations because of its inherent ambiguities as well as imprecision. Although Oracle has an SASL implementation, its documentation also contains undefined RFC specifications and informal descriptions.

By utilizing Abstract State Machines (ASMs), this research focuses on elucidating ambiguity in SASL. This elucidation is founded on two ASM concepts: a ground model that concisely captures the original SASL behavior and a refinement concept that precisely explicates the vague aspects of the behavior.

We also show some differences between RFCs and the Oracle implementation description. We believe that the research findings can be used as a starting point for further implementation and formal analysis.

List of Contents

List of Figure

List of code

List of Tables

List of Abbreviations

Abbreviation	Meaning
SASL	Simple Authentication and Security Layer
ASM	Abstract State Machine
RFC	Requests for Comments
AsmL	Asmeta Language
DASM	Distributed Abstract State Machine
ATD	Automated Train Door
SSF	Security Strength Factor
MAC	Message Authentication Code
QOP	Quality of Protection

CHAPTER ONE

GENERAL OVERVIEW

1.1 Introduction

Security protocols are the foundations of the infrastructures needed to communicate securely over public networks. When engineering a security protocol, it is recommended that its specifications are defined as accurately as possible. These specifications are initially written in natural language. Unfortunately, natural language is notoriously known for its potential ambiguities. For instance, the Simple Authentication and Security Layer (SASL) is an example of such protocols that have requirements described in natural language. SASL is introduced to supply services, such as authenticity, integrity, and confidentiality, to communication protocols [20,15].

To address the issue of ambiguities, formal methods are heavily emphasized. Formal methods are used to boost confidence in the protocol's correct requirements. These methods are mathematically based languages and analysis tools that can be used to perform the following tasks [8]:

- **Modeling** is a conceptual abstraction that describes the behavior of a protocol [17].
- **Model simulation** is the process of conducting experiments with a designed model to evaluate the protocol's performance under various operational conditions [17].

From formal Methods, we have chose the Abstract State Machine (ASM) method [11,9]. ASMs are a universal state machine formalism that can be used to model any algorithm at an adequate level of abstraction. The ASM method has been chosen for the following reasons:

1) The method's generality, means that the ASM method can be used to define any system, at a needed level of abstraction. 2) The ASM method has a rich and simple syntax, as well as precise semantics, which together will aid in the specification of models that balance between abstraction and accuracy. 3) The ASM method employs two concepts: the ground model, which is used to capture informal requirements in an abstract and precise manner, and the stepwise refinement concept that gradually refines the ground model towards a more detailed and cohesive one. These concepts, when are combined, they help to clarify the requirements for complex protocols like SASL.

1.2 Problem Statement

SASL framework was firstly stated in Requests for Comments (RFC) 2222 document [15], and then revised in RFC 4422 [20], using informal English language. Unfortunately, the RFCs are sometimes ambiguous because they are written in natural language, which corresponds with the informality and imprecision. Despite the fact that Oracle has an SASL implementation [21], its documentation also contains textual interpretations and some underdefined aspects of the RFCs. Furthermore, as the Oracle's source code of its implementation is not publicly accessible, their particulars are unknown.

1.3 Research Objectives

This research project aims at tackling the above-mentioned problems by:

1. Employing the Abstract State Machine (ASM) method because it can be used to specify systems in a mathematically rigorous, understandable, and scalable manner.
2. Implementing two ASM method strategies: First, the ground model directly captures informal SASL behavior in a way that is both understandable and precise. Second, the stepwise refinement strategy that allows us to elucidate and re-elaborate the ground model's under-defined notions.
3. Emphasizing the key differences between RFCs document and the Oracle implementation.

1.4 Related Work

Our work clarifies ambiguities in the informal SASL description, depending on the ASM method. As a result, this section will discuss other works associated with either clarifying vagueness or with the ASM method.

The ASM formalism is employed in [4] to obtain a formal model of the Kerberos Authentication System, which depends on the Needham and Schroeder protocol. The formal model serves as a foundation for determining the system's minimum assumptions and analyzing its security flaws.

The ASM ground model of a content adaptation system is used for interactions between client devices and the cloud in [7]. This work is supplemented in [3], through refining the former model into a more

elaborated one by paying attention to the interactions between the client and the middleware server in order to retrieve device-related information. The modeling process has also been aided by validation and verification activities that are built into the ASMETA framework.

In [27], the AsmL language is used to abstractly specify the encryption and decryption processes. However, this specification deviates from the theoretical model of ASMs because it is based on object-oriented features and constructs. Higher-order logic (HOL4) is used by the researchers in [5] to create a rigorous post-hoc specification for TCP, UDP, and the Sockets API that reflects the behavior of various implementations, such as FreeBSD 4.6, Linux 2.4.20-8, and Windows XP SP1. They check whether their specification is met by comparing it to tens of thousands of traces obtained from these implementations. In the context of our work, this paper is noteworthy because its authors are motivated by the desire to improve the clarification and precision over ambiguous RFC informal specifications, which can lead to inconsistent implementations. However, validating that the implementation satisfies the specification is not considered by our work. The study is primarily concerned with elucidating ambiguities in the RFC description and explicating uncertainty in the implementation's textual explanation. In addition, unlike HOL4, which requires extensively annotating the mathematical definitions along with informal specification, the specification is expressed using the ASM method because it is accessible and requires the least amount of notational coding.

1.5 Research Organization

this research is written as following:

- Chapter one presents an example of the protocols that have requirements described in natural language such as SASL with a description of one the official methods used for this purpose such as ASM.
- Chapter two presents some underlying information of the selected case study, SASL and the applied formal method, that is known as an Abstract State Machines (ASMs).
- Chapter three shows proposed method of the elucidating several ambiguities existed in the informal requirements specified for the SASL framework.
- Chapter four show result and discussion.
- Chapter five presents conclusions and further works.

CHAPTER TWO

THEORETICAL BACKGROUND

2.1 Introduction

This chapter presents some underlying information of the selected case study, which is the Simple Authentication and Security Layer (SASL), and the applied formal method, that is known as Abstract State Machines **(ASMs)**.

2.2 SASL: Case Study

The case study, which is known as Simple Authentication and Security Layer (SASL), is described in this section.

SASL is first presented in RFC 2222 [15], and later extended in RFC 4422 [20], as an essential structure for adding authentication facility to networking protocols via switchable mechanisms. Moreover , this structure can optionally provide security layer service(s), such as integrity and/or confidentiality. These services are provided through the use of a shared abstraction layer with a structured interface between the used protocols and mechanisms. Through this layer, any SASL-enabled protocol, for example: IMAP [25], SMTP [26], etc., is able to use any SASL-enabled mechanism, for example: PLAIN [18], DIGEST-MD5[24], etc, see Figure 2.1 that conceptually shows the SASL framework.

