JAIST Repository
https://dspace.jaist.ac.jp/

Title	証明スコア法による形式検証の調査研究及び新規事例 [課題研究報告書]
Author(s)	浅江, 尚輝
Citation	
Issue Date	2021-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/17548
Rights	
Description	Supervisor:緒方　和博, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Master's Research Project Report

An investigation of the proof score approach to formal verification and a new case study with the approach

Naoki Asae

Supervisor Kazuhiro Ogata

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

September, 2021

Abstract

Today, software systems permeate every part of our lives. Software systems have become an indispensable part of our lives. However, the number of reports of system failures caused by software systems has been increasing more and more. While the technology to develop software systems is advancing day by day, the technology to assure the quality of the software systems is not keeping pace. Since software systems are expected to play an increasingly important role in the future, there is an urgent need to establish new quality assurance techniques. One of the techniques to assure the quality of software systems is formal method. Formal methods can be divided into formal specification and formal verification, and formal verification can be further divided into model checking and theorem proving. Formal methods are one of the most promising techniques to assure the quality of software systems, but they are not yet widely used in software development. To make it popular, we need to increase the number of cases where formal methods are applied. Mutual exclusion is an important issue in the creation of secure software systems. The mechanism to achieve mutual exclusion is called mutual exclusion protocols. In this study, we formally verify that three mutual exclusion protocols (TAS protocol, Qlock protocol, and Anderson protocol) enjoy the mutual exclusion property. TAS protocol uses an atomic instruction test & set, Qlock protocol is an abstract version of the Dijkstra binary semaphore, and Anderson protocol is an array-based mutual exclusion protocol. For each protocol, the formal verification is conducted in two ways: (1) by writing proof scores in CafeOBJ, an algebraic specification language, and (2) by using CafeInMaude Proof Generator (CiMPG) and CafeInMaude Proof Assistant (CiMPA). Proof scores are programs written in CafeOBJ to conduct formal proofs. CafeInMaude is the world's second implementation of CafeOBJ in Maude that is a sister language of CafeOBJ. CafeInMaude is equipped with CiMPG and CiMPA. CiMPG takes proof scores and generates proof scripts that can be fed into CiMPA. While conducting the formal verification of Anderson by writing proof scores in CafeOBJ, we encountered a situation such that our proof attempt did not seem to be convergent: it seemed necessary to us an infinite number of similar lemmas. To tackle the situation, we have introduced an auxiliary variable into Anderson, where an auxiliary variable does not affect the behaviors of Anderson. We describe the situation in detail in the report. We have learned some lessons from the case studies and summarize the lessons in the report. In particular, by tackling the formal verification of mutual exclusion protocols with two different

approaches, manual proof by proof scoring and automatic proof by using CafeInMaude, I was able to understand the advantages and disadvantages of each approach. And, I was able to understand firsthand why they have not yet penetrated into the field of software system development, Although formal methods have been attracting attention as an effective technique for quality assurance of software systems. Since the purpose of this case study was to find out the reason, this is the most important lesson I learned from this case study. We also mention some pieces of our future work. For example, we do not know whether it is mandatory to introduce an auxiliary variable into Anderson so that we can formally verify that Anderson enjoys the mutual exclusion property. If so, we would like to clarify why we need to do so. Otherwise, we would like to complete the formal verification of Anderson without introducing any auxiliary variables.

Keywords: software quality assurance, formal methods, theorem proving, mutual exclusion protocols, auxiliary variable

Contents

1 Introduction **1**
- 1.1 Motivation 1
- 1.2 Report Outline 2

2 Preliminaries **4**
- 2.1 Formal methods 4
  - 2.1.1 Formal specification 4
  - 2.1.2 Formal verification 4
- 2.2 CafeOBJ 5
- 2.3 State machines 8
- 2.4 Tools for automatic proof 8
  - 2.4.1 Maude and CafeInMaude 8
  - 2.4.2 CiMPA 8
  - 2.4.3 CiMPG 10
- 2.5 Mutual exclusion protocols 10

3 Test and Set protocol **12**
- 3.1 Formal specification of TAS protocol 12
- 3.2 Formal verification of TAS protocol by proof scoring 14
- 3.3 Formal verification of mutual exclusion property of TAS protocol using CiMPA and CiMPG 16

4 Qlock protocol **19**
- 4.1 Formal specification of Qlock protocol 19
- 4.2 Formal verification of Qlock protocol by proof scoring 25

