Master's Thesis

Formal and Experimental Verification of Robot Control Protocols for Smart Buildings

WU Jingting

Supervisor Assoc. Prof. Razvan Beuran

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

March, 2025

Abstract

With the increasing complexity of IoT systems, assuring the IoT system trustworthiness has become a critical work. Based on the IoT System Trustworthiness Levels (TALs) proposed by Beuran, this thesis reports about a case study to ensure that robot control protocols of a smart building meet high trustworthiness levels through formal and experimental verification.

The target smart building for this study has four subsystems, namely the robot subsystem, Building OS, the robot control platform subsystem, and the robot subsystem. The target protocol is implemented on the robot control platform subsystem, with the function to command a robot to move to another floor through the elevator.

Our formal verifications focus on model checking. We specified the basic version of the protocol for single robot control and the improved version for multi robots control in Maude, and successfully checked deadlock by using Maude's search command and checked safety and liveness properties by using Maude LTL Model Checker. All checked properties satisfy the requirements, which let us confirm the process correctness of the protocols.

Our experimental verifications focus on emulation and fuzzing, identifying unexpected problems. We developed an emulator for emulating the communication and action of the four subsystems. The emulator outputs communication and action logs, all of which can help us to diagnose problems when the system has any abnormal behaviors. Based on the emulator, we conducted fuzzing, an automatic test method by generating a large amount of random data. Our fuzzing involves mutating messages related to the robot control protocols in each subsystem, aiming to affect message transmission. In order to speed up the fuzzing, we applied some strategies. By applying our strategies, although the improvement of the three subsystems of robots, building OS and elevators is not obvious, the coverage of the fuzzing of 30 seeds of the robot control platform subsystem has increased from 36% to 94%. Based on the fuzzing results, we summarized 10 problems that had not been found in model checking and divided them into 6 categories according to Common Weakness Enumeration (CWE). Part of the problems comes from

mistakes such as poor consideration in programming, and the other comes from abnormal situations such as data tampering. The discovery of these problems can be a supplement to the model checking.

To correct the problems, we modified the old model, got five new formal models according to the ten problems, and verified the properties again according to the requirements. At the same time, we conducted experimental verification of the modified emulator again and observed the behaviors again to confirm that our solution is effective.

The results of our experiment not only show the effective but also show that the two methods have a certain complementarity. During the model checking, some operations were omitted in the system abstraction, while such operations are not omitted in the emulator, which is a good auxiliary supplement for the model checking; the characteristics of random mutation in the fuzzing are difficult to cover all paths, while the characteristics of all reachable paths traversal in the model checking prove the correctness of the paths.

On the other hand, state is used in emulating the operation of the system and devices, applied to both modeling in formal verification and emulator development in experimental verification. The problems located by one party can be easily located in the other party. Such convenience is reflected in later modification and re-verification. With this process of formal verification, experimental verification, analysis, modification, re-verification and re-analysis, we believe that our method is effective in high trustworthiness levels assurance.

Keywords: Control Protocol, Smart Building, Model Checking, Emulation, Fuzzing.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Background

With the development of Society 5.0, the Internet of Things (IoT), made of devices which can connect to the network, becomes more and more common. A variety of IoT devices can be interconnected to integrate into a complex system that provides various services. Smart buildings are the buildings that adopt and implement such the systems. As a consequence of the integration, smart buildings can provide varieties of services, such as robot controls, HVAC controls, energy controls and more. However, while enjoying convenient services, we should also realize that whether this system is trustworthy needs to be assured.

To accomplish integration between subsystems within a smart building, a common approach is that the building assets transmit messages through communication protocols, and responses through control protocols according to the message. According to Christopher et al. [1], most attacks are against communication protocols and devices, and such attacks all affect the message transmission. Besides, due to many control protocols customized according to user needs, it may lead to abnormal operations [2]. Thus, as trustworthiness assurance for a smart building, it is necessary not only to work on preventing attacks, but also to consider whether the operation of the control protocol is reliable and how the attacks will affect the operation of the control protocol.

Beuran et al. [3] classified IoT System Trustworthiness Assurance Levels (TALs) into three categories, defined together with the appropriate verification methods as follows: TAL1: Checklist regarding regulations, TAL2: Experimental verification and TAL3: Formal and experimental verification. For such a complex system as smart buildings, although it is difficult and unnecessary to assure that all parts meet TAL3, it is necessary to assure that

the key parts meet TAL3 and other parts only need to meet TAL2 or TAL1. As the robot control protocol is one of the key parts of a smart building, assuring a high trustworthiness level in the control protocol operation is extremely important.

However, in papers on smart building-related research, the focus is more on the communication protocols and less on control protocols; in control protocol-related research, the discussion is more on whether there will be a collision between different control protocols, and less on the trustworthiness of a single control protocol. In papers on combining formal and experimental verification, many researchers have contributed to the verification of communication and cryptographic protocols by guiding experiment through formal methods [4, 5], different from our objective of a control protocol verification and the requirements which TAL3 require to separate formal verification and experimental verification. In order to fill this gap, we provide a research case and methodology to verify the control protocol.

1.2 Contributions

Our contributions mainly include the following:

- We successfully specified the robot control protocols of a smart building with Maude, and successfully completed the model checking of safety and liveness properties.
- We developed an emulator and conducted fuzzing for the protocols, and summarized the existing problems identified through fuzzing.
- Based on the identified problems, we have proposed some improvement plans and conducted the re-verification to assurance a high trustworthiness levels.
- We discussed the strengths and shortcomings of our methodology and a possible direction for a combination of the model checking and emulation approaches.

1.3 Thesis Structure

The remainder of this thesis is organized as follows. Chapter 2 introduces the basic knowledge of model checking and fuzzing and the tools used. Chapter 3 introduces the methodology. Chapter 4 summarizes and analyzes the existing problems of the protocols through the results. Chapter 5 introduces the

improvements and reverification of the protocols. Chapter 6 discusses the strengths and shortcomings of our method. Chapter 7 summarizes the thesis.

Chapter 2

Preliminaries

2.1 Formal Verification

In this section we discuss various aspects related to formal verification.

2.1.1 Model Checking

Model checking [6] is a kind of formal verification, which has been widely used in protocol verification [7, 8]. By defining the formal specifications, model checker will traverse all paths in state spaces to check the satisfiability of the specified properties. If there is a path does not meet the property, the path will be called a counterexample. If there is no counterexample, it means that the system meets the requirements. If there is a counterexample, diagnose the reason for the counterexample through analyzing the path. Basically, the steps of model checking are as follows:

1. Modeling the system model according to the system
2. Specifying properties according to requirements
3. Conducting the model checking

Regarding the system modeling, we used the Kripke Structure, which will be introduced in Sect. 2.1.2. The model mainly includes state, state transition and atomic state proposition. The atomic state proposition here refers to the propositions such as whether the robot is in a critical section in a certain state.

With regard to the requirements, we mainly focus on three properties: deadlock, safety, and liveness properties.

About the deadlock property, in our modeled system, we did not define a termination state, so we can use Maude's search command to find if there is a state that cannot transition to the next state, and if we find such the state, it means that the protocol probably has a deadlock problem.

The safety property is often described as "nothing bad should happen". In our emulated system, "bad thing" is that multiple robots enter the critical section at the same time.

The liveness property is often described as "something good will eventually happen". It can be imagined that if the system stays in a meaningless loop, the system will neither stop nor be considered as any "bad thing" happening, but it is meaningless. We used Linear Temporal Logic (LTL) to specify the safety and liveness properties, which also will be introduced in Sect. 2.1.2.

2.1.2 Kripke Structure

A labeled Kripke structure $lK \triangleq \langle lS, lI, lE, lP, lL, lT \rangle$, s.t.

- lS: A set of states.
- lI: The set of initial states s.t. $lI \subseteq lS$.
- lE: A set of events $lE \subseteq U$
- lP : A set of atomic state proposition s.t. $lP \subseteq U$ $lE \cap lP = \phi$
- lT: A total ternary relation s.t. $T \subseteq lS \times lE \times lS$.
- lL: A labeling function whose type is $lS \to 2^{lP}$

Because lK cannot be written in a Maude system, we use events-embedded-in-states Kripke structure K_{ees} to simulate the lK: $K_{ees} \triangleq \langle S_{ees}, I_{ees}, P_{ees}, L_{ees}, T_{ees} \rangle$, s.t.

- $S_{ees} = lE \times lS$.
- $I_{ees} = \{(\iota, s)\} \mid s \in lI$.
- $P_{ees} = lP \cup lE$, $L_{ees}((e, s)) = \{e\} \cup lL(s)$ for each $(e, s) \in S_{ees}$.
- $T_{ees} = \{((e, s), (e', s')) \mid e, e' \in lE, s, s' \in lS, (s, e', s') \in lT\}$.

A path π of a K_{ees} is $s_0; ...; s_i; s_{i+1}$ of S s.t. $(s_i, s_{i+1}) \in T_{ees}$ for each i. Let U be a universal set of symbols. Let $\mathcal{P}$ be the set of all paths. Let $\mathcal{K}$ be the set of all Kripke structures.

The formulas φ of linear temporal logic (LTL) for K_{ees} are as follows:

$$l\varphi ::= \top | p | \neg\varphi | \varphi \wedge \varphi | \bigcirc \varphi | \varphi \mathcal{U} \varphi$$

where $p \in P_{ees}$. Let F be the set of all formulas in LTL for K_{ees}. For all $K \in \mathcal{K}$, all $\pi \in \mathcal{P}$ and all $\varphi \in \mathcal{F}$, $K, \pi \models \varphi$ is inductively defined as follows:

- $K_{ees}, \pi \models \top$
- $K_{ees}, \pi \models p$ iff $p \in L(\pi()0)$
- $K_{ees}, \pi \models \neg\varphi$ iff $K, \pi \not\models \varphi$
- $K_{ees}, \pi \models \varphi_1 \wedge \varphi_2$ iff $K_{ees}, \pi \models \varphi_1$ and $K_{ees}, \pi \models \varphi_2$
- $K_{ees}, \pi \models \bigcirc\varphi$ iff $K_{ees,\pi^1} \models \varphi$
- $K_{ees}, \pi \models \varphi_1 \mathcal{U} \varphi_2$ iff there exists i s.t. $K, \pi^i \models \varphi_2$ and for all $j < i$ $K, \pi^j \models \varphi_1$.

- $\bot \triangleq \neg\top$
- $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \varphi_2)$
- $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$
- $\Diamond l\varphi \triangleq \top \mathrm{U}\ \varphi$
- $\Box\varphi \triangleq \neg(\Diamond\neg\varphi)$
- $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \Box(\varphi_1 \Rightarrow \Diamond\varphi_2)$

The symbol $\bigcirc$ is called the next connective. The symbol $\mathcal{U}$ is called the until connective. The symbol $\Diamond$ is called the eventually connective. The symbol $\Box$ is called the always connective. The symbol $\rightsquigarrow$ is called the leads to connective. We will discuss how to model in detail in Chapter 3.

2.1.3 Maude System

Maude is a language and system based on rewriting logic [9]. In the Maude system, the basic units are called modules, which include functional modules and system modules. Functional modules are used for defining equations, while system modules are used for defining rules. In the Maude system, a state is expressed as an associative–commutative collection of name–value pairs, where called observable components [7]. We can use functional modules to define the name and value of a state, and system modules to define state transition rules.

A functional module is like the following:

```
1 fmod ROBOTSTATE is :
2     pr STATE .
3     sort RobotState .
4     subsort RobotState < State .
```

```
5       op ...
6       eq ...
7   endfm
```

Among them, **ROBOTSTATE** is the name of this module. **pr** imports another module. **sort** defines a new type. **subsort** defines the hierarchy of types. **op** stands for operator, and **eq** stands for equations.

A system module is like the following:

```
1   mod PROTOCOL is:
2       pr ...
3       var ...
4       rl ...
5       crl ... if ...
6   endfm
```

Among them, **var** defines some variables used in the rules within this module. **rl** defines an unconditional rewrite rule. **crl** defines a conditional rewrite rule.