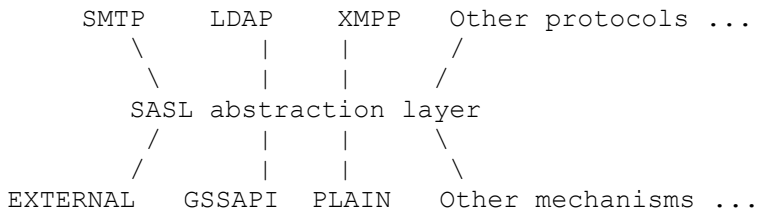

Figure 2.1 Conceptual SASL Structure

According to RFC 2222/4422 [15,20], the client and server which belong to SASL protocol application first negotiate the selection of a suitable mechanism, then the authentication negotiates is launched. In essence, the client requests a SASL connection to the server. Later, the server responds with a list of authentication mechanisms that are supported. Subsequently, the client chooses the best mechanism. After that, the client initiates the authentication by sending to the server an authentication command, which includes the chosen mechanism and, optionally, authentication data. The authentication exchange continues until the client or server either succeeds, fails, or aborts the authentication. When the chosen mechanism underpins the security layer, the client and server negotiate the use of a security layer through the authentication exchange. If they both agree to use it, they must then agree on the maximum size of the cipher text buffer that each side can receive. However, the RFCs specification leaves a number of questions unanswered, including:

- How the server announces its list of mechanisms;
- How the client can choose the best mechanism;
- When the client and the server come to an agreement to utilize the security layer and how this layer can be used; and
- How the maximum size of cipher text buffer size is negotiated.

Some of the above questions stem from the ambiguity and lack of details in the informal description of the API routines that exist in the documentation of Oracle implementation [21].

As reported by Oracle implementation [21], the application calls a suitable API routine to communicate with the abstract layer, and the called routine itself calls a plug-in interface for mechanisms. One of these routines is the sasl_client_start() routine which is called by the client to pick the best

mechanism depending on the security attributes. The main attributes that limits the selection of a mechanism are the **security polices**, including but not limited to NOPLAINTEXT, NOACTIVE, NOANONYMOUS, as well as the **maximum Security Strength Factor (SSF)** [21] for the client, server, and mechanism. The SSF is an integer number that represents the strength of the security layer as follows:

- **If it is 0**, it means authentication.
- **In case it is 1**, it refers to both integrity and authentication.
- **When it is greater than 1**, it indicates confidentiality and integrity besides authentication, and the length of the assumed encryption key.

The server also uses a routine called sasl_listmech() to obtaint a list of mechanisms that meet the security policies.

2.3 The Applied Formal Method: Abstract State Machines

Abstract State Machines (ASMs) is a rather new formal method that originally is introduced by Yuri Gurevich [32, 33] as a universal formalism for building a model to any algorithm at an acceptable level of abstraction. He developed the basic hypothesis of ASMs formalism as: each algorithm, regardless of the way for abstracting it, is emulated step-by-step by a suitable ASM. The concept of abstract states gives ASMs their generality. In classical state machines, like turing machines and finite state machines, states have symbolical representation via a collection of symbols. Abstract states, on the other hand, have syntactical and semantical representations that are achieved by mathematical structures of items from domains with defined functions and predicates.

ASMs also have change relations that are defined by rules for describing the updating .The function of interpretations from one state to the next. ASM

specifications specifies how the system's state changes over time as a result of transition rules. ASMs are being farther upgraded into a practically and arithmetically well-founded method for designing and analyzing a high-level system [11,9]. For real-world problems, this method links the gap between human comprehension, formalization, and executable state machines. ASMs enhance the development process of a system by establishing a precise high-level mode that is destined for producing executable code. ASMs have a wide range of useful applications, like specifying different systems, such as sequential, parallel, and distributed [11], specifying dynamic databases [6], determining a specification for programming languages like UML, Java, and Prolog [10, 30, 12, 29, 19], demonstrating compiler correctness [31], modeling and simulating security protocols [14, 13,28], and so on.

The ASM method is based on three fundamentals concepts:

- **Basic abstract states** are mathematical abstractions which are relied on abstract states, to define the systems structure, and on transition rules, to specify the dynamic behavior of systems;
- **A ground model** is a mechanism for capturing the determined requirements of a systems through a succinct and rigorous conceptual model;
- **Stepwise refinement** is a general approach to build a hierarchical refined models from an abstract ground model depending on the design decision. The final produced model is more comprehensive and implementation-linked one.

2.3.1 The First Concept: Basic Abstract State Machines

ASMs, or basic ASMs, are created to describe the situation in which a single agent can perform multiple actions at the same time. Later, as discussed in

section 2.3.4, this concept is extended to distributed multi-agents that takes action and reaction in a concurrent or inconcurrent pattern.

Basic ASMs are formally defined as finite sets of move rules with the following form:

If $Condition$ **Then** $Updates$

This is the key form for transiting from one abstract state into another. The Condition (or guard) is a predicate formula that has a first-order relation with true or false interpretation. While, the term "Updates" refers to a limited number of update functions of the following:

$$f(t_1, \dots, t_n) \coloneqq t$$

where f is a selected name of an n-ary function, $(t_1, \dots, t_n)$ are first-order arguments (terms) of of a function, t is the value of updating a function. In other words, when a finite set of functions changes their values, the transition occurs from one state to another. A basic ASM is made up of four parts:

- **A Signature (Σ):** denotes the desired definitions functions and domains.
- **Initial states** (I): are a set of states specified by imposed conditions on the signature.
- **Transition rules (TR):** are a set of relations that defines sets of update dedicated for abstract states.
- **Main rule (R):** represents a main machine's transition rule without arity.

2.3.1.1 Illustrated Example

This section shows an illustrated example, called Automated Train Door (ATD) [16], in order to familiarize the reader with ASMs formalism.

With ATD system, a train's physical door is controlled by a computerized controller, which receives sensitive data and issues commands. The ATD system is typically made up of the following parts: a) computerized controller; b) door_sensor; c) train_sensor; d) emergency_sensor; e) actuator; and f) door. The operating principle of the ATD system is best comprehended in terms of its parts. The computerized controller handles the sensor inputs to give an open or close door command. The door sensor transmits data about the position of the door and the presence of obstacles. The train sensor sends out signals that show the train's motion and position in relation to the platform. When an emergency situation arises, such as a fire or toxic gas, the emergency sensor sends out an emergency sign. The actuator operates the door in accordance with the issued control command. The requirements state that the door should not close on a person who is standing in the doorway, that the door should not be opened while the train is in movement situation or when it is not completely aligned or positioned with a platform, and that the passengers should be able to escape in the event of an emergency.

2.3.1.2 Vocabulary Employed in ASMs

This section explains some vocabulary used in ASMs to help understanding the way of executing this machine.