5 Anderson protocol **29**
- 5.1 Formal specification of Anderson 29
- 5.2 Formal verification of Anderson by proof scoring 34
- 5.3 Formal verification by introducing an auxiliary variable 35

5.4 Formal verification of mutual exclusion property of Anderson using CiMPA and CiMPG 39

6 Lessons Learned **46**
6.1 Functional programming 46
6.2 Date structure 47
6.3 Formal method 47
6.4 Proof scoring 48
6.5 Visualization of the system 48
6.6 Difference between manual and automatic 49

7 Conclusion **50**
7.1 Summary 50
7.2 Future work 53

List of Figures

3.1 Initial state of TAS protocol 13
3.2 Transition RS to CS of TAS protocol 13
3.3 Transition CS to RS of TAS protocol 14

4.1 Initial state of Qlock protocol 20
4.2 Initial state of queue 20
4.3 Operation of *enq* 21
4.4 Operation of *enq* - 2 21
4.5 Operation of *deq* 22
4.6 Operation of top 22
4.7 Operation of *in* 23
4.8 Transion from rs to ws of Qlock protocol 23
4.9 Transion from ws to cs of Qlock protocol 24
4.10 Transion from cs to rs of Qlock protocol 24

5.1 Initial state of Anderson protocol 30
5.2 Transition from rs to ws of Anderson protocol 31
5.3 Transition from ws to cs of Anderson protocol 31
5.4 Transition from rs to ws of Anderson protocol - 2 32
5.5 Transition from rs to ws of Anderson protocol - 3 32
5.6 Transition from cs to rs of Anderson protocol 33
5.7 Example of case splitting 34

List of Tables

Chapter 1

Introduction

1.1 Motivation

The role of software in information systems has increased with the emergence of fundamental Information Communication technologies such as artificial intelligence and the Internet of Things. Today, many things such as various transportation systems and financial systems, are controlled by (computer) software, and it is no exaggeration to say that the quality of human lives depends on the quality of the software. It would not be surprising if the quality of human lives depends on the quality of the software itself, as our dependence on it becomes stronger and stronger in the future. Meanwhile, reports of software-induced system failures continue to pour in: according to the IPA, as many as 60 domestic information system failures were reported in the six months from July to December 2019. Mizuho Bank's ATM failure and Docomo account fraudulent withdrawals are still fresh in our minds. In other words, while the technology for developing software is advancing day by day, the technology for guaranteeing the quality of software has not kept pace. Methods and methodologies have been developed to solve the software crisis, such as object orientation, integrated development environments, agile software development, version control systems, etc. However, it is said that there is no essential solution.

The purpose of this study is to investigate the proof scoring and case studies conducted with the proof scoring and to conduct a new case study by applying the proof scoring to a new example. Through this, we will also try to identify barriers to the penetration of the method in the field and to suggest improvements to remove them. Software system quality assurance is considered to be one of the most important processes in the software system development life cycle. Formal methods have been attracting atten-

tion as a promising method for software system quality assurance. However, the lack of case studies and literature (especially in Japanese), as well as a large number of formal specification languages and tools, still hinders the introduction of formal methods in software development. To promote the new introduction of formal methods, the effectiveness of formal methods should be demonstrated through case studies. In particular, few case studies have demonstrated the effectiveness of formal verification compared to formal specification. Therefore, this research focuses on the proof score method, which is one of the theorem proving methods, and investigates the case studies conducted in the past, and conducts a new case study. By achieving the purpose of this research, we can show the effectiveness of theorem proving, and it is expected to help theorem proving to spread in the field of software development. And it can contribute to ensuring a quality of software systems.

1.2 Report Outline

The following is a report on the research project.

- Chapter 1: Introduction
  This chapter will introduce the overview, aims, and significance of the thesis.

- Chapter 2: Preliminaries
  This chapter will describe the techniques required and the tools used in the study.

- Chapter 3: Test and Set protocol
  This chapter will describe the specification and proof scores of Test and Set(TAS) protocol, a type of mutual exclusion protocol.

- Chapter 4: Qlock protocol
  This chapter will describe the specification and proof scores of Qlock protocol, a type of mutual exclusion protocol.

- Chapter 5: Anderson protocol
  This chapter will describe the specification and proof scores of Anderson, a type of mutual exclusion protocol. We will also describe the problems that prevented Anderson's verification and the solutions to them.

- Chapter 6: Lessons Learned
  This chapter will describe what learned through this study.

- Chapter 7: conclusion
  This chapter, the summary of the research report and future work will be discussed.