We use two kinds of commands for verification. One is **search** command and the other one is **modelCheck** command. **search** command takes two states, returns the path from the first state to the second state. Especially, **search state 1 =>* state 2** can help me find all the paths from **state 1** to **state 2**. If the paths are unbounded, we can't complete the model checking, while **search state 1 =>! state 2** will show that the only canonical final states are allowed, which usually indicates the end state or deadlock state. **modelCheck** command takes a state and an LTL formula, which can help us check if the formula is satisfied.

However, in practice, model checking suffers from the state explosion problem [6], especially for such a complex system as smart buildings; it is difficult to model all the states. Thus, we have to make a trade-off decision between different abstractions when formalizing a system to avoid state explosion, which may cause a lack of some important states. In order to assure high trustworthiness levels, experimentation can not only serve as a comparison, but also as an auxiliary to avoid the lack when writing the system model.

2.2 Experimental Verification

2.2.1 Emulation

Simulation is a method to generate data, which can help us to collect useful data. By generating data, Open-SBS [10], as an open source smart building simulator, can support research in the field of Ambient Intelligence

environments. Our research objectives are different from that. Our focus is on comparing the results of model checking, especially supplementing and improving the possible missing during model checking. Thus, we choose the real communication protocol to emulate the communication process. In this way, we can find potential problems in the process of sending and receiving messages. For the same reason, besides emulation, we also adopt fuzzing in order to find potential problems as many as possible.

2.2.2 Fuzzing

Fuzzing is an auto testing technique that can generate random inputs [11], may lead to abnormal message transmission, which is also helpful for finding potential problems, and has also been widely used in protocol verification. In particular, because they all manifest as abnormal message transmission, protocol fuzzing can be regarded as a simulation of attacks or device failures.

In fuzzing, randomly generating different data is called "mutation". Coverage-based fuzzing is a common technique. By measuring code coverage, the fuzzer generates input that can cover more code as much as possible. This method helps to improve the efficiency of fuzzing. However, most coverage-based fuzzing tools require compiled programs, such as programs written in C/C++ language. However, our emulator is developed in Python, which is an interpreted language that cannot directly output the amount of code like compiled languages. Therefore, we need to use fuzzing tools suitable for Python, such as Atheris [12, 13].

2.2.3 Atheris Fuzzing Tool

We used Google's Atheris to complete the fuzzing. Atheris is actually based on Libfuzzer, which is a famous fuzzer for C/C++. Libfuzzer needs compiled programs, but it also accepts the input of the shared library. Therefore, the way Atheris measures the coverage is to find the part of the code that we want to measure the coverage, and instrument the code. Atheris will transform this part into shared libraries to help Libfuzzer measure the coverage. In this thesis, we only use the **@atheris.instrument_func** decorator to instrument the functions which are regarding the state transition of the protocols, and all of the functions are collected in a specific class.

However, in practice, the system running the same code in different states may cause different problems. Even if the code is tested by fuzzing, it does not mean that there are no problems. In order to assure a high trustworthiness level, formal verification, a technology that can verify whether the protocol meets specific properties by mathematical methods is also necessary.

Chapter 3

Methodology

This thesis combines model checking and fuzzing, which makes up for the shortcomings of each approach, and also fulfills the requirement of TAL3. The experiments were carried out with a MacBook Air having an Apple M2 CPU and 16 GB memory. The overall workflow of the verification process is as shown below (see also Figure 3.1):

Step 1. Design the protocol

Step 2. Formally model the protocol

Step 3. Specify Properties

Step 4. Conduct model checking

Step 5. Emulate the protocol

Step 6. Conduct fuzzing

Step 7. Analyze the results; if necessary, propose improvements, then repeat from step 1

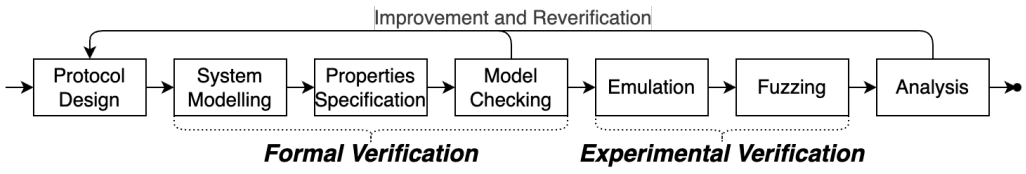

Figure 3.1: Overall Verification Workflow.

3.1 Control Protocol Description

3.1.1 Basic Version

The basic version of the robot control protocol is shown in Figure 3.2.

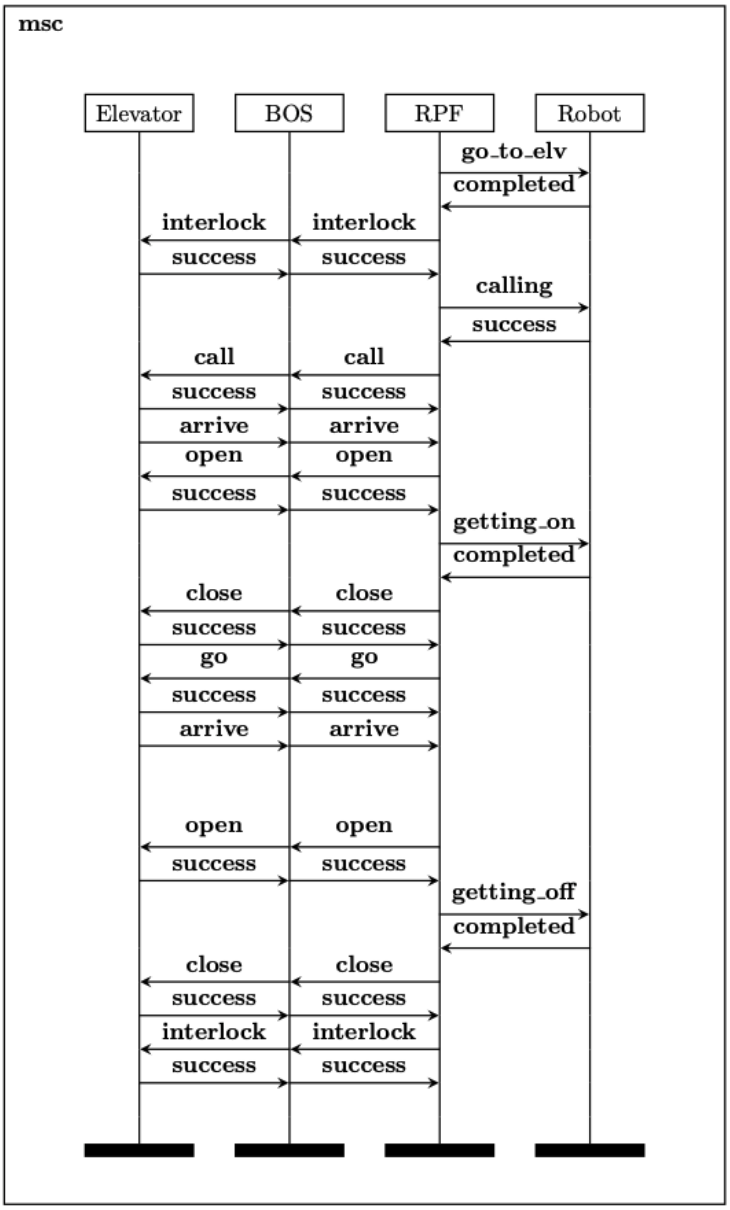

Figure 3.2: Robot Control Protocol Diagram.

The protocol involves four subsystems: elevator subsystem, building

OS (BOS), robot platform control subsystem (RPF), and robot subsystem. Despite the building OS should be the core of a smart building, this thesis mainly focuses on the robot control protocol and will not discuss the building OS in detail.

The robot control protocol is implemented on the robot control subsystem, which is a platform for remotely controlling and collaborative work among robots in the building. The robot subsystem and the elevator subsystem are the entities of robots and elevators in the building. The system involves two kinds of messages: information message and control message. Information message contains the current entity status like the current floor. Control message and its replies indicate the entity control rules. There are mainly 11 kinds of control messages:

1. interlock(*Boolean*): Command an elevator to the corresponding specific status.
2. call(*target_floor*): Command an elevator to move to the target floor.
3. open: Command an elevator door to open.
4. close: Command an elevator door to close.
5. go(*target_floor*): Command an elevator to move to the target floor.
6. GoToElv: Command a robot to go to the front of the elevator.
7. GettingOn: Command a robot to get on the elevator.
8. GettingOff: Command a robot to get off the elevator.
9. Schedule_Work: Command a robot to work following it's schedule.
10. Charge: Command a robot to charge.
11. Calling: Command a robot to enter calling status.

An elevator sends continuously information messages, the payload of which mainly contains the following:

- inDrivingPermission: It indicates whether the elevator is in a robot specific mode.
- floor: It indicates the current floor of the elevator.
- door: It indicates whether the elevator door is opening.

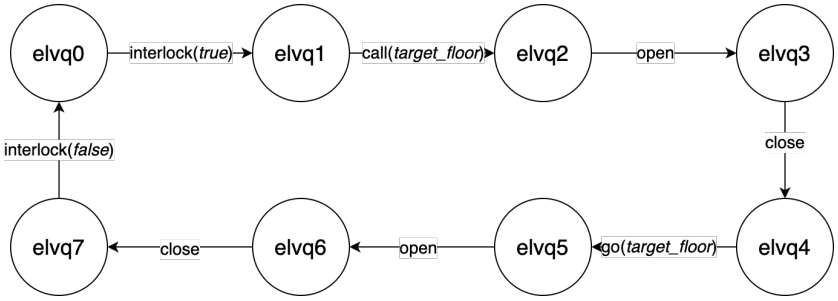

Figure 3.3: Elevator State Transition Diagram.

According to the received control messages, the elevator will transition to different states. The state transition of an elevator is shown in Figure 3.3.

A robot sends information messages continuously, which payload mainly contains the following:

- name: It indicates the robot name.
- floor: It indicates the current floor of the robot.
- statue: It indicates the status of the robot.
- position: It indicates the position of the robot.

Similar to elevator, the state transition of a robot is shown as figure 3.4.

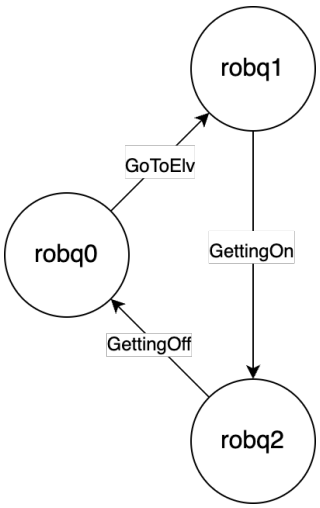

Figure 3.4: Robot State Transition Diagram.

The function of the Building OS is to forward messages between the robot control subsystem and the elevator subsystem, without any state transition.

According to the replies, the robot control subsystem will transition to different states, which is shown as figure 3.5.

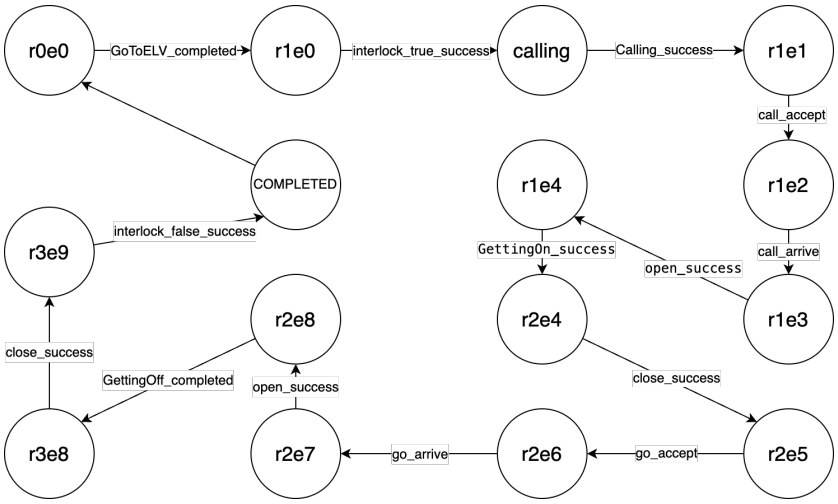

Figure 3.5: RPF State Transition Diagram.