- **Domain**

A domain (also known as a universe) is a “finite or infinite” collection of items used by a machine. An ASM state $\mathfrak{S}$ has a super domain or super,

universe $\mathfrak{D}$ that is partitioned into smaller domains belonging to certain categories.

- **Function**

In ASMs, a function is defined via its name and parameters using the below form:

$$f(t_1, \dots, t_n)$$

where *f* is a name selected to a function, and $t_1, \dots, t_n$ are the n parameters of a function; n is the number of function parameters (also called the function's arity). Functions without arity (i.e., zero arity or nullary functions) are called constants. A selected name of a function is established in the signature part. When the parameters t_i of f are evaluated to their new values, for example v_i, and an $f(v_1, \dots, v_n)$ is evaluated to v in the current state, then the function f is updated to v as well to be the new value of this function in the next state. A function name together with its associated parameter's values account for a location. Typically, a value of a location is deduced from a pair of function's name and its indicated parameter values.

Functions are divided into two categories: **Basic functions** and **Derived functions**. Derived functions are supplementary functions that have a specification or computational design to return values for presented read-only parameters. The basic functions differ from the derived functions in that they do not have specific specification for updating their values; rather, they are updated by the machine (its rules), the environment (the machine's user), or both. Basic functions are more divided into static and dynamic functions, depending on how their values are updated.

Static functions have values that never change during machine execution, i.e. they have constant values.

Dynamic functions are functions that have values changing from one state to the next, i.e., these functions are similar to variables in usual programming language. Dynamic functions are divided into three categories: controlled functions, monitored functions, and shared functions. The dynamic functions that can only be updated and read through the machine itself is called **controlled functions**. While when the values of dynamic functions can be read from the external environment and written by the machine's user, then the dynamic functions are known as **monitored functions**. The dynamic **shared functions**, on the other hand, are updated and read by the rules of the machine together with its external environment.

- **Signature**

A signature is a fixed set of names related to functions and domains that must be declared. There is a finite number of arity for each function name. The Boolean function's values: *True* and *false*, and under-defined function: *undef* represent 0-ary (zero arity) functions that are presumed in the signature.

When creating an ASM, the signature is defined. For example, the signature of the ATD example is shown in figure 2.2. In this figure, the listed domains: *Availability, MotionStatus, PositionStatus, DoorStatus*, and *Status*, reflect the availability situation of both an emergency and obstacle: *existing* or *not*, the state of train's motion: the train is in *moving* or *stopped* situation, the possible set of train's position: the train is *aligned* with the platform or *not*, the potential set of the door's states: the door is *opened,* the door is *closed*, the door is *opening*, and the door is *closing*, and the feasible operational situation of the controller: *sensing* information or *implementing* a command.

The functions: *obstacleSensor, emergencySensor trainMotionSensor*, and *trainPositionSensor* represent monitored predicates for denoting the presence of an obstacle and an emergency in the external environment, as well as the input of the environment that relates to the train's motion and position, respectively. The controlled functions: *state*, *emergency*, *doorStatus*, *obstacleStatus*, *trainPosition, and trainMotion*, indicate the mode of the operation, the emergency, and the up to date state of the door, the obstacle's state at the doorway, the position of the train and its motion, respectively. The 0-ary functions which are *sensing* and *executing* are employed to refer to an atomic object that is a member of the *Status* Domain. The derived function: *safeSituation* returns *true* value; in case that the door is opened or it is opening in an emergency situation.

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

Figure 2.2: The signature of the automated train door example

- **State and Set of Update**

A *state* $\mathfrak{S}$ for a signature $\sum$ is an arithmetical structure incorporating: the superdomain $\mathfrak{D}$, together with the evaluation of terms and formulas defined for every function name assumed in $\sum$. An update of $\mathfrak{S}$ is manifested as the below form:

(*loc*, *v*)

where *loc* is a state's location, and *v* is an item from the D. A collection of updates is called an ***update set***.

- **Rules**

In ASMs, a rule (also called transition rule) constitutes an update set on transition states to characterize the behavior of the system. A rule can be either one of the basic rules listed below, or a complex rule made up of several basic rules. The following are the basic TR transition rules:

- **Skip**: It results in an update set that is empty.
- **Updating ($f(t_1, \dots, t_n) := t$)**: the task of this rule is updating the value the function $f(t_1, \dots, t_n)$ to the value t, such that $t_1, \dots, t_n$, and t are terms of first-order calculus.
- **Block (*M par N*)**: It simultaneously evaluates *M* and *N* rules and causes unified update sets calculated by *M* and *N*.
- **Sequential (*M seq N*):** It implements the rules *M* and *N* in a successive style, begining with *M*.
- **Conditional (*if Q then M otherwise N*)**: It examines whether a Boolean expression, which is *Q*, has a true value, then it implements the *M* rule, otherwise it implements *N* rule.
- **Let (*let x=t in M*):** It assigns the *t* value to a logical variable *x*, and implements *M*. The produced update set is the set calculated by *M*.
- **Forall *(forall x with Q do M)***: It simultaneously implements the *M* rule for each *x* satisfying *Q*.
- **Choose (*choose x with Q do M ifnone N*)**: It selects *x* that meets a given condition *Q* and then implements the *M* rule. However, it just implements *N* when there is no x satisfying *Q*.
- **Call ($r(t_1, \dots, t_n)$)**: It implements the rule r with its presented parameters $t_1, \dots, t_n$, such that this rule is specified earlier.

The machine has also a rule with zero arity, called main rule, that illustrates only one step of ASM. This rule is executed iteratively. Perhaps, a main rule

fails or soundly terminates when there is no new update set is yielded or when no rule is applicable.

To illustrate the above rules, the ATD example is used for this purpose. This example has one main rule and seven descriptive complex rules. From the complex rules, four rules are dedicated to update the state of the door, i.e., *doorState*, that is *CLOSED* at the start. These rules are: *ClosingToClosedOrOpened, OpenedToClosing, ClosedToOpening , and OpeningToOpened*. In addition three out of the seven complex rules are allocated to receive the sensors' data. These rules include: ObstacleSensing, TrainMotionSensing, and EmergencySensing. While he main rule, which has the name: TrainDoorController, is specified to capture the ATD's behavior.

Figure 2.3 lists the main rule for ATD example, see Figure 2.3. In this rule, in case that the current state of the ATD is *SENSING*, then the three dedicated rules for sensing information are called; Otherwise, the ATD system is in EXECUTING state. At this state the emergency must first be examined to handle it through opening the door. When there is no emergency, the four rules that are modeled to change the door's state are called in parallel.

mechanism of ASMs by default [9].

From the four rules, only the ClosedToOpening rule is displayed as it similar to the rest rules, see figure 2.4. In this rule, the controller first examines whether the door's state is CLOSED and the train motion's state is STOPPED and its position is ALIGNED with the platform, then the door state is updated to OPENING.

The controller updates the state of the door into OPENED, if it is at OPENING state.