Chapter 2

Preliminaries

2.1 Formal methods

One of the leading techniques to ensure software quality is formal method. It is expected that the application of formal methods to software design and mathematical analysis will improve the reliability of the design. It is considered to be a promising technology to guarantee the quality of software. On the other hand, it is difficult to accurately estimate the cost of implementation and the resulting performance, so it is not yet widely used in development. Formal methods consist of formal specifications and formal verification.

2.1.1 Formal specification

A formal specification is a description of the system specification in a formal language. By describing the specification in a formal language, it is possible to determine whether the system has been built according to the specification (formal verification). It also has the advantage that inconsistencies in the specification can be detected during the specification writing process, preventing costly rework later in the development process. The effectiveness of formal specifications has already been demonstrated in the development of Felica firmware in Japan.

2.1.2 Formal verification

Formal verification has not yet penetrated the field. Formal verification can be roughly divided into model checking and theorem proving. Model-checking has the advantage of automatic verification, but it requires cramming the software to be verified into a small space. To truly guarantee the

quality of software, it is necessary to use theorem proving. A proof score is a proof or plan of proof written in an algebraic specification language. A theorem-proof that is performed by writing a proof score is called proof scoring.

Theorem proving

Mathematical proofs of the target specification guarantee that the specification is met in all cases, even if no test cases are chosen.Mathematical proofs of the target specification guarantee that the specification is satisfied in all cases, even if no test cases are chosen. However, unlike model checking, theorem proving is difficult to automate and has little practical use.

Proof scoring

Proof score is a proof or plan of proof written in an algebraic specification language. Theorem proving carried out by writing a proof score is called the proof scoring. As proof scores are written by hand, there is a risk of human error.

A proof score contains mainly the following:

- Specification of the module to be used in the proof.
- constants that represent arbitrary elements of some sort (i.e. behave like variables)
- case splitting
- lemma
- Inductive assumptions (if induction is used)
- reduction instructions

2.2 CafeOBJ

CafeOBJ is an algebraic specification language developed for formal specification and formal verification[1, 2, 3, 4]. It is possible to perform interactive verification by interpreting written equations as rewrite rules and executing them. The grammar of CafeOBJ is as follows:

- module
  To write a specification in CafeOBJ, first, declare a module, and then write various declarations in it. A module is declared using `module` (or its abbreviation `mod`) as follows:

```
mod <module-name> {
<module-elements>
}
```

- Sort
  In algebra, sort is a concept that corresponds to type in programming languages.
  A sort is declared using [...]:

```
[ <sort-name> ... <sort-name> ]
```

- Import
  Allowing one module to use the declarations of another predefined module is called importing a module.
  Protecting, extending, and using (and their abbreviations `pr, ex,` and `us`) are used to import modules:

```
pr(<module-exp>)
ex(<module-exp>)
us(<module-exp>)
```

- Operator
  Operators are declared using `op:`

```
op <op-name> : <arity> -> <sort-of-op>  <operator-attribute>
```

  <op-name> is the operator name, <arity> is the arity, and <sort-of-op> is the sort of operator. The arity is a concept corresponding to the argument, which is a sequence of sort names. The sort-of-operator `*` is a concept corresponding to the type of the return value.

- equation
  Let `T` be the set of terms created from an index (a pair of sort and

operator declarations). An equation declares an equivalence relation between two terms t1, t2 $\in$ T. The equations described in the specification are used in reasoning as axioms.

Equations are declared using `eq`:

```
eq <lhs> = <rhs> .
```

`<lhs>` and `<rhs>` are terms on the same sort. These terms can contain variables. Variables are declared on a particular sort and can be assigned to any term on that sort.

- variable
  As mentioned earlier, variables can be declared in equations. However, it is convenient to declare frequently used variables together in advance. The scope of variables declared in this way is within the module in which they are declared.

  To declare a variable outside of an equation, use `var`:

  ```
  var <var-name> : <sort-name>
  ```

  Multiple variables of the same sort can be declared using `vars`:

  ```
  vars <var-name> ... <var-name> : <sort-name>
  ```

- reduction
  In CafeOBJ, reduction means to consider the declared equation as a rewriting rule from the left side to the right side and to rewrite the given terms one after another using the rewriting rules (the order in which the rules are applied is arbitrary) to make the expression as simple as possible. A term that cannot be rewritten anymore using any rewriting rule is called a normal form. Simplification is equational reasoning, which uses equations in only one direction.