3.1.2 Improved Protocol Versions

In this thesis, after verifying the basic protocol, we designed some improved protocol versions and re-verified them, corresponding to the workflow shown in Figure 3.1. For example, we designed a multi robots control version. Multi robots control uses a waiting queue. Only the first row robot of the queue can put the elevator into the robot special state and enter the elevator, while the other robots stay and wait.

3.2 Model Checking

3.2.1 System Modeling

We use the following observable components to specify the basic version of the protocol, all of which correspond to S_{ees} of K_{ees} :

- (ELV: s): It represents the state of the elevator.
- (movingStatus[ELV]: ms): It represents the moving status of the elevator.
- (BOS: s): It represents the state of the building OS.

- (RPF: s): It represents the state of the robot control subsystem.
- (msgCount[RPF]: n): It represents the number of messages sent by the robot control subsystem in the current state.
- (ROB: s): It represents the state of the robot.
- (movingStatus[ROB]: ms): It represents the moving status of a robot.
- (nw: $msgs$): It represents messages in the network.
- (tran: t): It represents the state transition taken most recently.

I_{ees} is the initial state of all observable components, defined as follows:

```
eq ic = (ELV: elvq0)
        (movingStatus[ELV]: stay)
        (BOS: bosq)
        (RPF: r0e0)
        (msgCount[RPF]: 0)
        (ROB: robq0)
        (movingStatus[ROB]: stay)
        (nw: void)
        (tran: notran) .
```

We defined the state transition for all subsystems through 46 rewriting rules, which correspond to T_{ees} of K_{ees}. For instance, a rewriting rule is as follows:

```
rl [sendGoToElv] :
    (RPF: r0e0)
    (msgCount[RPF]: 0)
    (nw: NW)
    (tran: T)
    => (RPF: r0e0)
       (msgCount[RPF]: 1)
       (nw: (msg(B2R, gotoelv) NW))
       (tran: sendGoToElv(RPF)) .
```

The above description means that when the control protocol is in $r0e0$ state and the message counter is 0, a message will be sent to the network, and the message counter will become one.

Part of the atomic state propositions are listed below, all of which correspond to P_{ees} of K_{ees}:

- $applied(t)$ holds if and only if t is the transition that was applied most recently.

- *enabled*(t) holds if and only if t is the transition that can be applied next.
- *crit* holds if and only if the robot is in the critical section.
- *want* holds if and only if the protocol at the $r0e0$ state means robot should move to another floor .
- *robotarrive* holds if and only if the protocol at the *completed* state means the robot arrive the target floor .

For the improved versions, according to the situation, observable components and rewriting rules will be modified. As an example, for the multiple robots control, we added new observable components (interlock: *bool*) to represent the interlock status of an elevator and (queue: *empty*) to represent a waiting queue of robots.

3.2.2 Property Specification

We used linear temporal logic (LTL) [7] for specifying property formulas, and the Maude LTL model checker to check.

For the basic version, we verified the liveness properties. Liveness property is reflected by that if a robot wants to go to the destination, it must reach the destination. For the improved versions, besides the liveness properties, we also verified the safety properties according to the situation. Safety property is reflected by that the multiple robots will not enter the critical section at the same time.

The liveness property of basic version:

$$(want \rightsquigarrow robotarrive)$$

The liveness property of multi robot version:

$$\begin{aligned}&(want(rob_1) \rightsquigarrow robotarrive(rob_1))\wedge\\&(want(rob_2) \rightsquigarrow robotarrive(rob_2))\wedge\\&(want(rob_3) \rightsquigarrow robotarrive(rob_3))\end{aligned}$$

The safety property:

$$\begin{aligned}\Box(&\neg(crit(rob_1) \wedge crit(rob_2))\wedge\\&\neg(crit(rob1) \wedge crit(rob3))\wedge\\&\neg(crit(rob2) \wedge crit(rob3)))\end{aligned}$$

The model checker traverses execution paths of the state space to find counterexamples. Although finding no counterexample means that the

system meets the requirements to be verified, finding a counterexample does not mean that the system must not meet the requirement. We will give two specific examples in Sects. 3.2.3 and 3.2.4. To avoid such problems, model checking is required under the fairness assumptions. There are two main types of fairness assumptions, namely weak fairness assumptions and strong fairness assumptions.

3.2.3 Weak Fairness Assumptions

Consider a state transition as that shown in Figure 3.6. When checking $q0 \rightsquigarrow q2$, as $q1$ can transition to $q1$, the path $q0; (q1)^{\infty}$ will be regarded as a counterexample. Let t_1 be the state transition $q1 \rightarrow q2$, let $enabled(t_1)$ hold if and only if in $q1$, let $applied(t_1)$ hold if and only if in $q2$. It can be seen that in the path $(q1)^{\infty}$, $enabled(t_1)$ always holds. An example of the weak fairness assumption is that when the system is always in $q1$, it will eventually transition to $q2$.

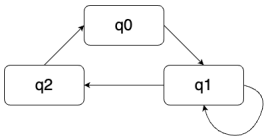

Figure 3.6: State Transition Example 1.

Weak fairness:

$$(\Diamond\Box enabled(t_1)) \Rightarrow (\Box\Diamond applied(t_1))$$

This is reflected in our system, consider a scenario where a robot receives a command but consistently do not execute it. Let $q1$ be a state where a robot received a command but have not executed it yet. The $q0$ transition to $q1$ means the robot received a command. The $q1$ transition to $q2$ means that the command has been successfully executed, whereas $q1$ transition to $q1$ means that the command still not been executed. In the real system, when a robot successfully execute a command, the robot will send a reply for successful execution, while if the control system doesn't receive the reply for successful execution over a period of time, it will raise a timeout exception. Because this situation has been considered in the real system, this path is not need to be regarded as a counterexample in the model checking, the robot can be assumed that it will eventually execute the command after receiving. Hence, we need to check under the weak fairness assumptions.

3.2.4 Strong Fairness Assumptions

Consider a state transition as shown in Figure 3.7. When checking $q4 \rightsquigarrow q6$, as $q5$ can transition to $q7$, the path $q4;(q5;q7)^{\infty}$ will be regarded as a counterexample. Similarly, let t_2 be the state transition $q5 \rightarrow q6$, let $enabled(t_2)$ hold if and only if in $q5$, let $applied(t_2)$ hold if and only if in $q6$. It can be seen that in the path $(q5;q7)^{\infty}$, $enabled(t_2)$ continually holds. An example of the strong fairness assumption is that when a system transition to $q5$ infinitely many times, it will eventually transition to $q6$.

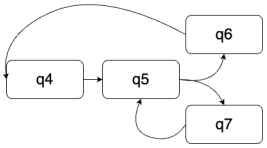

Figure 3.7: State Transition Example 2.

Strong fairness:

$$(\Box\Diamond enabled(t_2)) \Rightarrow (\Box\Diamond applied(t_2))$$

This is reflected in our system, which is related to the resilience of the protocol. It can be imagined that when the system is running, if an exception occurred, the protocol should have a certain degree of exception handling capability. $q7$ can be regarded as an exception handling state. When in $q5$ and an exception occurred, the protocol will transition to $q7$, handles the exception and returns to $q5$. However, if the same exception keeps occurring, the system will transition to $q7$ again, return to $q5$ and transition to $q7$ again and again. In the real system, if the protocol is in an infinite state transition loop, the control system will raise a timeout exception. Similar to Sect. 3.2.3, we don't have to regard this path as a counterexample in the model checking. Hence, the model checking is required under the strong fairness assumption so as to assume that the exception handling will eventually succeed.

3.2.5 Divide and Conquer Method

In the case of multi robots, when checking the liveness property, the checker will find a counterexample, which shows that a robot has never entered the waiting queue. As mentioned before, because there is no need to regard such a path as a counterexample, we should use the weak fairness assumptions for model checking. However, under weak fairness assumptions, our model checker has not any outputs after a period of time, which may be caused

by the state explosion problem. Therefore, we need to apply the divide and conquer method to avoid state explosion.

About the divide and conquer method, on the basis of the fairness assumption, we added the quasi-fairness assumption. Let all the state transitions related to the robot joining the waiting queue be the weak fairness assumptions, which allows us to conjecture that whenever a robot is not in a queue, it will eventually enter the queue, which is expressed as `qfair12(I)`.

The `qfair12(I)` condition is defined as follows: $\Diamond\Box\neg queue?(robot_i) \rightsquigarrow \Box\Diamond queue?(robot_i)$. Moreover, the atomic state proposition $queue?(robot_i)$ is defined as:

```
1    ceq (queue: Q ) C |= queue?(I) = true if I \in Q .
```

This represents whether the $robot_i$ is in the waiting queue. The quasi-fairness assumption qfair12 is qfair12($robot_1$) $\wedge$ qfair12($robot_2$) $\wedge$ qfair12($robot_3$). The divide and conquer method refers to dividing model checking fair $\Rightarrow$ liveness property into model checking the fair $\Rightarrow$ qfair12(I) and the qfair12 $\Rightarrow$ liveness property.

3.3 Protocol and Device Implementation

3.3.1 Protocol Implementation

We use a state machine to model the devices and protocols. We analyzed the operation of the protocol, added state information, and developed an emulator named Smart Building Control System Emulator (SBCSE) using Python. The emulator architecture is shown in Figure 3.8; note that the control protocol is emulated, but the devices are simulated. For more details about SBCSE, see [14].

3.3.2 Device Implementation

The device motion module is part of the SBCSE, which aims to provide definitions and management methods for devices, including robots and elevators.

Specifically, the device definitions include not only the state involved in the state machine mentioned before, but also some information such as the motion status and the position of the device. On top of that, we further defined the device's motion, the examples are the robot working, going to the elevator, getting on & off the elevator, the elevator move up & down, and the elevator door open & close, etc. we implemented these in the emulator as variables and functions. Moreover, we also designed some functions to

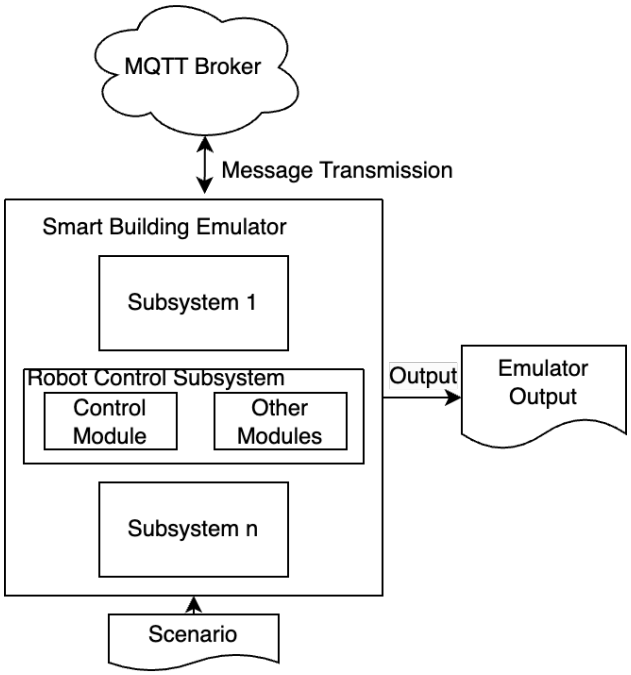

Figure 3.8: Smart Building Emulator Architecture.

adjust these for simulation. A typical function is to find a suitable path for robot movement.

In terms of simulation, the robot simulator, the elevator simulator, and their related modules will operate the attributes or functions of the devices according to the messages. Meanwhile, based on the requirements, the emulator will calculate the time. When the interval reaches or exceeds the set threshold, the corresponding attribute or functions will also be operated.