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

Figure 2.3 Train Door Controller's main rule

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

Figure 2.4: The Closed_To_Opening rule for program Train Door Controller

- **Machine Step and Machine Run**

The process of simultaneously firing or executing, firing all transition rules in a given state to produce the next state is referred to as a computation step for a machine. While a run is a finite or infinite series of consecutive states that result from the execution of steps.

2.3. 2 The Second Concept: Ground model

A ground model concept represents a conceptual model for capturing informal system requirements in a precise, succinct, pliable, and understandable manner. The ground model can be created graphically by carrying out *control state ASMs*. This state is an ASM that has rules are diagrammatically and textually shown in Figure 2.5, such that the machine stays at a given control state *i*, if the $condition_j$ is not met.

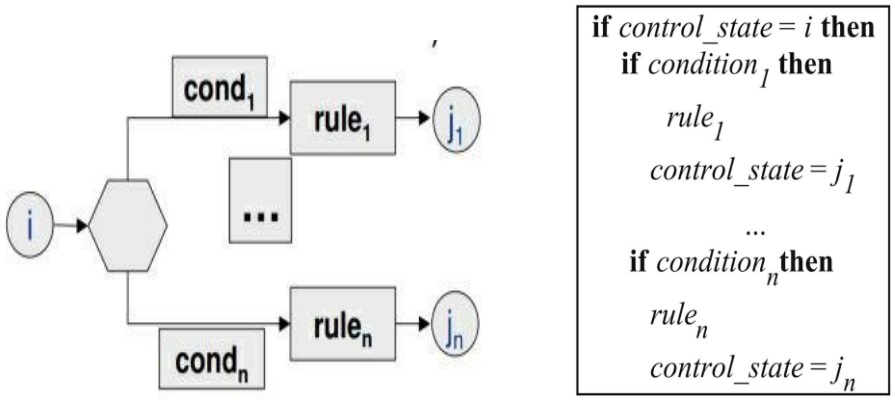

Figure 2.5:Control state ASMs

2.3.3 The Third Concept: Stepwise Refinement

Stepwise Refinement is a method of consecutive refinement that allows a designer to gain a more elaborated model from a brief one. This can be accomplished by fine-tuning (refining) the signature, the flow of operation, or both. At each refining step, the gained model must be proven right in relation to the previous upper one, i.e., the more refined model executes the functionality of the abstract model and its own as well, while maintaining the main system's attributes, such as safety. The ground mode, i.e., the first abstract model, is similar to the system's description perspective (usually informal description), whereas the final refined model is similar to the programmer's perspective (executable code).This incremental change in system design allows for the detection of silent assumptions and ambiguities in the system's requirements during passing abstraction levels.

2.3.4 Distributed ASMs

The basic ASMs are extended into a Distributed ASMs (DASM) in order to encompass the formalization of multiple agents that use a synchronous and asynchronous way for acting and reacting [11].

With synchronous DASM, each agent runs their own machine in parallel and synchronously, using a global clock for the system, and every agent participates to the global states via the union of each signature related to each machine. The synchronous machine's runs are a sequence of steps that are completely ordered.

Asynchronous DASM presents a united global system perspective for coincident successive computations of each agent per se, such that each agent has its own speed to run its basic ASM in its state without the need for a global clock. In DASM, a computation step performed by a single agent is called move (rather than step. A run for DASM is a partial order set of agent's moves that are partially ordered.

2.3.5 An Executed ASM Tool: ASMETA Framework

Various tools have been developed around ASMs over the last two decades with the aim of providing executable models and underpining the analysis tasks, including: simulation and verification. From all the accessible tools, the ASMETA framework was selected for the context of this project. ASMETA, or the ASM mETAmodelling Framework, is one of the most inclusive modeling and analysis tools designed for ASMs [1, 23]. Recognizing the shortcomings of existing ASM tools, such as restricted coverage of aspects during the complete development process, expressing ASMs into a language accompanied by syntax that is strictly based on the execution environment, and the inapplicability of unifying ASM tools, the

ASMETA framework was created to provide ASMS with an unchanging notation and interoperable tools based environment.

The Java programming language is used to perform ASMETA. On the website [8], the ASMETA framework is available for educational use.

Various tools are included in the ASMETA framework, such as the ASMETA Language (AsmetaL) that has a concrete syntax for ASMs. In addition, AsmetaL is notated in a way which is very similar to both basic ASM and multi-agents ASM [11]. AsmetaL includes a Standard Library, a set of predeclared ASM domains (Integer, Boolean, String, etc.) and different functions declared on those domains, as well as an AsmetaLc compiler for compiling and translating AsmetaL into specifications that can be implemented by the simulator.

AsmetaL has four sections: the first section is called header that is dedicated for importing the Standard Library and for specifying domains and functions; the second section is known as a body that is devoted for inserting the textual notation of static concrete domain, derived and static functions, and the rules; the third notation is the main rule; and the final section is the initialization for determining the initial values of the controlled functions. For instance, see the AsmetaL specification for the ATD example in Code 2.1.

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

Code 2.1: AsmetaL specification of the ATD example

Furthermore the ASMETA framework has another tool called ASMETA Simulator (AsmetaS) for building a run of the specification that need to be simulated [2]. The AsmetaS underpins different

useful tasks across the simulation, such as consistency inspecting in order to find any inconsistent updates, arbitrary simulation for randomly simulating the specification, where the simulator by itself selects the input values in a random way.

CHAPTER THREE

THE PROPOSED METHOD

3.1 Introduction

This chapter presents a methodology for elucidating several ambiguities existing in the informal requirements specified for the SASL framework.

3.2 The Developed Methodology

We develop a methodology that elucidates the main ambiguities starting with the RFC document to catch its informal explanation, through the ground model concept of ASMs, then it moves to explicating the potential uncertain parts depending on additional document sources by applying ASM refinement concept. The essential aim of this methodology is to accurately clarify the following behavioral parts of SASL: how the server reports the mechanisms that is available to use; how the client picks its preferable mechanism; how the client decide the maximum size related to the cipher buffer; at which time and how the negotiation of the security layer is performed.

For the reason that the SASL has an extensive and complicated behavior, we divide the SASL framework into three stages: the mechanism negotiation stage, the authentication negotiation stage, and the security layer negotiation stage. At each stage, we will display (if required) the ground model for client and server sides, which is shown via the control state ASM, and then we will direct our action at refining the ground model rules to explicate the discrepancies in the RFCs and Oracle documentation[1].

[1] The full rules for the refined model depending on RFC 2222/4422 are available online at https://doi.org/10.5281/zenodo.1204257, while for the refined model which depend on the interpretation of Oracle implementation documentation are available at https://doi.org/10.5281/zenodo.1204242

AsmetaL is used to express every refined rule. The translation from the control state ASM graphical notation to the AsmetaL specification is achieved via using the mapping illustrated in figure 2.5.