  The command red simplifies the term `<term>`:

  ```
  red <term> .
  ```

2.3 State machines

A state machine $\mathcal{M} \triangleq (\mathcal{S}, \mathcal{I}, \mathcal{T})$ consists of a set $\mathcal{S}$ of states, a set $\mathcal{I} \subseteq \mathcal{S}$ of initial states and a binary relation $\mathcal{T} \subseteq \mathcal{S} \times \mathcal{S}$ over states. The set $\mathcal{R}$ of reachable states with respect to $\mathcal{M}$ is inductively defined as follows: (1) for each $s \in \mathcal{I}$, $s \in \mathcal{R}$ and (2) for each $(s,\acute{s}) \in \mathcal{T}$, if $s \in \mathcal{R}$, then $\acute{s} \in \mathcal{R}$. A state predicate p is an invariant property with respect to $\mathcal{M}$ if and only if $p(s)$ holds for all $s \in \mathcal{R}$.

This report uses observational transition systems (OTSs) that are state machines such that states are recognized by values observed through observer functions, state transitions are represented by transition functions, and state transitions are defined in terms of equations that define how state transitions change the values observed through observer functions.

Let `init` denote an arbitrary initial state and o : Sys D_{o_1} ... D_{o_n} -> D_o be one observer function. `init` for o is defined as follows:

$$\texttt{eq}\ o(\texttt{init}, X_1, ..., X_n) = \mathit{val\text{-}o}(X_1, ..., X_n).$$

This means that the initial value observed by o with X_1, ..., X_n is $\mathit{val\text{-}o}(X_1, ..., X_n)$. Let t : Sys D_{t_1} ... D_{t_m} -> Sys be one transition function. t for o is defined as follows:

$$\texttt{ceq}\ o(t(S, Y_1, ..., Y_m), X_1, ..., X_n) = \mathit{val\text{-}o\text{-}t}(X_1, ..., X_n, Y_1, ..., Y_m)\ \texttt{if}\ \mathit{cond\text{-}t}(S, Y_1, ..., Y_n)\ .$$

This means that if the condition $\mathit{cond\text{-}t}(S, Y_1, ..., Y_n)$ holds, t changes the value observed by o to $\mathit{val\text{-}o\text{-}t}(X_1, ..., X_n, Y_1, ..., Y_m)$.

2.4 Tools for automatic proof

2.4.1 Maude and CafeInMaude

Maude[5] is a kind of algebraic specification language. CafeInMaude[6] is a tool for introducing specifications written in CafeOBJ into the Maude system.

2.4.2 CiMPA

CafeInMaude Proof Assistant(CiMPA)[6] allows users to prove properties on their CafeOBJ specifications. This is an automatic theorem proving system. While it has the advantage of automatic theorem proving, it has the disadvantage of being less flexible than proof scoring. The available commands, that must be used inside open-close environments, are:

- `:goal`(EqS).
  This command introduces a new goal composed of the equations in *EqS*. In this study, An equation is used that implies that each invariant is replaced by true.

- `:ind on` (V).
  This command indicates that induction will be carried out on the variable V. In this study, we will particularly focus on the execution of the variable S of type `sys`.

- `:def` $LAB =$ `:ctf`$\{Eq\}$.
  This command maps the label LAB to the equation Eq, so it can be later used for case distinction.

- `:def` LAB `= :ctf[`T`]`
  This command maps the label LAB to the term T, so it can be later used for case distinction.

- `:apply`(si).
  This command applies simultaneous induction on the variable V set with command `:ind` to the current goal. Given the sort S of V, it generates as many new goals as the numbers of constructors defined for `S` (including sub sort relations) in the current module. In the formal verification of mutual exclusion protocols, goals are generated for `init`, which means the initial state, and the number of sections defined in each protocol.

- `:apply`(tc).
  This command applies the theorem of constants to the current goal. It generates as many new goals as sentences stated in the goal.

- `: apply` (rd).
  This command reduces the current goal using equations. If all the sentences in the goal are reduced to true then the goal is proven. This command substitutes the current goal with a new one with the sentences reduced. Corresponds to `red` in a proof score.

- : `apply`(LAB).
  This command uses the equation or term associated with LAB for case distinction in the current goal. If it is an equation it generates two new goals, the first one stating the equation holds and the second one stating the equation does not hold. If it is a term with sort `S` it generates as many new goals as constructors are defined for `S` in the current module; each goal states that the term is equal to a specific constructed term.

- `:imp`[LAB] .
  This command takes the equation identified by the label LAB, which must be defined in the current module and must have true as a right-hand side, and generates a new goal where the lefthand side of the equation implies the former goal. In formal verification of mutual exclusion protocols, it is used to apply lemma.