In order to support protocol testing, the robot also has a protocol test mode. In protocol test mode, the robot will not execute the path for the simulated work, which would take a very long time; instead, it will wait a short time and start the next task. This allows us to pay more attention to the protocol testing. Since the objective of this thesis is to verify the protocol, most experiments are conducted in the protocol test mode.

We also designed some management classes for handling robots and elevators, both of which are singleton classes and can be used to manage the copy and destruction of device instances to save snapshots in some fuzzing situations to optimize testing strategies (see Sect. 3.4.2 for details). The robot management class is also used for multi robots handler.

3.4 Fuzzing Method

We used Atheris for fuzzing, which is a coverage-guided fuzzing engine [12]. The fuzzing architecture is shown in the Figure 3.9. The fuzzing method involves generating random messages sent and received by the robot control subsystem to observe whether the emulator behaviors are abnormal. If any abnormal behavior appears, logs are analyzed to diagnose problems. We randomly generate seeds, and the seed is both the seed that generates the Smart Building Emulator scenario and the fuzzer seed. Although the same seed can generate the same scenario, unfortunately, it does not always generate the same fuzzing case.

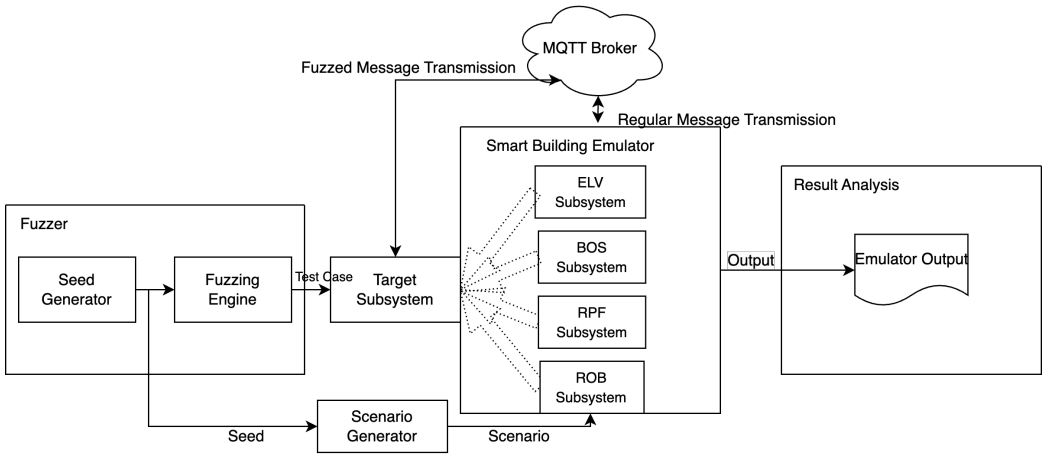

Figure 3.9: Fuzzing Architecture.

For the fuzzing method, this thesis will mainly explain from three parts: Sect. 3.4.1 explains how to generate test messages; Sect. 3.4.2 explains how to optimize the test strategy; Sect. 3.4.3 explains the emulator's abnormal behaviors.

3.4.1 Test Message Generation

This section explains the test message generation in regard to two aspects, the types of messages and the target subsystems, as described next.

Types of Messages For the types of messages, although fuzzing is characterized by the generation of random test case types in general, there are mainly three types in our system: correct messages, incorrect parameters

messages and incorrect messages. Table 3.1 summarizes the three message types and three effects according to the effect of different message types.

Correct messages and incorrect parameters messages all have correct formats. There is not necessary to repeat the correct messages. An example of incorrect parameters message is that the control system sends a command, which is received by the target but has not been executed yet, however, the control system receives a reply for successful execution. Regarding the effect, if the command format is correct, the problem could be caught by the control module possibly, and there will be corresponding problem handling or exception raising. Sometimes it can be equivalently considered to be caused by malicious message tampering or incorrect data transmission, both of which result in incorrect parameters. Exceptions might help us evaluate how the protocol handles incorrect data.

Incorrect message means even the format is incorrect. Consider a scenario in which the control system sends command 1, but receives a reply for command 2 has been completed, or a reply including garbled characters. Concerning the effect, if the command format is incorrect, the system will not accept it. Sometimes it can be equivalently considered to be caused by message loss or device failure, both of which result in not receiving valuable messages. This helps us to evaluate how the protocol handles data loss or detects failure.

Table 3.1: Fuzzed Message

Received / ***Sent***	**Correct Msg.**	**Incorrect Parameter Msg.**	**Incorrect Msg.**
Correct Msg.	Regular Operation	Exception	Message Loss
Incorrect Parameters Msg.	Exception	Exception	Message Loss
Incorrect Msg.	Message Loss	Message Loss	Message Loss

Target Subsystems In regard to the target subsystems, we distinguished between the robot control subsystem fuzzing and other subsystems fuzzing according to the robot control protocol. Only the robot control subsystem sends control messages, while other subsystems respond and reply control messages and send information messages. We overridden or implemented the corresponding message sent function. When a new fuzzing starts, the inputted seed will also specify the target of this fuzzing when generating the emulator scenario.

3.4.2 Optimization Strategies

As Atheris is a coverage-guided fuzzer, the first optimization strategy is to apply the coverage-guided features. The protocol was emulated by a state machine, which different states send different messages and transition to the next state based on the replies. When testing the protocol, the main is testing the state transition. Thus, covering as many as state during fuzzing is important. The first strategy is instrumenting the functions related to the state transition, and Atheris will automatically cover as many state as possible.

The second optimization strategy is to save snapshots of the current state. If the control system cannot receive an acceptable reply within a period of time, there will be a timeout exception raised. In fact, the characteristics of fuzzing determine that most test cases will trigger timeout exceptions. During the fuzzing, the emulator is not necessary to retest from the beginning if a timeout exception is triggered. The fuzzing module will save the device snapshots when the system transition to a new state, and if a timeout exception is triggered, the emulator will only reset to the saved snapshot state.

To evaluate the effectiveness of our optimization strategy, we randomly selected 30 seeds for each target, and ran tests under different strategies with each test set to 1000 test runs. In these experiments, we used four mosquito processes with different PIDs to listen to four different ports separately, ran four different emulator programs, and each program conducted different subsystems fuzzing.

The tool coverage.py [15] is a tool that lets us view the code coverage of the emulator running. After 1000 x 30 runs, we recorded the coverage and test runtime, as shown in Tables 3.2 and 3.3. It should be noted that the time we recorded is the testing runtime reported by the fuzzer, not the emulator runtime. It's also important to note that our table only gives a rough idea that our work is effective. Due to fuzzing is random, if there aren't enough samples, the specific numbers in the table might not be very accurate. These numbers are more for showing the general improvement trend, not for detailed analysis. It should also be noted that the coverage mentioned in this thesis only refers to the coverage of the class of the protocol. When developing the emulator, we use a special class to implement the state machine of the robot control protocol.

Our strategies show weak performance in coverage for the fuzzing of the elevator subsystem, robot subsystem, and Building OS. However, our strategies show a significant improvement in coverage for the fuzzing of the robot control subsystem. Most of the problems mentioned later in Chapter 4

Table 3.2: Coverage for the Four Subsystems with 30 Seeds Each

	None	Coverage-Guided	Snapshot	Both
Elevator	47%	52%	50%	52%
BOS	47%	49%	50%	52%
RPF	36%	39%	64%	94%
Robot	37%	40%	37%	40%

None: Fuzzing with no strategies applied
Coverage-Guided: Coverage-guided fuzzing
Snapshot: Save and load the snapshots fuzzing
Both: Fuzzing with both strategies above applied

Table 3.3: Execution Time for the Four Subsystems with 30 Seeds Each

	None	Coverage-Guided	Snapshot	Both
Elevator	0.5h	0.5h	42.4h	42.4h
BOS	0.5h	0.5h	34.9h	36.3h
RPF	0.4h	0.4h	258.3h	217.9h
Robot	0.4h	0.4h	20.2h	20.7h

None: Fuzzing with no strategies applied
Coverage-Guided: Coverage-guided fuzzing
Snapshot: Save and load the snapshots fuzzing
Both: Fuzzing with both strategies above applied

do come from the fuzzing of the robot control subsystem, so we believe that the proposed strategies do help to improve the efficiency of fuzzing.

3.4.3 Emulator Behaviors

Expected behaviors are categorized as:

1. Protocol exception occurred.
2. All tasks set in the scenario are completed without obvious problems.
3. Not all tasks are completed, but no obvious problems are found.

Regarding the first behavior, it represents expected exception occurred. We defined some protocol exceptions, such as timeout exception. When a protocol exception occurs, it means that there is an exception caught by the control module but cannot be handled. In this case, other modules are responsible for handling this exception, which can be regarded as a normal behavior from the perspective of the control protocol.

The second behavior indicates the emulator's normal termination. Nevertheless, if the emulator does not normal terminate, it still can be regarded as a normal behavior. As it was mentioned in Sect. 3.4.2, due to the characteristics, fuzzing usually triggers timeout exceptions, and when a timeout exception is triggered, the emulator will return to the saved snapshot. At the beginning of each test, we should set a number of test runs. If a test keeps triggering a timeout exception and going back to the saved snapshot again and again, we may not be able to complete all the tasks set in the scenario although all test runs are completed. However, there is no new undefined exception occurred. Thus, we classify this situation as the third normal behavior.

Abnormal behaviors are shown as follows:

1. Program exception occurred.
2. Obvious abnormal entries in the logs.
3. The emulator doesn't raise an exception, but no meaningful behavior.

The first abnormal behavior differs from the first normal behavior, which unexpected exceptions occurred is an abnormal behavior. About the second abnormal behavior, for example, a robot starts a new action but it's old action doesn't completed yet, or a log with single entry of action start but multiple entries of this action completed. Another example is, a command is sent to robot 2 but robot 1 response the command. For the third abnormal behavior, if some issues such as deadlock occurred, these issues will cause the emulator to neither stop nor run, leading to obvious abnormal behavior wouldn't appeared. However, we can notice there is no meaningful behavior in the emulator, which also be regarded as an abnormal behavior. In most cases, it can be found through the situation that the emulator is still running but no subsystems change state for a long time.

Chapter 4

Results and Analysis

In this chapter and the next chapter, we will show some results, analysis, and improvement plans for some problems. It should be noted that due to the further improvement of the program, the seed in this thesis may not trigger the same problems in the latest version of the emulator.

4.1 Model Checking Results

For the basic version of the protocols, the results are shown in Table 4.1.

Table 4.1: Model Checking Results

	Deadlock*	Safety†	Liveness†
Single Robot Version	✓		✓
Multi Robot Version	✓	✓	✓

*Using Maude **search** Command
†Using Maude LTL Model Checker

4.1.1 Evaluation of Models

The command **search ic =>* C:Config** can output all the paths that can be reached from the initial state. We checked the path to confirm that the model is consistent with the protocol process shown in Figure 3.2. Judging from the transition of the observable component **(tran: T)** in the path, our model maintains consistency with the target protocol.

We use **search =>! C:Config** to check whether our model has any deadlock states, and the results which output "no solution" demonstrate the absence of deadlock.

```
Maude> search ic =>! C:Config .
search in RCP-FORMULA : ic =>! C:Config .

No solution.
states: 77  rewrites: 88 in 1ms cpu (1ms real) (61624 rewrites/second)
```

Figure 4.1: Basic Version Deadlock.

```
Maude> search ic =>! C:Config .
search in RCP-FORMULA : ic =>! C:Config .

No solution.
states: 19748  rewrites: 113479 in 270ms cpu (272ms real) (419002
    rewrites/second)
```

Figure 4.2: Multi Version Deadlock.

4.1.2 Model Checking Results

We verified the liveness property in the basic version of the protocol and verified the safety and liveness properties of the multiple robots version. The result of the basic version is shown in Figure 4.3.

```
Maude> red in RCP-FORMULA : modelCheck(ic,liveness) .
reduce in RCP-FORMULA : modelCheck(ic, liveness) .
rewrites: 118 in 1ms cpu (1ms real) (85631 rewrites/second)
result Bool: true
```

Figure 4.3: Basic Version Liveness Property.