3.2.1 Mechanism Negotiation Stage

Figure 3.1 depicts the client-side ground model for this stage. This figure is a literal translation of RFC 2222/4422. The client initiates this stage by requesting the server for delivering its supported list of mechanisms. Whenever the client receives this list, the guard that is entitled with "At least one mechanism in the list is supported" examines whether the client permits any mechanism in the received list. If this is the case, the client chooses a suitable mechanism from the server's list and enters the final state, which is "Sending an authentication request"; Otherwise, an abort response is sent by the client and it waits for an abort reply from the server. When the client receives the abort reply, it terminates this stage by accessing the "Abort" state.

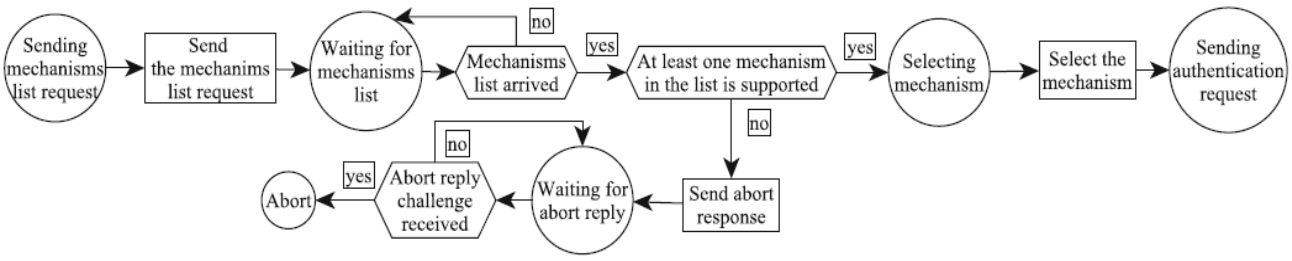

Figure 3.1 Ground model of the client side for mechanism selection stage

Figure 3.2 Ground model of the server side for mechanism selection stage

The key steps performed by the server for this stage are scheduled in the ground model shown in Figure 3.2. The server initially remains in the "Waiting for mechanisms list request" state till it receives a client's list request. At that time, the server pulls up a list of accessible mechanisms to provide it to the client. The server then moves to the last state of this stage, which is "Waiting for authentication requests", after transmitting this list.

From Figure 3.1, it is unclear how the client selects the preferred mechanism. RFC 4422 [20] states that the client is responsible for selecting its preferable mechanism. The ‘’r_selectmech’’ rule provided in Code 3.1 (a) specifies this. The mechanism selection in this code is performed via the monitored function "insertMechanism", i.e., it is achieved in an interactive way with the client. Any mechanism from the "arrivedMechList" set should be chosen. In “selMech”, an arbitrary selected mechanism is saved.

```
rule r_selectmech=
  if contains(arrivedMechList, insertMechanism) then
       selMech:=insertMechanism
  endif
```

(a) The r_selectmech rule according to RFC 4422

```
function mHasGreatestSSF($m in Mechanisms, $c in Client)=
 forall $x in arrivedMechList with
   (($x!=$m) and allin(policies($x), policies($c)) implies
                                          ssf($m)>=ssf($x))
rule r_selectmech=
  choose $m in arrivedMechList with
     allin(policies($m), policies(self)) and
     mHasGreatestSSF($m, self)=true do
         selMech:=$m
```

(b) The refined r_selectmech rule according to Oracle implementation document

Code 3.1: The r_selectmech rule

In contrast, according to the Oracle implementation documentation [11], the best mechanism is chosen by the client based on the maximum SSF of mechanism and the security policy of client. It is possible to elaborate this explanation further by modifying the rule in code 3.1(a)

into the one presented in Code 3.1(b). We expanded the modeling vocabulary in the revised rule.

More precisely, let the “ssf($m)” function represent the *SSF* value for each mechanism “$m” belonging to the Mechanisms domain, “policies($m)” represent a set of the security policies established for each mechanism “$m”, and “policies(self)” represent the set of security policies dedicated for the client. The client agent interprets the 0-ary function “self” as itself. Every policies set is one or more items from the Policies domain which includes: {NOACTIVE, NOPLAINTEXT, NOANONYMOUS, NODICTIONARY, MUTUALAUTH}. If the selected mechanism has the highest *SSF* value, the “mHasGreatestSSF” function returns true value. The refined rule selects the best mechanism from the arrived server's list which is represented by the function: “arrivedMechList”, such that this selection must ensure that the security policies set of the selected mechanism contains all the elements belonging to the client's set and the SSF value of the selected mechanism is greater than all the *SSF* values of the mechanisms whose sets contain the client's security policies set.

On the server side, as shown in figure 3.2, obtaining the available mechanisms list has to be clarified. RFC 4422 [20] specifies that the server just announces the list of the available mechanisms. The “r_getmechs” rule in code 3.2 (a) specifies this specification. In this code, let “mList($c, self)” be a collection of the advertised mechanisms that will be delivered to the “$c” client. One or more SASL mechanisms which are available for server use are included in the “saslmechs(self)” set. In the “r_getmechs” rule, the server will simply copy all the items in

the “saslmechs(self)” set and give them to the “mList($c, self)”, which is initially an empty set, to represent the advertised mechanisms' list.

```
rule r_getmechs($c in Client)=
  mList($c,self):=saslmechs(self)
```

(a) The r_getmechs rule according to RFC 4422

```
rule r_getmechs($c in Client)=
  let ($i=0) in
    while $i<size(saslmechs(self)) do
      seq
       let($m=at(asSequence(saslmechs(self)), iton($i))) in
         if (exist $p in policies(self) with
             contains(policies($m), $p)=true) then
               mList($c,self):=including(mList($c,self),$m)
         endif
       endlet
       $i:=$i+1
      endseq
  endlet
```

(b) The refined r_getmechs rule according to Oracle implementation document

Code 3.2: The r_getmechs rule

The Oracle implementation documentation [21], on the other hand, describes getting a list of available mechanisms as "The server can call sasl_listmech() to get a list of the available SASL mechanisms that satisfy the security policy”. It is unclear from this quoted statement whether there is a particular policy for SASL mechanisms and what is the meaning of that satisfy the security policy. According to Oracle's Java security guide [22], each SASL mechanism has a unique policy set, for example the {NOPLAINTEXT, NOACTIVE, NODICTIONARY}, { NONANONYMOUS, NOPLAINTEXT }, and {NONANONYMOUS}, for the EXTERNAL, DIGEST-MD5, and PLAIN mechanisms, respectively.