- `:sel(G).`
  This command selects the goal G as the next one.

2.4.3 CiMPG

CiMPG stands for Cafe in Maude Proof Generator[6]. This can be used to generate proof scripts from proof scores. By inputting the proof scripts into CiMPA, the advantages of both proof scoring and CiMPA can be used for verification.

2.5 Mutual exclusion protocols

In a system, if there is a resource that is shared by multiple processes (or agents), it is required that there is at most one process accessing the resource at all times. Preventing other processes from using a resource while allowing one process to use it is called mutual exclusion, and the procedure/mechanism for achieving mutual exclusion is called mutual exclusion protocol. In the mutual exclusion protocol, the initial position of each process is at the Reminder Section(RS), and when using the shared resources, it enters the Critical Section(CS). After that, it returns to the RS when it finishes using the shared resource. Each process repeats the above steps. The property that there is always at most one process in CS is called mutual exclusion property. This

is the property that a mutual exclusion protocol should satisfy and should be verified using formal verification.

Chapter 3

Test and Set protocol

This chapter describes the specification and formal verification of TAS protocol. Formal verification was tackled using two approaches: proof scoring and automatic proof using CiMPG and CiMPA.

3.1 Formal specification of TAS protocol

Test and Set (TAS) is the simplest mutual exclusion protocol. TAS protocol written in Algol-like pseudo-code is as follows:

loop{
"Remainder Section"
rs : **repeat while** test&set(*locked*);
"Critical Section"
cs : *locked* := `false`;
}

We define the variable *locked*, whose value is `true` or `false`. The initial value of *locked* is `false`. As soon as a process transitions from RS to CS, it rewrites the value of *locked* to `true`. When a process transitions from RS to CS, it rewrites the value of *locked* from `true` to `false`, and while the value of *locked* is `true`, other processes in the RS cannot transition to CS.

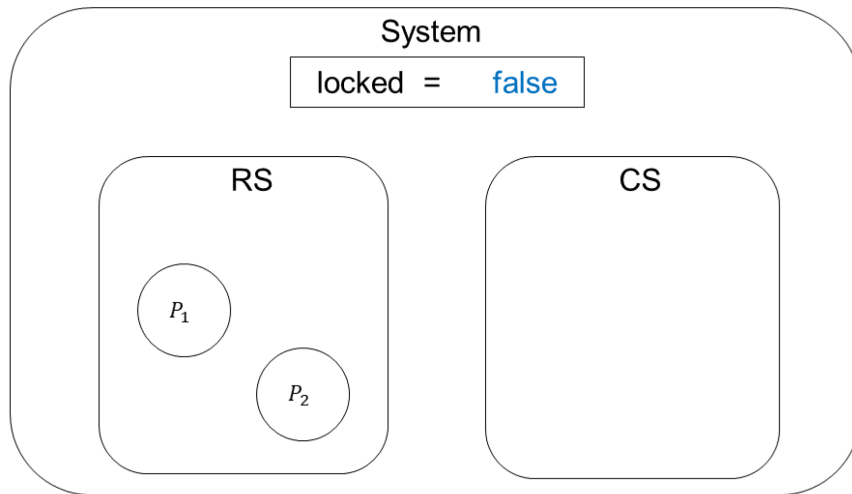

Figure 3.1: Initial state of TAS protocol

The initial value is `false`. The initial position of the processes is RS .

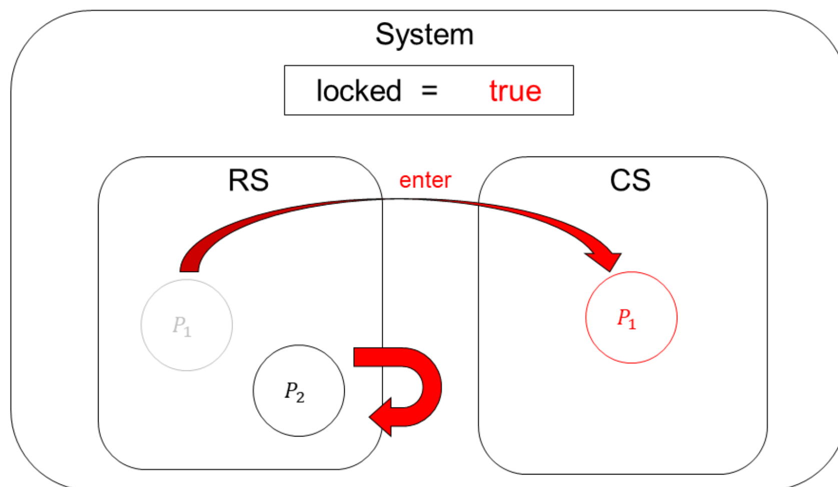

Figure 3.2: Transition RS to CS of TAS protocol

When the process transitions from RS to CS, rewrite the value of *locked* from `false` to `true`. While the value of *locked* is `true`, other processes stay in RS.