The results of the multiple robots version are shown in Figures 4.4 and 4.5. All these properties satisfy the requirements. Moreover, to avoid the situation in which the properties hold vacuously (i.e., because their conditions can never be satisfied), we also checked that the property RCP-FORMULA $\models \sim$ qfair does not hold in the multiple robots version (see Figure 4.6).

From the results of the model checking, we can conclude that there are no problems in deadlock, safety and liveness properties in the basic process of our protocols.

4.2 Fuzzing Results

4.2.1 Problem Diagnosis

During the fuzzing, we analyzed the logs and summarized ten major problems, which were classified according to Common Weakness Enumeration(CWE) [16] as follows: CWE-20 *Improper Input Validation*, CWE-754 *Improper Check for Unusual or Exceptional Conditions*, CWE-833 *Deadlock*,

```
Maude> red in RCP-FORMULA : modelCheck(ic,safety) .
reduce in RCP-FORMULA : modelCheck(ic, safety) .
rewrites: 163534 in 351ms cpu (359ms real) (465332 rewrites/second)
result Bool: true
```

Figure 4.4: Multi Robots Version Safety Property.

```
Maude> red in RCP-FORMULA : modelCheck(ic, fair1(1) -> qfair12(1)) .
reduce in RCP-FORMULA : modelCheck(ic, fair1(1) -> qfair12(1)) .
rewrites: 261020 in 330ms cpu (333ms real) (790591 rewrites/second)
result Bool: true
Maude> red in RCP-FORMULA : modelCheck(ic, fair1(2) -> qfair12(2)) .
reduce in RCP-FORMULA : modelCheck(ic, fair1(2) -> qfair12(2)) .
rewrites: 261020 in 327ms cpu (329ms real) (796331 rewrites/second)
result Bool: true
Maude> red in RCP-FORMULA : modelCheck(ic, fair1(3) -> qfair12(3)) .
reduce in RCP-FORMULA : modelCheck(ic, fair1(3) -> qfair12(3)) .
rewrites: 261020 in 324ms cpu (325ms real) (805545 rewrites/second)
result Bool: true
Maude> red in RCP-FORMULA : modelCheck(ic, qfair -> liveness) .
reduce in RCP-FORMULA : modelCheck(ic, qfair -> liveness) .
rewrites: 473437 in 434ms cpu (435ms real) (1089793 rewrites/second)
result Bool: true
```

Figure 4.5: Multi Robots Version Liveness Property under Fairness Assumption.

CWE-571 *Expression is Always True*, CWE-691 *Insufficient Control Flow Management*, and CWE-431 *Missing Handler*, as summarized in Table 4.2.

Table 4.2: Problems Found Through Fuzzing

No.	Description	CWE
1	Improper Input Validation	CWE-20
2	Command an elevator to go to an incorrect floor	CWE-754
3	An elevator go to an incorrect floor	CWE-754
4	Command a robot to pass through a closed door	CWE-754
5	Deadlock	CWE-833
6	Expression is Always True	CWE-571
7	Insufficient Control Flow Management	CWE-691
8	Missing Handler	CWE-431
9	Inappropriate parameter selection	N/A
10	Unknown Reasons	N/A

CWE-20 includes problem 1, which due to the failure to handle the garbled characters. In some cases if a message containing garbled characters is sent to the robot control subsystem, it will cause the robot control subsystem to raise an undefined exception. CWE-754 includes problems 2-4, which are all caused by incorrect message data. Problems 5-8 stem from the wrong code during programming, are separately classified under CWE-833,

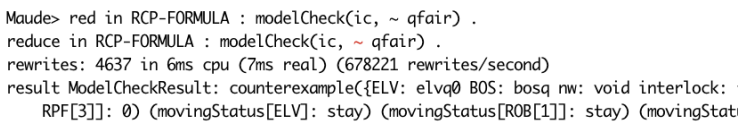

Figure 4.6: Multi Robots Version RCP-FORMULA $\models \sim$ qfair

CWE-571, CWE-691, CWE-431. Problem 9 arises from other programming problems. Problem 10 appears from some unknown reasons that cannot be easily reproduced.

4.2.2 Problem Analysis

Problems 2-4 Problem 2 can be seen as that the commands sent by the robot control subsystem to be tampered with. For example, consider a scenario where the robot control subsystem sends a command to an elevator to go to the 8th floor. If this command is tampered with to go to an incorrect floor, the elevator will go to the incorrect floor. This case may cause someone to go to an unauthorized zone. Problem 3 is based on problem 2, with even the message indicating the elevator's arrivaleing tampered with. One example is the situation in which the elevator moves to an incorrect floor; if the message indicating the elevator's aarrivalis tampered with, it may result in a false positive acknowledgment. Problem 4 can be seen as that the replies of some commands received by the robot control subsystem to be tampered with. Consider a scenario in which an elevator received a command to open the door, but the door keeps closing. However, the reply is tampered with to indicate the door is open. Such a message may lead to the robot control system commanding a robot to move through a closed door.

Problem 5 Problem 5 will cause deadlock. When a robot is in a working status for the first command, if the robot receives the second command, it will reply with a message to indicate successfully receiving the command but discard it without executing it. After completing the first command, the robot will enter a waiting status for the third command. Meanwhile, the robot control subsystem will continue to wait for feedback that the robot has successfully executed the second command, due to it receiving the reply of successfully receiving the second command, and will not send the third command. This case will result in a deadlock status. It can be seen from the log that between 14:17:04 and 14:17:34 in Figure 4.7, the robot was performing the command **schedule_work**, but at 14:17:28, it received the command of **gotoelv**, at which time the robot had actually discarded the

command.

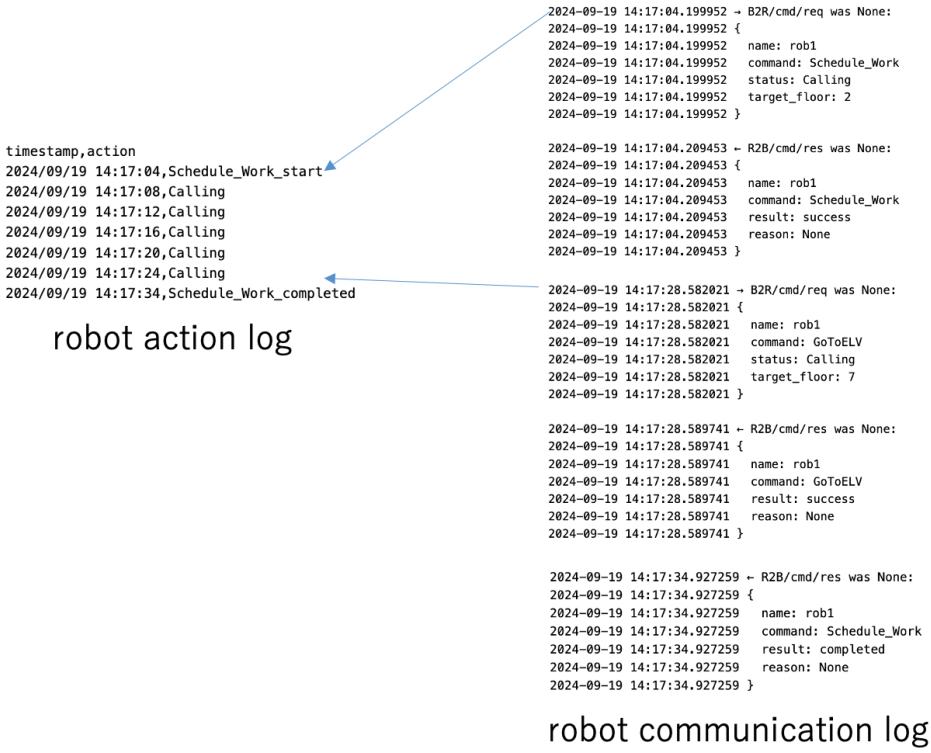

Figure 4.7: Problem 5 Deadlock.

Problem 6 Problem 6 is about an if conditional statement that is always true after a specific operation. For example, this Python code:

```
1 def interlock(self):
2   if self.bcp_status = S.E0:
3     self.handler.send_interlock_command (True)
4     if self.check_success (ELV. INTERLOCK_SUCCESS) :
5       self.bcp_status = S.E1
6     elif self.bcp_status == S.E9:
7       self.handler.send_interlock_command(False)
8       if self.check_success (ELV.INTERLOCK_SUCCESS) :
9         self.bcp_status = S.COMPLETED
```

When the protocol is in the $E0$ state, this function will be called, and send a message in line 3, and according to the reply to set the **ELV.INTERLOCK_SUCCESS** to $True$ and transition to the next state. However, when in the $E9$ state, the function will be called again, but due to this variable still being $True$, which means regardless of whether there is a

reply to the message sent in line 7, the if conditional statement in line 8 will always be $True$, and will enter the next state.

Problem 7 To illustrate Problem 7, imagine a scenario where if the robot control subsystem receives replies, it will immediately change the attributes according to the replies. But due to insufficient control flow management, if it receives a reply to the command that has not been sent, the attributes will also be changed. In fact, this is also a problem of message authentication.

Problem 8 During the fuzzing, we found some code like this.

```
1 try:
2     # do something 1
3 finally:
4     # do something 2
```

The problem is that if there is an exception occurred during *do something 1*, the program will enter the code block *do something 2* without any error message. This caused us to be confused when analyzing the logs. We can only find abnormal behaviors in the logs, but it is difficult to locate the specific code. Another case which is included in problem 8 is incorrect error status code output. When a certain exception occurs, it should output the corresponding error code; however, the log sometimes records an error error code.

Problem 9 Problem 9 arises from inappropriate parameter selection during programming. One such example is that the robot control subsystem will raise a timeout exception when it does not receive a reply within a period of time. However, some periods are set too short, causing the system to raise timeout exceptions even for acceptable delays.

Problem 10 Some problems cannot be easily reproduced. One guess is that because our emulator has a speed controlling feature, and most of the time, the fuzzing is set at 10x speed, such unreproducible problems sometimes occurred. However, while collecting data for this thesis, we ran more than 120 seeds at normal speed (1x), and this kind of problems didn't show up. This makes us think the problem might happen because of thread competition when the speed is too fast. We have not completely solved this problem. One way to avoid it is to limit the maximum running speed of the emulator during fuzzing.

Chapter 5

Improvement and Reverification

The problems found through fuzzing were not found through model checking, because most of the problems come from improper programming implementation and abnormal message transmission, which have not been considered when modeling the protocol. We proposed solutions to some of the problems, constructed new formal models, and conducted reverification, as summarized in Table 5.1.

To evaluate the formal verification, we checked the required properties of new models. If these properties satisfied the requirement, we will record the problem done. To evaluate the experimental verification, we can use the seeds which can trigger the problem to observe whether the seed will still lead to abnormal behaviors; if not, we will record the problem as solved. We can also check the coverage files and logs to confirm whether the modified part is running.

Regarding problem 1, we have no way to formally verify whether there still have problems that lack input validation through the model checking. As for the problems 2-4, we added some methods to identify whether the message has been tampered with and constructed a new formal model to check. Concerning problems 5-6, we modeled the execution order of the code. However, about the problem 7, modifying the execution order cannot solve all the problems, since message authentication should be assigned to the communication module rather than handled at the control module. For the problems 8 and 9, there are also not the problems caused by the control protocol, it's also not necessary to construct a new formal model. For problem 10, since it cannot be reproduced, we cannot solve it.