In order to obtain a better grasp of the actual meaning of "satisfy the security policy", we inspect the server’s response which is "sending the available mechanisms’ list to the client" in several SASL mechanism

instances. For example, in the EXTERNAL mechanism example [11], the server delivers the { EXTERNAL, DIGEST-MD5}. This means that the NOPLAINTEXT policy, which is underpinned by the server, is shared by all of the mechanisms in this list. The server also transmits, in the DIGEST-MD5 mechanism example [24], the {DIGEST-MD5, PLAIN} list. We can see that the NONANONYMOUS policy is shared by both mechanisms. This indicates that the server supports the NONANONYMOUS policy and it sends the mechanisms that comply with it. As a result, the "r_getmechs" rule presented in code 3. 2(a) can be re-elaborated into the rule in code 3.2 (b).

In the revised "r_getmechs" rule, the server obtains a mechanism for its use from "saslmechs", if the policy set defined for this mechanism contains a policy belonging to the server's policies set. In other words, at least one server's policy is included in the policy set specified for each mechanism in the advertised mechanisms list.

3.2.2 Authentication Negotiation Stage

In SASL, this is the longest stage. We do not present all the ground model for this stage due to space constraints[2] becouse we are focussion on just ambigouse part.

One understated aspect of this stage, is the negotiation over utilizing the security layer and the maximum size of the cipher buffer.

In RFC 4422 [11], it was specified that when the chosen mechanism underpins a security layer, then a negotiation about utilizing this layer must be conducted.

[2] All ground models are available online at https://doi.org/10.5281/zenodo.1200216

More over, the way of conducting this negotiation is not described. RFC 2222 [15], though, specifies this as the exchange of a bit-mask which defines the security layer level as follows: 1: no security layer, 2: integrity, and 4: privacy. The given bit-mask defines the undeclared privacy service while ignores the confidentiality service.

However, instead of a bit-mask, the SSF indicator, which has the following values: 0: authentication, 1: authentication and integrity, and >1: authentication, integrity, confidentiality, and key length, is used in the Oracle implementation documentation [11]. Despite providing this indicator, how the client and server agree on the employing the security layer is unclear.

Depending on the Java security guide presented by Oracle [22], if the selected mechanism's SSF value is higher than or equal to 1, then the server sends its underpinned **Quality of Protection (QOP)** list, which contains one or more items from the following: **auth**: *authentication*, **auth-int**: *authentication and integrity*, and **auth-conf**: *authentication, integrity, and confidentiality*. After that, the client then chooses a protection value from the server's sent list based on its SSF value and transmits it to the server. To store the session SSF value, which is identical to the client's protection value, the server validates that the client's protection value is belonging to its list. The agreed-upon security layer service is represented by the SSF value. However, there is little information in this document regarding how the client and server set the maximum buffer size; in case that they agree to use the security layer service.

As reported by the DIGEST-MD5 SASL mechanism example [24], when the server delivers its desired maximum buffer size, then the client

will examine whether there is a value of buffer size in the received challenge. If so, the client will calculate the buffer size for this session by subtracting 16 bytes from the minimum value of the received size and the client's underpinned one. If the size does not exist, the client will use the default buffer size of 65536. Following this explanation, code 3.3 details how the client decides the maximum buffer size.

```
if contains(receivCh(self), "maxbuf")=true then
  choose $max in Maxbuf with
  eq(at(receivCh(self), iton(indexOf(receivCh(self), "maxbuf")+1)),
  toString($max)) do
      if $max<maxBuf(self) then
         maxBufDetermined:=$max-16
      else
         maxBufDetermined:=maxBuf(self)-16
      endif
else
   maxBufDetermined:=65520
endif
```

Code 3.3: Determining the maximum cipher buffer size by the client

In code 3, the "receivCh(self)" is a sequence of strings that acts as the server's received challenge, integer domain "Maxbuf" comprises the following potential buffer sizes: 65535, 131071, 262143, 16777215, and “maxBuf(self)” is the maximum buffer size supported by the client. First, the client determines if the "receivCh(self)" includes the server's maximum buffer size, then the client calculates it, or sets it to the default value. As “receivCh(self)” is a series of strings, the client picks an integer value from the “Maxbuf” domain, such that the picked value is equal to a string value included in the “receivCh(self)”. The buffer size is then determined by comparing the selected value to the client's buffer size, which is kept in “maxBufDetermined”.

3.2.3 Security Layer Negotiation Stage

This stage is a non-mandatory stage. Conducting this stage is contingent on the negotiation in the aforementioned stage. This negotiation entails interchanging a bit-mask in accordance with RFC 2222 [14], as well as

exchanging SSF values in accordance with Oracle documentation [11]. Because the bit-mask does not describe the confidentiality service, we rely on the Oracle implementation documentation [11] together with RFC 4422 [25] to specify this stage. In addition, we depend on the RFC 2831 for the DIGEST-MD5 SASL mechanism [24] to specify the confidentiality and integrity secured messages, because the RFC 2222/4422 and the Oracle implementation documents do not provide description for this specification. We annotate the key details for modeling this stage in figure 3.3.

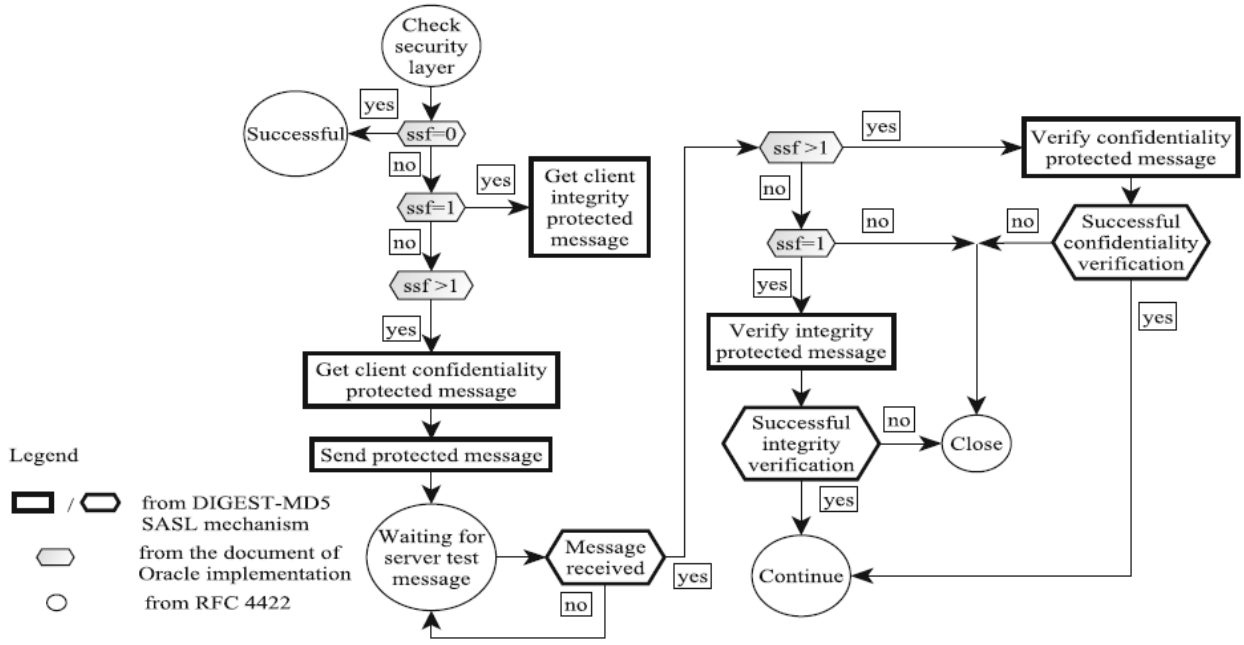

Figure 3.3 Ground model of security layer negotiation phase – Client side

The ASM ground model for client-side security layer service negotiation is depicted in figure 3.3. The client begins this phase by verifying the SSF value that both the client and the server agreed upon during the authentication phase. If this value is 0, the client will arrive at the successful state. This state shows that the client has been successfully authenticated and that there is no security layer in place. If the SSF value