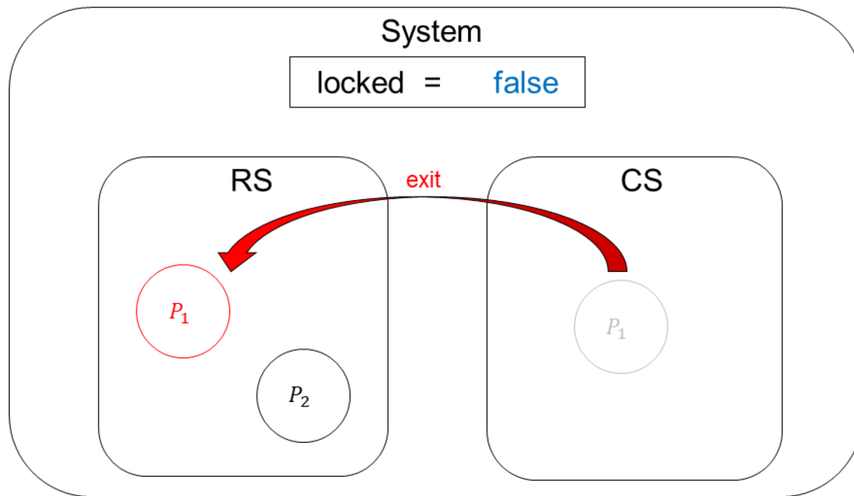

Figure 3.3: Transition CS to RS of TAS protocol

When the process transitions from CS to RS, rewrite the value of *locked* from `true` to `false`.

3.2 Formal verification of TAS protocol by proof scoring

In verification of mutual exclusion property, it is necessary to distinguish between as many cases as there are transition functions. And in it, we need to consider the case where any process p, q is equal to r, or one of them is different from the other, or each of them is different from the other. The proof scores of the mutual exclusion property of TAS protocol are described below.

```
open TAS .
   ops p q : -> Pid .
   red mutex(init,p,q) .
close

open TAS .
   op s : -> Sys .
   ops p q r : -> Pid .
   -- eq c-enter(s,r) = true .
```

```
   eq pc(s,r) = rs .
   eq locked(s) = false .
   eq p = r .
   eq q = r .
   red mutex(s,p,q) implies mutex(enter(s,r),p,q) .
close

open TAS .
   op s : -> Sys .
   ops p q r : -> Pid .
   -- eq c-enter(s,r) = true .
   eq pc(s,r) = rs .
   eq locked(s) = false .
   eq p = r .
   eq (q = r) = false .
   red inv1(s,q) implies mutex(s,p,q) implies mutex(enter(s,r),p,q)
.close
```

(The following is omitted)

Proof scoring mainly uses induction to describe the proof score. In induction, it is necessary to write the proofs for the base case and the induction case respectively. The base case in the verification of mutual exclusivity of mutual exclusion protocols is the initial state. Induction case means the other case. Induction case proofs need to describe the proof score for all cases. The details will be discussed later in the section on formal verification of the Anderson protocol. Here, `TAS` is the name of the module in which the formal specification of the protocol is described, and `mutex` is a state predicate for mutual exclusion property, as follows

```
   eq mutex(S,P,Q) = (pc(S,P) = cs and pc(S,Q) = cs implies (P =
Q)) .
```

A collection of `open` to `close` is called a fragment. Evaluate the expression with `red` and if it returns `true`, it means that the proof of the fragment was successful.