Table 5.1: Reverification Results

No.	Description	F	E
1	Improper Input Validation	N/A	Done
2	Command an elevator to go to an incorrect floor	Done	Done
3	An elevator go to an incorrect floor	Done	Done
4	Command a robot to pass through a closed door	Done	Done
5	Deadlock	Done	Done
6	Expression is Always True	Done	Done
7	Insufficient Control Flow Management	N/A	Done
8	Missing Handler	N/A	Done
9	Inappropriate parameter selection	N/A	Done
10	Unknown Reasons	N/A	N/A

F: Formal Verification
E: Experimental Verification

5.1 Problem 2: Command Elevator to Go to Incorrect Floor

Since the control messages are tampered with while the replies and shared information messages are not, we can detect whether there is a problem by the robot control subsystem through comparing the sent messages and the replies. The improvement we propose is that if the robot control module detects that the elevator has gone to an incorrect floor through the comparing, it will try to resend the same message. Back to the example in Sect. 4.2, because after the elevator executes the command to move, it will have some messages to indicate arrival and position, then the robot control subsystem can detect whether it is correct through these messages and the floor of the target the elevator should move to. If the elevator arrives an incorrect floor, the robot control subsystem will return to the last state ($r1e2 \rightarrow r1e1$, $r2e6 \rightarrow r2e5$) and send the command again. The modified state transition is shown as Figure 5.1. In the emulator, if the same command message still fails to command the elevator to the correct floor after a certain number of resents, it will raise "E001" code exception and assign the exception to other modules for handling.

The new objective is to verify if the resend process works, we formalized this resend process without formalized resend times and error code. There is a new observable component (floor[ELV]:$boolean$) indicating whether the elevator arrives the correct floor, and changed the past two rewriting rules to the following four:

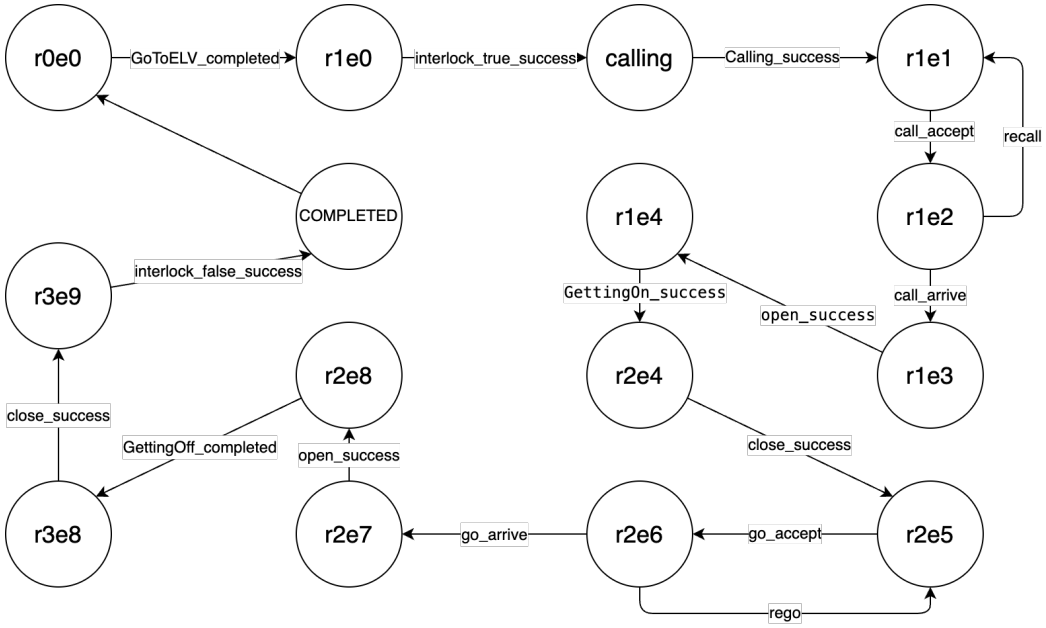

Figure 5.1: Problem 2 Modified State Transition.

```
1  rl [sendCallArrive1] :
2          (ELV: elvq2)
3          (movingStatus[ELV]: moving)
4          (floor[ELV]: true)
5          (nw: NW)
6          (tran: T)
7          => (ELV: elvq2)
8              (movingStatus[ELV]: stay)
9              (floor[ELV]: false)
10             (nw: (msg(E2D, callarrive) NW))
11             (tran: sendCallArrive1(ELV)) .
12
13 rl [sendCallArrive2] :
14     (ELV: elvq2)
15     (movingStatus[ELV]: moving)
16     (floor[ELV]: true)
17     (nw: NW)
18     (tran: T)
19     => (ELV: elvq2)
20         (movingStatus[ELV]: stay)
21         (floor[ELV]: true)
22         (nw: (msg(E2D, callarrive) NW))
23         (tran: sendCallArrive2(ELV)) .
```

After receiving messages, the elevator may go to an incorrect or correct floor. These rewriting rules indicate that the message be/not be tampered

with.

```
1  rl [recvCallArrive1] :
2       (RPF: r1e2)
3       (msgCount[RPF]: 0)
4       (floor[ELV]: false)
5       (nw: (msg(D2B, callarrive) NW))
6       (tran: T)
7       => (RPF: r1e1)
8          (msgCount[RPF]: 1)
9          (floor[ELV]: true)
10         (nw: (msg(B2D, call) NW))
11         (tran: recvCallArrive1(RPF)) .
12
13 rl [recvCallArrive2] :
14      (RPF: r1e2)
15      (msgCount[RPF]: 0)
16      (floor[ELV]: true)
17      (nw: (msg(D2B, callarrive) NW))
18      (tran: T)
19      => (RPF: r1e3)
20         (msgCount[RPF]: 0)
21         (floor[ELV]: true)
22         (nw: NW)
23         (tran: recvCallArrive2(RPF)) .
```

These indicate that the different handling when the elevator goes to a correct or incorrect floor.

Under the strong fairness assumptions, the liveness of the system can be successfully checked. Among them, `liveness6` and `liveness7` indicate that the elevator must be able to arrive correct floor after receiving the command of call and go respectively. `fair3`, `fair5`, `fair9` and `fair11` are assumptions that the elevator will eventually move to a correct floor. `liveness` indicates that the robot must be able to arrive destination it wants to go to. The results shown in Figure 5.2.

5.2 Problem 3: Elevator Goes to Incorrect Floor

In this situation, it is possible that the target is arriving the incorrect floor, but the messages indicate the correct arrival. At this time, the robot control subsystem cannot detect it due to the lack of correct messages, but the robot itself can. Therefore, the improvement plan we propose is that when a robot finds itself on an incorrect floor, it will try to send messages about being on the incorrect floor, which we define as "E002" code. When the robot

```
Maude> red in RCP-FORMULA : modelCheck(ic,fair3 /\ fair5 -> liveness6) .
reduce in RCP-FORMULA : modelCheck(ic, fair3 /\ fair5 -> liveness6) .
rewrites: 2362 in 9ms cpu (9ms real) (256934 rewrites/second)
result Bool: true
Maude> red in RCP-FORMULA : modelCheck(ic,fair9 /\ fair11 -> liveness7) .
reduce in RCP-FORMULA : modelCheck(ic, fair9 /\ fair11 -> liveness7) .
rewrites: 3915 in 21ms cpu (22ms real) (183707 rewrites/second)
result Bool: true
Maude> red in RCP-FORMULA : modelCheck(ic,fair3 /\ fair5 /\ fair9 /\ fair11 -> liveness) .
reduce in RCP-FORMULA : modelCheck(ic, fair11 /\ (fair9 /\ (fair3 /\ fair5)) -> liveness) .
rewrites: 16391 in 1025ms cpu (1060ms real) (15990 rewrites/second)
result Bool: true
```

Figure 5.2: Problem 2 Liveness Property.

control subsystem receives a message about "E002", it will try to handle it. Unlike for problem 2, we did not modify the state transition of the protocol, but added a method to detect whether the robot is on the correct floor. The model is also needs minimal changes, adding an observable component (floor[*ROB*]:*boolean*) indicating whether the robot is on the correct floor and the corresponding rewriting rule to go to the correct and incorrect floor.

```
1  rl [sendGettingOffCompleted1] :
2       (ROB: robq0)
3       (movingStatus[ROB]: moving)
4       (floor[ROB]: B)
5       (nw: NW)
6       (tran: T)
7       => (ROB: robq0)
8          (movingStatus[ROB]: stay)
9          (floor[ROB]: true)
10         (nw: (msg(R2B, gettingoffcompleted) NW))
11         (tran: sendGettingOffCompleted1(ROB)) .
12
13 rl [sendGettingOffCompleted2] :
14      (ROB: robq0)
15      (movingStatus[ROB]: moving)
16      (floor[ROB]: B)
17      (nw: NW)
18      (tran: T)
19      => (ROB: robq0)
20         (movingStatus[ROB]: stay)
21         (floor[ROB]: false)
22         (nw: (msg(R2B, gettingoffcompleted) NW))
23         (tran: sendGettingOffCompleted2(ROB)) .
```

The new liveness property means that the robot arrived at the correct floor. The liveness property can be successfully checked as shown in Figure 5.3.

```
Maude> red in RCP-FORMULA : modelCheck(ic,fair13 -> liveness) .
reduce in RCP-FORMULA : modelCheck(ic, fair13 -> liveness) .
rewrites: 972 in 2ms cpu (3ms real) (331853 rewrites/second)
result Bool: true
```

Figure 5.3: Problem 3 Liveness Property.

5.3 Problem 4: Command Robot to Pass Through a Closed Door

If the messages are tampered with, these messages may cause the robot control subsystem to control a robot through a closed door. However, in fact, because the robot has a certain environmental perception ability, it can detect whether there are physical obstacles on the path and stop moving, we define this situation as "E004" exception. Like the problem 3, the robot will sends messages include "E004". When the "E004" exception occurs, the protocol will back to last state ($r1e4 \rightarrow r1e3$, $r2e8 \rightarrow r2e7$) try to send the command to open the door again like Figure 5.4. If it cannot handle the exception within a certain number of times, it will assign the exception to other modules for handling. Similar to the previous work, we still simply added the observable component and the corresponding rewrite rules to indicate normal and abnormal situations.

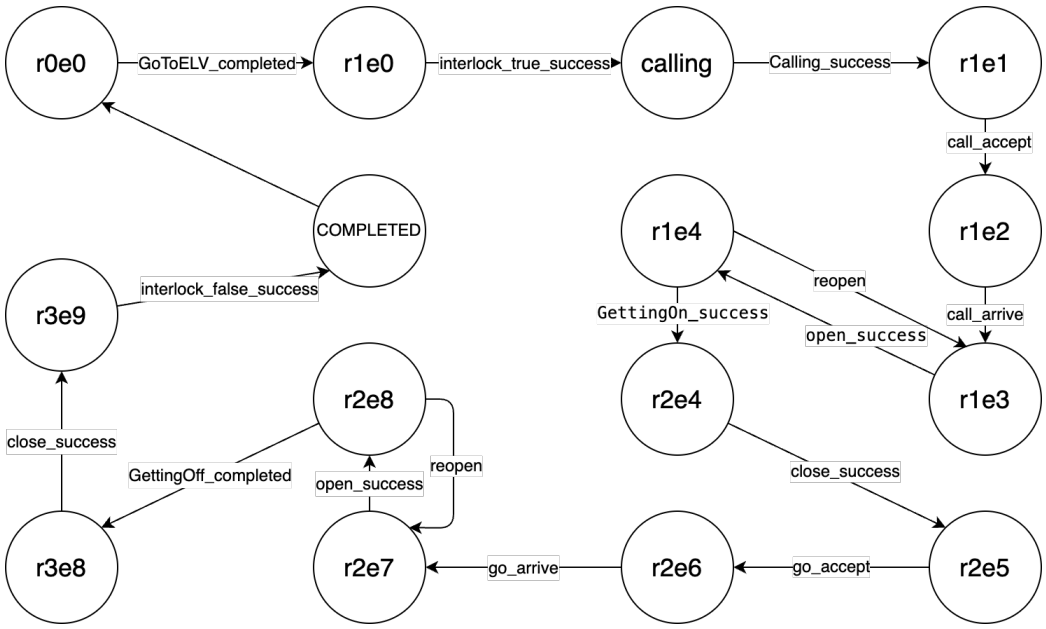

Figure 5.4: Problem 4 Modified State Transition.