is one, the next protocol messages must be integrity protected. As a result, the flow proceeds to implement the "Get client integrity protected message", which returns a test message with the computed Message Authentication Code (MAC) for the message itself [24]. If the SSF value is larger than one, the confidentiality protection is provided for the next protocol messages. Therefore the client executes the "Get client confidentially protected message" to encrypt a test message with the computed MAC[24].

The client then transmits the encrypted message to the server, changing its status to "Waiting for server test message". The client will verify the agreed SSF value whenever it gets a protected message from the server. When this value exceeds one, the client will implement a confidentiality validation(decrypting the message, computing the MAC and comparing it with the received one). If the SSF value is 1, the client will verify the integrity of the data (computing the MAC and comparing it with the received one). If the verification is successful, the client is moved to the final state "Continue", which implies that the client can continue to communicate with the server following SASL. When the verification fails, the client closes the connection with the server and changes its state to close.

For this stage, we do not present the server ground model because it is close to the client's model, except that the server waits for a protected message from the client before delivering its own message.

CHAPTER FOUR

RESULTS AND DISCUSSION

4.1 Introduction

This chapter presents our results in relation to elucidating the key ambiguities in the SASL case study's informal requirements.

4.2 Results and Discussion

The essential aim of this project is to elucidate the vague descriptions in SASL using ASMs. Our proposed methodology initiates by reflecting the textual explanation in RFCs using the ground model concept, and then uses the step refinement concept to re-elaborate this explanation depending on additional document sources. The key ambiguities that have been researched are outlined in table 4.1, together with the source documents for both the ambiguity and its formal elucidated specification.

No.	**Ambiguity**	**Document Source of Ambiguity**	**The elucidated Specification**	**The document source for elucidation**
1	The client chooses the its preferable mechanism	RFC 4422 [20]	Code 3.1 (b)	Oracle implementation document [11]
2	The server announces the available mechanisms list	RFC 4422 [20], and Oracle implementation document [11]	Code 2 (b)	DIGEST-MD5 SASL mechanism [24], Oracle implementation document [11], and its Java security guide [24]
3	Determining the maximum size of the cipher text buffer	RFC 4422 [20], and Oracle implementation document [11]	Code 3.3	DIGEST-MD5 SASL mechanism [16]
4	How and when the security layer is negotiated	RFC 2222 [15]	Ground model in Figure 3.3	Oracle implementation document [11], DIGEST-MD5 SASL mechanism [24], and RFC 4422 [20]

Table 4.1 The document source for each ambiguity and its formal elucidated specification

Table 4.1 shows the following:

(1) RFC 4422 [20] is unclear about selecting the optimal mechanism because it just specifies that the client chooses the optimal one. We try to explicate this using the Oracle implementation description [11], which indicates that the client chooses the optimal mechanism with the highest SSF, and in accordance with its security policy.

(2) Both RFC 4422 and RFC 2222 which simply states that the server advertises the list, and the Oracle implementation, which claims that the server advertises the mechanisms that meet the security policy, are ambiguous. Based on the document sources presented in table 4.1, we turn the informal description of Oracle into a formal one. We deduce that meeting the security policy necessitates that every mechanism in the announced list supporting at least one server's policy.

(3) In RFC 4422 [20] and the Oracle implementation, deciding the maximum buffer size is under-stated. We use the explanation provided by the DIGEST-MD5 SASL method [24] to explicate this.

(4) The utilizing of security layer in RFC 2222 requires additional explanation, since it specifies that useing this layer is constrained with the specified bit-mask, which ignores the confidentiality service. As a result, we depend on the Oracle implementation, which use the SSF rather than a bit-mask to indicate when to use this layer. To illustrate how the client and server negotiate this layer, we use the DIGEST-MD5 SASL mechanism [24].

This study demonstrates how the ASM formalism, particularly with its both a ground model and refinement concept, is efficient at elucidating the ambiguity. The informal specification can be captured in a comprehensible manner and at the required amount of details using the

ground model. Then, through the incremental refinement, it can be transformed into a precise and improved mathematical specification.

The temporal aspects of SASL are not considered in our ASM design because neither RFC 2222/4422 nor Oracle implementation papers provide a specification for that.

CHAPTER FIVE

CONCLUSIONS AND FUTURE DIRECTIONS

5.1 Conclusion

We have presented the ASM specification to clarify ambiguities in SASL case study. Our attention has been paid to the vague parts in RFC 2222/4422 and Oracle implementation papers, such as selecting optimal mechanism, sending a list of mechanisms, specifys when and how the security layer can be utilized, and establishing the maximum cipher buffer size.

We have manifested how to produce an understandable specification using two ASM concepts: a ground model and progressive refining. The ground model is able to represent the required behavior, which is specified in RFCs, through a comprehensible fashion using the ground model. While the incremental refinement assisted us in accurately elucidating the unclear section of the intended behavior, relying on additional document sources to guide us.

We formalize the informal specification by stating it in the ASM formalism, which is mathematically well-founded, accurate, and straightforward.

5.2 Future Directions

To extend our research, we are intending to investigate the security aspects of SASL to demonstrate whether our formal specification is secured. We are planningto use an acceptable security analysis technique to extract the SASL's security requirements, and then validate them at the verification level.

References

[1] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Model-driven language engineering: The ASMETA case study. In International Conference on Software Engineering Advances, pages 373–378. IEEE, 2008.

[2] Angelo Michele Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A Metamodel-based language and a simulation engine for Abstract State Machines. Journal of Universal Computer Science, 14(12):1949–1983, 2008.

[3] Arcaini, P., Holom, R.M., Riccobene, E.: ASM-based formal design of an adaptivity component for a Cloud system. Formal Aspects Comput. 28(4), 567–595 (2016).

[4] Bella, G., Riccobene, E.: Formal analysis of the Kerberos authentication system. J. Univers. Comput. Sci. 3(12), 1337–1381 (1997).