3.3 Formal verification of mutual exclusion property of TAS protocol using CiMPA and CiMPG

I conducted formal verification by CiMPA and CiMPG. The proof script generated using CiMPG is shown below.

```
:goal
eq [tas1 :nonexec] : inv1(S:Sys, P:Pid, P0:Pid) = true .
eq [tas :nonexec] : inv2(S:Sys, P:Pid) = true .

:ind on (S:Sys)

   :apply(si)

:sel(2)
   :apply(tc)

   :apply (rd)

   :apply (rd)

:sel(3)
   :apply(tc)

   :def csb1#3 = :ctf eq pc(S#Sys, P#Pid) = cs .

   :apply(csb1#3)

   :def csb2#3 = :ctf eq P@Pid = P#Pid .

   :apply(csb2#3)

   :apply (rd)

   :def csb3#3 = :ctf eq P0@Pid = P#Pid .

   :apply(csb3#3)
```

```
:apply (rd)

:imp [tas1] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

:apply (rd)

:imp [tas1] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

:apply (rd)

:def csb4#3 = :ctf eq pc(S#Sys, P#Pid) = cs .

:apply(csb4#3)

:def csb5#3 = :ctf eq P@Pid = P#Pid .

:apply(csb5#3)

:apply (rd)

:def csb6#3 = :ctf eq pc(S#Sys, P@Pid) = cs .

:apply(csb6#3)

:imp [tas1] by P0:Pid <- P#Pid ; P:Pid <- P@Pid ;

:apply (rd)

:apply (rd)

:imp [tas] by P:Pid <- P@Pid ;

:apply (rd)
```

(The following is omitted)

In the formal verification of TAS protocol, two goals were generated because of the need to prove mutual exclusion property and another lemma. Also, TAS protocol has three constructors, `init`, `enter`, and `exit`, so the proof script is divided into three parts. They do not necessarily generate proof scripts in the order described in the proof score.

By inputting the above proof script into CiMPA, we have completed the formal verification of the TAS protocol.

Chapter 4

Qlock protocol

In this chapter, to prove the mutual exclusion property of Qlock protocol, we describe the specification of the protocol and the operations of the queue, after that, verify it by proof scoring and automatic proof using CiMPG and CiMPA.

4.1 Formal specification of Qlock protocol

Qlock protocol is a mutual exclusion protocol that uses a queue, a kind of data structure, and unlike TAS protocol, it has three sections: a reminder section, a critical section, and waiting section. Unlike TAS protocol, there are three sections: the Remainder Section, the Critical Section, and the Waiting Section. Any selected process located in the Remainder Section first transits to the Waiting Section, where its ID is inserted into the queue.
Qlock protocol written in Algol-like pseudo-code is as follows:

loop {
"Remainder Section"
rs: *enq(queue,i)*;
"Waiting Section"
ws: **repeat until** *top(queue) = i;*
"Critical Section"
cs:*deq(queue)*;
}

An illustration of the system is shown below.

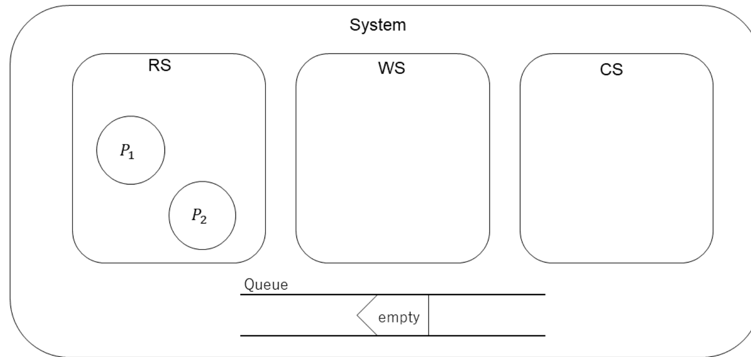

Figure 4.1: Initial state of Qlock protocol

Where *i* is the identifier of an arbitrary process, *enq* is an operation to insert an identifier at the end of the queue, *top* is an operation to refer to the first identifier of the queue, and *deq* is an operation to retrieve the first identifier of the queue. When a process wants to use a shared resource, the identifier of the process is added to the end of the queue, and the process transition from RS to WS(Waiting Section). When it comes to the top of the queue, it goes into CS. When the process finishes using the shared resources, remove it from the queue and return from CS to RS.

The operations of the queue are described below.

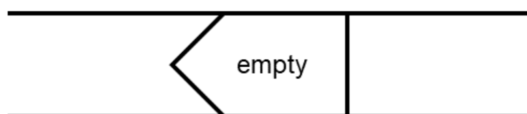

Figure 4.2: Initial state of queue

Initially, an empty queue is created.

- **enq**(empty,X) = X empty .