The code below represents that the door opening command was sent, but the door was not successful/successfully opened.

```
rl [recvOpen11] :
    (ELV: elvq2)
    (door: close)
    (nw: (msg(D2E, open) NW))
    (tran: T)
    => (ELV: elvq3)
       (door: close)
       (nw: (msg(E2D, opensuccess) NW))
       (tran: recvOpen11(ELV)) .

rl [recvOpen12] :
    (ELV: elvq2)
    (door: close)
    (nw: (msg(D2E, open) NW))
    (tran: T)
    => (ELV: elvq3)
       (door: open)
       (nw: (msg(E2D, opensuccess) NW))
       (tran: recvOpen12(ELV)) .
```

The code below represents that if the door is not successfully opened, return to the $r1e3$ state and try to send it again.

```
rl [sendGettingOn1] :
    (RPF: r1e4)
    (ELV: elvq3)
    (door: close)
    (tran: T)
    => (RPF: r1e3)
       (ELV: elvq2)
       (door: close)
       (tran: sendGettingOn1(RPF)) .

rl [sendGettingOn2] :
    (RPF: r1e4)
    (msgCount[RPF]: 0)
    (door: open)
    (nw: NW)
    (tran: T)
    => (RPF: r1e4)
       (msgCount[RPF]: 1)
       (door: open)
       (nw: (msg(B2R, gettingon) NW))
       (tran: sendGettingOn2(RPF)) .
```

Under the strong fairness, the liveness of the system can be successfully checked. Among them, **liveness8** and **liveness9** indicate that the door must

be able to open after receiving the command of open. **fair6** and **fair12** are assumptions mean the door will eventually open. **liveness** indicates that the robot must be able to arrive the correct destination.

```
Maude> red in RCP-FORMULA : modelCheck(ic, fair6 -> liveness8) .
reduce in RCP-FORMULA : modelCheck(ic, fair6 -> liveness8) .
rewrites: 534 in 1ms cpu (2ms real) (294864 rewrites/second)
result Bool: true
Maude> red in RCP-FORMULA : modelCheck(ic, fair12 -> liveness9) .
reduce in RCP-FORMULA : modelCheck(ic, fair12 -> liveness9) .
rewrites: 544 in 1ms cpu (1ms real) (377777 rewrites/second)
result Bool: true
Maude> red in RCP-FORMULA : modelCheck(ic, fair6 /\ fair12 -> liveness) .
reduce in RCP-FORMULA : modelCheck(ic, fair6 /\ fair12 -> liveness) .
rewrites: 1208 in 5ms cpu (5ms real) (229222 rewrites/second)
result Bool: true
```

Figure 5.5: Problem 4 Liveness Property.

5.4 Problem 5: Deadlock

Problem 5 did not appear in our previous formal verification, it reminds us that our previous formal model did not formalize other commands of the robot. We have added more states to the robot, as shown in the Figure 5.6.

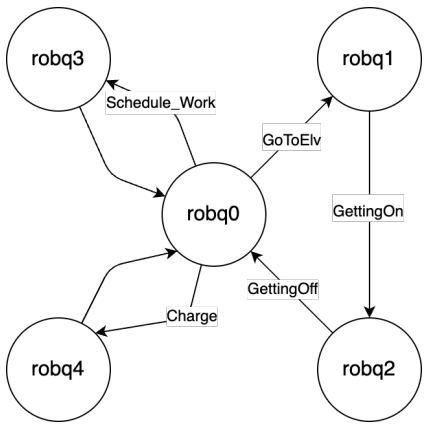

Figure 5.6: New Robot State Transition Diagram.

We formalized the state transition with some new rewriting rules, including those shown below.

```
1  rl [sendScheduleWork] :
2       (RPF: r0e0)
3       (msgCount[RPF]: 0)
4       (nw: NW)
5       (tran: T)
6       =>  (RPF: r0e0)
7           (msgCount[RPF]: 1)
8           (nw: (msg(B2R, schedulework) NW))
9           (tran: sendScheduleWork(RPF)) .
10 rl [sendGoToCharge] :
11      (RPF: r0e0)
12      (msgCount[RPF]: 0)
13      (nw: NW)
14      (tran: T)
15      =>  (RPF: r0e0)
16          (msgCount[RPF]: 1)
17          (nw: (msg(B2R, gotocharge) NW))
18          (tran: sendGoToCharge(RPF)) .
```

The results of the new model checking are as follows (Figure 5.7 and 5.8). *fair1* is a strong fairness assumption used to assume that the command to control the robot to the elevator will eventually be sent. If it is not send, there is a possibility that other working commands have been sent to the robot, with never enter the state transition $robq_0 \rightarrow robq_1$.

```
Maude> search ic =>! C:Config .
search in RCP-FORMULA : ic =>! C:Config .

No solution.
states: 85  rewrites: 104 in 1ms cpu (1ms real) (81377 rewrites/second)
```

Figure 5.7: Problem 5 Deadlock (New Model).

```
Maude> red in RCP-FORMULA : modelCheck(ic,fair1 -> liveness) .
reduce in RCP-FORMULA : modelCheck(ic, fair1 -> liveness) .
rewrites: 622 in 1ms cpu (2ms real) (339519 rewrites/second)
result Bool: true
```

Figure 5.8: Problem 5 Liveness (New Model).

5.5 Problem 6: Expression is Always True

In order to avoid such problems, in the model check, we added some member variables used in the emulator to the rewriting rule. These member variables

represent whether the robot control subsystem receives a reply to certain messages, and it is the improper management of these variables that leads to the existence of Problem 6. We formally specified such variables in Maude, as illustrated below.

The new initial configuration is:

```
eq ic = (ELV: elvq0)
        (movingStatus[ELV]: stay)
        (BOS: bosq)
        (RPF: r0e0)
        (RPF[GOTOELVSUCCESS]: false)
        (RPF[GOTOELVCOMPLETED]: false)
        (RPF[CALLINGSUCCESS]: false)
        (RPF[INTERLOCKTRUESUCCESS]: false)
        (RPF[OPENSUCCESS]: false)
        (RPF[CALLACCEPT]: false)
        (RPF[CALLARRIVE]: false)
        (RPF[GETTINGONSUCCESS]: false)
        (RPF[GETTINGONCOMPLETED]: false)
        (RPF[CLOSESUCCESS]: false)
        (RPF[GOACCEPT]: false)
        (RPF[GOARRIVE]: false)
        (RPF[GETTINGOFFSUCCESS]: false)
        (RPF[GETTINGOFFCOMPLETED]: false)
        (RPF[INTERLOCKFALSESUCCESS]: false)
        (msgCount[RPF]: 0)
        (ROB: robq0)
        (movingStatus[ROB]: stay)
        (nw: void)
        (tran: notran) .
```

An example of new rewriting rules follows:

```
rl [sendInterlock1] :
   (RPF: r1e0)
   (msgCount[RPF]: 0)
   (RPF[INTERLOCKTRUESUCCESS]: false)
   (nw: NW)
   (tran: T)
   => (RPF: r1e0)
      (msgCount[RPF]: 1)
      (RPF[INTERLOCKTRUESUCCESS]: false)
      (nw: (msg(B2D, interlock1) NW))
      (tran: sendInterlock1(RPF)) .

rl [recvinterlocksuccess1] :
   (RPF: r1e0)
   (msgCount[RPF]: 1)
   (RPF[INTERLOCKTRUESUCCESS]: false)
   (nw: (msg(D2B, interlocksuccess1) NW))
```

```
18      (tran: T)
19      => (RPF: calling)
20         (msgCount[RPF]: 0)
21         (RPF[INTERLOCKTRUESUCCESS]: true)
22         (nw: NW)
23         (tran: recvInterlockSuccess1(RPF)) .
```

The new model checking result of the liveness property is shown in Figure 5.9.

```
Maude> red in RCP-FORMULA : modelCheck(ic,liveness) .
reduce in RCP-FORMULA : modelCheck(ic, liveness) .
rewrites: 266 in 1ms cpu (1ms real) (148024 rewrites/second)
result Bool: true
```

Figure 5.9: Problem 6 Liveness Property.

5.6 Experiment Summary

In this chapter, we have modified our formal model for 5 of the 10 problems we identified in Chapter 4.

Table 5.2: Model Checking Result Overview

	Deadlock	Safety	Liveness
Single Robot Version	✓		✓
Multi Robot Version	✓	✓	✓
Model Problem 2	✓		✓
Model Problem 3	✓		✓
Model Problem 4	✓		✓
Model Problem 5	✓		✓
Model Problem 6	✓		✓

Among them, problems 2-4 are actually caused by abnormal message transmission, which is a situation that was not considered in the protocol designed. We needed additional handling of these situations, therefore we modified the state transition of the basic version, and added a certain ability of abnormal message detection. When modifying the model, we added some observable components that indicate abnormal status, and added some new rewriting rules.

Problems 5-6 are some errors that were omitted in formalizing but existed in the emulator. Among them, problem 5 is that we only specified the state

transition related to the interaction between the robot and the elevator when formalizing. In order to model checking for problem 5, we added several new rewriting rules to represent other commands of the robot. Problem 6 is a programming error in the specific implementation. We added some member variables of the protocol class in the old rewriting rules. So far, we have created a total of 7 models and checked total 15 properties for different requirements, as shown in Table 5.2. It can be seen that it is not troublesome to recreate a new model, especially as long as if we know what state causes the error, we can easily locate the rewrite rules that need to be modified.

Chapter 6

Discussion

6.1 Strengths and Shortcomings

Chapter 4 showed the complementary nature of model checking and fuzzing. On the one hand, no exception related to the verified properties occurred during fuzzing, which highlights the effectiveness of model checking in detecting potential logical problems. On the other hand, the problems found through fuzzing all of which were not found during model checking, which highlights the importance of fuzzing in detecting potential operational problems.

Chapter 5 demonstrated that the combination of model checking and emulation is not only helpful in verifying protocol properties and finding problems, but also very convenient to modify and reverify because the two use the same state transition modeling. In the modification of the model after the problem is found, because the specific error state is known in the emulator, we can quickly locate the part in Maude that needs to be modified. When modifying, most work focuses on the protocol problem state adding some new observable components, corresponding functional modules, rewrite rules and assumptions, and do not go to great lengths.

One shortcoming is that when the emulator was developed, fuzzing module wasn't considered. In particular, in the case of the fuzzing of the robot control subsystem, the emulator can input the same seed and output the same results for a long time, but in the case of the fuzzing of other subsystems cannot do this, while inputting the same seed can only generate the same initial environment, and the mutations will become different quickly. The guess is that the subsystems in the emulator are implemented by threads, and the random trigger order of the threads leads to such a result. In the robot control subsystem, the robot control protocol has low level rely on the trigger order of other subsystems. It will not respond immediately after

receiving the messages, but stores the value of the received message and responds in the dedicated module according to the value. However, other subsystems respond immediately after receiving the message, or because of this difference lead to such the problem.

Finally, we had to implement the system twice in system modeling and emulation according to the workflow shown in Figure 3.1. However, ideally, the implementation should be done only once, as discussed next.

6.2 Automated Translation from Emulator Runtime Data to Maude Code

As a preliminary idea, we propose a method to generate Maude code about part of the model of the basic version of the protocol shown in Sect. 3.1.1 from the emulator. This approach is shown in Figure 6.1.

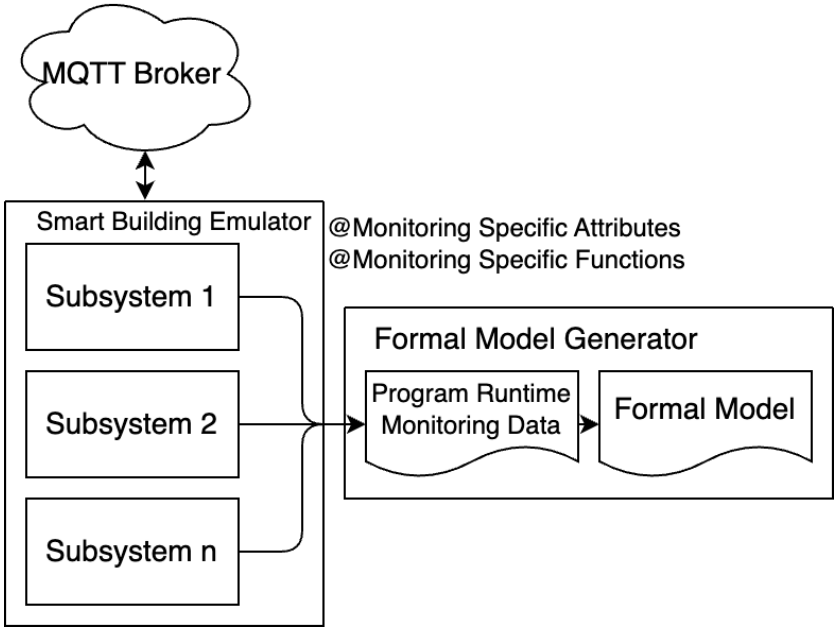

Figure 6.1: Formal Model Generator.