[5] Bishop, S., Fairbairn, M., Norrish, M., Sewell, P., Smith, M., Wansbrough, K.: Engineering with logic: HOL specification and symbolic-evaluation testing for TCP implementations, pp. 55–66. ACM Press (2006) .

[6] Bradley Fordham, Serge Abiteboul, and Yelena Yesha. Evolving databases: An application to electronic commerce. In International Database Engineering and Applications Sym(1997).

[7] Chelemen, R.M.: Modeling a web application for cloud content adaptation with ASMs. In: International Conference on Cloud Computing and Big Data (CloudCom-Asia), pp. 44–55. IEEE (2013).

[8] Edmund M Clarke and Jeannette M Wing. Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR), 28(4):626–643, 1996.

[9] Egon B¨orger and Alexander Raschke. Modeling Companion for Software Practitioners. Springer, Heidelberg, 2018.

[10] Egon B¨orger and Dean Rosenzweig. A mathematical definition of full Prolog. Science of Computer Programming, 24(3):249–286, 1995.

[11] Egon B¨orger and Robert St¨ark. Abstract State Machines: A method for high-level system design and analysis. Springer, Heidelberg, 2003.

[12] Egon B¨orger and Wolfram Schulte. A programmer friendly modular definition of the semantics of Java. In Formal Syntax and Semantics of Java, pages 353–404. Springer, 1999.

[13] Giampaolo Bella and Elvinia Riccobene. A realistic environment for crypto-protocol analyses by ASMs. In Workshop on Abstract State Machines, pages 127–138, 1998.

[14] Giampaolo Bella and Elvinia Riccobene. Formal analysis of the Kerberos authentication system. Journal of Universal Computer Science, 3(12):1337–1381, 1997.

[15] John Myers. Simple Authentication and Security Layer (SASL). RFC 2222, 1997. http://www.rfc-editor.org/rfc/rfc2222.txt.

[16] John Thomas and Nancy Leveson. Performing hazard analysis on complex, softwareand human-intensive systems. In 29th International System Safety Conference. International System Safety Society Unionville, VA, 2011.

[17] K Preston White and Ricki G Ingalls. Introduction to simulation. In Winter Simulation Conference (WSC), pages 1741–1755.

[18] K. Zeilenga. The PLAIN Simple Authentication and Security Layer (SASL) Mechanism. RFC 4616, 2006.

[19] Kevin Compton, Yuri Gurevich, James Huggins, and Wuwei Shen. An automatic verification tool for UML. Univ. of Michigan, EECS Dept. Tech. Report CSE-TR423, 2000.

[20] Melnikov,A., and K. Zeilenga. Simple Authentication and Security Layer (SASL). RFC 4422, 2006.

[21] Oracle. Writing applications that use SASL. In Developer's Guide to Oracle Solaris R 11 Security, chapter 7, pages 126–148. Oracle, 2012.

[22] Oracle: Java SASL API Programming and Deployment Guide. In: Java Platform, Standard Edition Security Developers Guide, chap. 10, pp. 21–28. Oracle (2016).

[23] Paolo Arcaini, Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A model-driven process for engineering a toolset for a formal method. Software: Practice and Experience, 41(2):155–166, 2011.

[24] Paul Leach, Chris Newman, and A. Melnikov. Using Digest Authentication as a SASL Mechanism. RFC 2831, 2000.

[25] R. Siemborski and A. Gulbrandsen. IMAP Extension for Simple Authentication and Security Layer (SASL) Initial Client Response. RFC 4959, 2007.

[26] R. Siemborski and A. Melnikov. SMTP Service Extension for Authentication Initial Client Response. RFC 4954, 2007.

[27] Rosenzweig, D., Runje, D., Slani, N.: Privacy, abstract encryption and protocols: an ASM model - part I. In: B¨orger, E., Gargantini, A., Riccobene, E. (eds.) ASM 2003. LNCS, vol. 2589, pp. 372–390. Springer, Heidelberg (2003). https://doi.org/10.1007/3-540-36498-6 22.

[28] Rui Xue and Deng-Guo Feng. New semantic model for authentication protocols in ASMs. Journal of Computer Science and Technology, 19(4):555–563, 2004.

[29] Steven C Cater and James K Huggins. An ASM dynamic semantics for standard ML. In International Workshop on Abstract State Machines, pages 203–222. Springer, 2000.

[30] Varsha Awhad and Charles Wallace. A unified formal specification and analysis of the new Java memory models. In International Workshop on Abstract State Machines, pages 166–185. Springer, 2003.

[31] Wolf Zimmermann and Thilo Gaul. On the construction of correct compiler backends: An ASM-approach. Journal of Universal Computer Science, 3(5):504–567, 1997.

[32] Yuri Gurevich. Evolving algebras 1993: Lipari guide. Oxford University Press, 324:9–36, 1995.

[33] Yuri Gurevich. May 1997 draft of the ASM guide. 1997.

الخلاصة

طبقة المصادقة والأمان البسيطة (SASL) ، كما يوحي اسمها، إطار عمل للسماح لبروتوكولات الإنترنت بدعم خدمات الأمان، بما في ذلك المصادقة والتحقق اختياريا من السلامة والسرية. تم تعريف SASL لأول مرة في RFC 2222، ثم تمت تحديثه في RFC 4422، وكلاهما استخدما اللغة الطبيعية لتحديد مواصفات SASL. ومع ذلك، فإن اللغة الطبيعية تميل عادة إلى تفسيرات غير سليمة بسبب الغموض المتأصل بها وكذلك عدم الدقة. على الرغم من أن أوراكل لديها تنفيذ لـ SASL ، لكن وثائقها تحتوي أيضا على مواصفات RFC غير محددة وأوصاف غير رسمية. باستخدام آلات الحالة المجردة (ASMs) ، يركز هذا البحث على توضيح الغموض في SASL. ويستند هذا التوضيح على اثنين من المفاهيم الرئيسية لـ ASM: مفهوم نموذج الأرض الذي يلتقط بإيجاز السلوك الأصلي لـ SASL ومفهوم التنقيح الذي يوضح بدقة الجوانب الغامضة للسلوك. كما نعرض بعض الاختلافات بين RFCs ووصف تنفيذ أوراكل. ونعتقد بأن النتائج التي توصلنا إليها يمكن أن تستخدم كنقطة انطلاق لتنفيذ وتحليل رسمي اضافيان.

وزارة التعليم العالي والبحث العلمي

جامعه بابل - كليه العلوم للبنات

قسم علوم البنات

توضيح غموض طبقة المصادقة والأمان البسيطة

بحث مقدم الى

كليه العلوم للبنات،جامعه بابل

جزءا من متطلبات نيل درجه الدبلوم العالي في العلوم /علوم الحاسوب

مقدمه من قبل

جنان جاسم حسين طعيس

بإشراف

د. فرح الشريفي