We can use a decorator to monitor the attribute changes and function calls. In order to successfully generate a formal model, we need to monitor the following aspects:

1. Attributes that we want to model. These attributes correspond to all other observable components except (nw: $msgs$) and (tran: t).

2. The function of sending messages. These correspond to the observable component (nw: $msgs$).
3. Received Message. These also correspond to the observable component (nw: $msgs$).

During the emulator running, the decorator will save the information about the order of function calling or attribute changing, which contains the information of state transition. We can use Algorithm 6.1 to extract the state transition information needed to generate a formal model.

Algorithm 6.1 Export State Transition

Input: D = $\{d_0, d_1, ..., d_n\}$ // Program Runtime Monitoring Data
Output: T // State Transition

```
1: set T = empty list
2: set tran = empty list
3: for d = d_0 to d_n do
4:    if d_mode == message then
5:       for d' = d_next to d_n do
6:          if d'_name == d_name then
7:             if d'.mode == message then
8:                set T = T ∪ tran
9:                set tran = empty list
10:               break;
11:            else
12:               set tran = tran ∪ d
13:            end if
14:         end if
15:      end for
16:   end if
17: end for
```

After a subsystem sends or receives messages, the member variables will be changed. The goal is to export the changes. The algorithm will traverse all the program runtime monitoring data and collect the parts involved in message sending or receiving from the same subsystem.

The set of all attributes values is lS, the set of the attributes values that are first assigned is lI, the set that the algorithm 6.1 exported is lT, the lE in this paper are temporarily replaced by serial numbers. The Maude code generated in this way can be successfully checked with liveness property formula similar to basic version of the protocol in Sect. 3.2.2 (see Figure 6.2).

An example of the generated rewriting rules is as follows:

```
1  rl [tran0] :
2       (RCP.retries: 0)
3       (RCP.rcpstatus: r0e0.RCP.rcpstatus)
4       (nw: NW)
5       (tran: T)
6       =>
7       (RCP.retries: 1)
8       (RCP.rcpstatus: r0e0.RCP.rcpstatus)
9       (nw: (msg(GoToELV.ROB.recvcmd) NW))
10      (tran: tran0) .
```

```
Maude> red in RCP-FORMULA : modelCheck(ic,liveness) .
reduce in RCP-FORMULA : modelCheck(ic, liveness) .
rewrites: 171 in 0ms cpu (1ms real) (202606 rewrites/second)
result Bool: true
```

Figure 6.2: Model Checking Result of Liveness Property for the Automatically Generated Model.

However, this method can only export state transitions on the surface. For example, the robot waiting queue can be modeled within our handcrafted models, while cannot be modeled through this method. This method can only be used at the beginning part to help us generate the simplest version of the model. In the analysis, improvement and reverification part after step 6 in Figure 3.1, since it may be necessary to add new functions to adjust the model to satisfy different requirements, the method may become unworkable.

Chapter 7

Conclusion and Future Work

7.1 Conclusion

We applied the IoT System Trustworthiness Levels, focusing on the implementation of TAL3, followed the workflow mentioned in Chapter 3 to analyze the operations of the robot control protocols and entities of a smart building, and constructed 3 state machine models of 4 subsystems. Based on the state machine models, we conducted formal verification and experimental verification. Then, we comparatively analyzed these results, proposed improvement plans, and re-verified the protocols, which reflects the flexibility of our method.

Our formal verification method involves using Maude to formally specify the protocols. We conducted model checking to check whether the protocols meet specific properties. We used Maude search command and Maude LTL Model Checker, applied weak fairness assumptions, strong fairness assumptions, and the divide and approach, specified different LTL formulas to verify the deadlock, safety and liveness properties according to varying requirements for a total of 7 models of robot control protocols. The formal verification results showed that the basic version and the improved versions all meet requirements, which helps us assure the logical correctness of the protocol.

Our experimental verification method involves using an emulator to emulate the protocols and simulate entities under the state machine models. The emulation of the protocol makes the experimental verification not merely use the model-based techniques but rather close to the real system, which can be a good auxiliary for model checking to make the models of the system better. The simulation of entities help the problems found in the experimental verification easier to locate in the models, which also allows

emulator be a good auxiliary for model checking.

After the emulator was developed, we conducted fuzzing to test whether the protocols still have unexpected problems. Our fuzzing focus on mutating messages related to the message sending and receiving about the protocol. We applied two methods to speed up the fuzzing. By applying our strategies, although the improvement of the three subsystems of robots, building OS and elevators is not obvious, the coverage of the fuzzing of 30 seeds of the robot control platform subsystem has increased from 36% to 94%. We summarized 3 types of emulator abnormal behavior identified via fuzzing. By observing the abnormal behaviors and analyzing the code, we identified 10 types of problems, most of the improvement plans based on avoiding these problems, which helps us assure the operational correctness. Part of the problems caused by mistakes such as poor consideration in programming, and the other comes from abnormal situation such as data tampering, all of which problems were not considered in modeling. The result comparison of model checking and fuzzing reflects the complementary nature of our method.

7.2 Future Work

Our future work will focus on optimizing testing strategies. In current fuzzing implementations, the fuzzer generates input data based on a predefined range, which remains constant throughout the entire testing cycle. Each test begins by generating input from the same initial bytes and mutates on the basis of this initial input. However, the initial input generated in each round is same, leading to repetitive first inputs and redundant mutation processes. For example, in the testing of a robot control subsystem, the process always starts by sending the command **interlock(false)**. Furthermore, the mutation process always begins from the very start. This issue is particularly noticeable when testing robot and elevator subsystems. The testing frequently simulates that the robot or elevator is failure, which resulting in low coverage of the testing. As future work, it may be worth exploring dynamic adjustments to the input generation range. For example, instead of always starting from the same initial state, the test could begin from a specific state after sending a sequence of correct, non-fuzzed messages. This approach could help focus the testing on unexplored paths and improve efficiency and code coverage.

Secondly, the emulator capabilities could be improved. One direction is try to solve the problem of having the same seed input but different outputs. A possible solution is to change from using threads to coroutines. Another possible direction is to insert a bug in emulator that cannot be found by

experimental verification but can be found by formal verification. In our formalized model, the order of messages in the network is not preserved, but in our emulator, messages are always received in order. Adding disordered message transmission to the emulator can not only further improve our model according to the behaviors of the emulator, but also help to explore more potential problems.

Finally, we will focus on merging some duplicate work in the workflow. We already tried to merge Modeling and Emulation in this thesis, but the proposed method is not perfect. A possible direction refers to letting the method become workable in all parts of the workflow (see Figure 7.1).

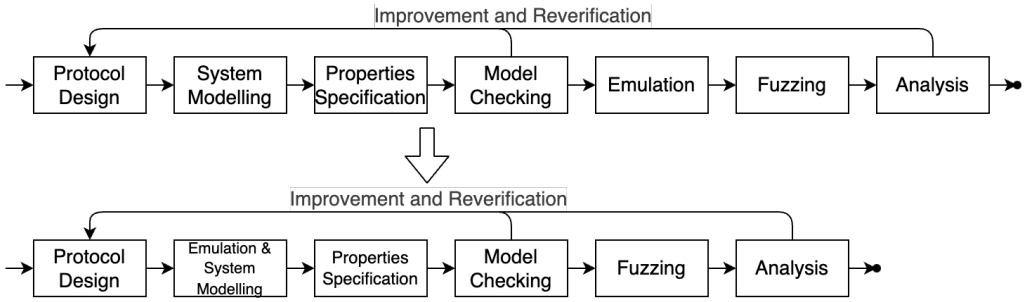

Figure 7.1: Future Work.

NOTE: Generative AI technology was used for translation and grammar checking during the preparation of this thesis.

Publications

[1] J. Wu, R. Beuran, "Formal and Experimental Verification of Robot Control Protocols for Smart Buildings", Symposium on Cryptography and Information Security (SCIS 2025), Kokura, Japan, January 28-31, 2025.

References

[1] C. Morales-Gonzalez, M. Harper, M. Cash, L. Luo, Z. Ling, Q. Z. Sun, and X. Fu, "On building automation system security," *High-Confidence Computing*, vol. 4, no. 3, p. 100236, 2024. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667295224000394

[2] Y. Sun, T.-Y. Wu, X. Li, and M. Guizani, "A rule verification system for smart buildings," *IEEE Transactions on Emerging Topics in Computing*, vol. 5, no. 3, pp. 367–379, 2017.

[3] R. Beuran, S. E. Ooi, A. O. Barbir, and Y. Tan, "Iot system trustworthiness assurance," in *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ser. ASIA CCS '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 1222–1224. [Online]. Available: https://doi.org/10.1145/3488932.3527287

[4] J. Yang, S. Arya, and Y. Wang, "Formal-guided fuzz testing: Targeting security assurance from specification to implementation for 5g and beyond," *IEEE Access*, vol. 12, pp. 29 175–29 193, 2024.

[5] M. Ammann, L. Hirschi, and S. Kremer, "Dy fuzzing: Formal dolev-yao models meet cryptographic protocol fuzz testing," in *2024 IEEE Symposium on Security and Privacy (SP)*, 2024, pp. 1481–1499.

[6] C. Baier and J.-P. Katoen, *Principles of model checking*. MIT press, 2008.

[7] K. Ogata, "A divide & conquer approach to liveness model checking under fairness & anti-fairness assumptions," *Frontiers of Computer Science*, vol. 13, pp. 51–72, 2019.

[8] M. Liu, D. D. Bui, D. D. Tran, and K. Ogata, "Formal specification and model checking of an autonomous vehicle merging protocol," in *2021*

IEEE 21st International Conference on Software Quality, Reliability and Security Companion (QRS-C), 2021, pp. 333–342.

[9] M. Clavel, F. Durán, S. Eker, S. Escobar, P. Lincoln, N. Martı-Oliet, J. Meseguer, R. Rubio, and C. Talcott, "Maude manual (version 3.1)," *SRI International University of Illinois at Urbana-Champaign http://maude. lcc. uma. es/maude31-manual-html/maude-manual. html*, 2020.

[10] H. E. Degha, F. Z. Laallam, O. Kazar, I. Khelfaoui, B. Athamena, and Z. Houhamdi, "Open-sbs: Smart building simulator," in *2022 International Arab Conference on Information Technology (ACIT)*, 2022, pp. 1–15.

[11] V. J. Manès, H. Han, C. Han, S. K. Cha, M. Egele, E. J. Schwartz, and M. Woo, "The art, science, and engineering of fuzzing: A survey," *IEEE Transactions on Software Engineering*, vol. 47, no. 11, pp. 2312–2331, 2019.

[12] Google, "Atheris: A coverage-guided, native python fuzzer," https://github.com/google/atheris, 2020, accessed: 2025-01-18.

[13] ——, "How the Atheris Python fuzzer works," https://security.googleblog.com/2020/12/how-atheris-python-fuzzer-works.html, 2020, accessed: 2025-01-18.

[14] X. Weng and R. Beuran, "Smart building control system emulation platform for security testing," *29th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2024)*, pp. 220–223, 2024.

[15] "Coverage.py," https://coverage.readthedocs.io/en/7.6.9/, accessed: 2025-01-18.

[16] MITRE Corporation, https://cwe.mitre.org, accessed: 2025-01-18.