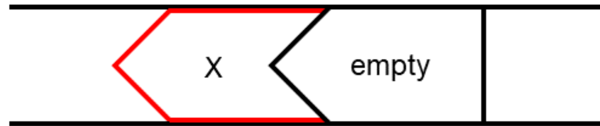

Figure 4.3: Operation of *enq*

enq is a function to insert an element (process ID) into a queue. If you call *enq* with the first argument being empty and the second argument being the ID of an arbitrary process, you will see the above figure. In the figure, the new elements inserted are shown in red. Adding an element to an empty queue will add it to the front of the queue.

- **enq**(X Q,Y) = X enq(Q,Y) .

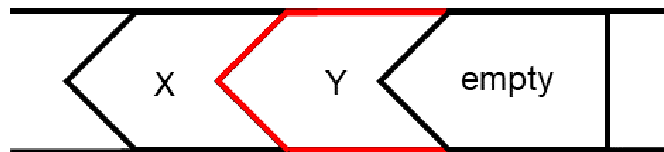

Figure 4.4: Operation of *enq* - 2

Calling *enq* with the first argument being a non-empty queue, that is, a queue with one or more elements, and the second argument is the ID of an arbitrary process, will result in the above figure. In the figure, the new elements inserted are shown in red. If you add an element to a queue that has one or more elements, it will be added before the empty one.

- **deq**(X Q) = Q .

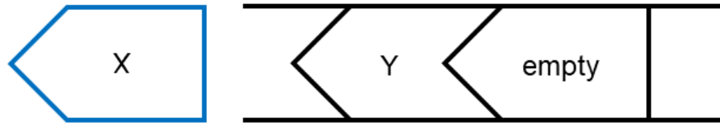

Figure 4.5: Operation of *deq*

The queue needs not only operations to insert new elements, but also operations to delete them. *deq* is a function for deleting the first element of a queue. When *deq* is called, the top element of the queue is taken out.

- **top**(Y Q) = Y .

Figure 4.6: Operation of top

When *top* is called, refer to the element at the top of the queue. In the above figure, the first element is Y, so the value of Y is returned.

- X **/in** (Y Q) = (if X = Y then true else X /in Q fi) .

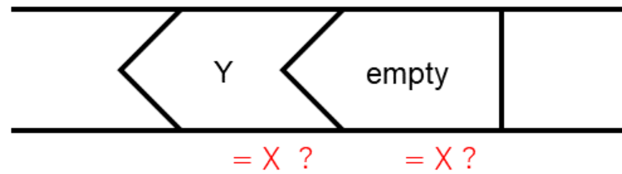

Figure 4.7: Operation of $\backslash in$

\in is the operator to search for an element in order from the top of the queue. Returns true if X is found, false otherwise.

The state transitions of Qlock protocol are described below.

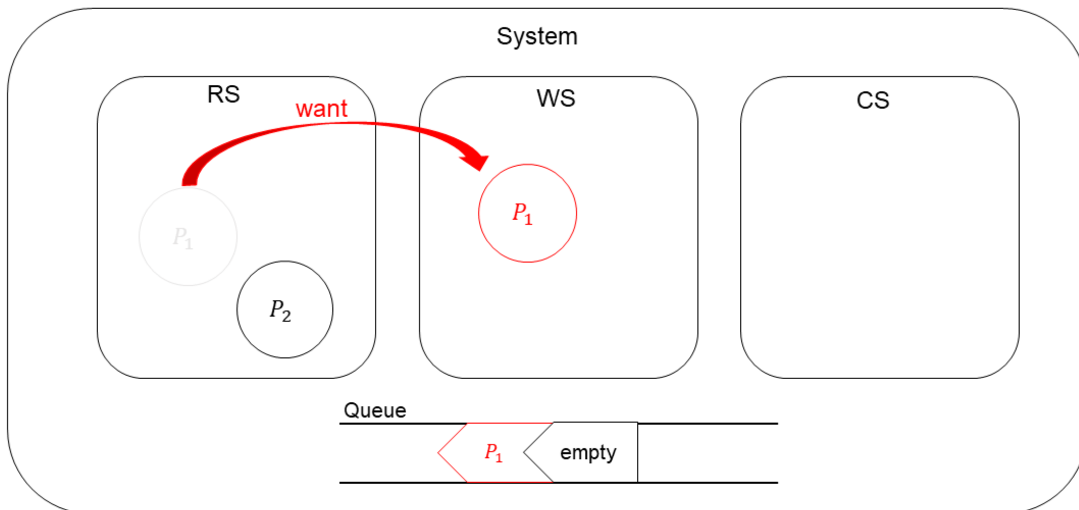

Figure 4.8: Transion from rs to ws of Qlock protocol

When the process transitions from RS to WS, add the process to the queue.

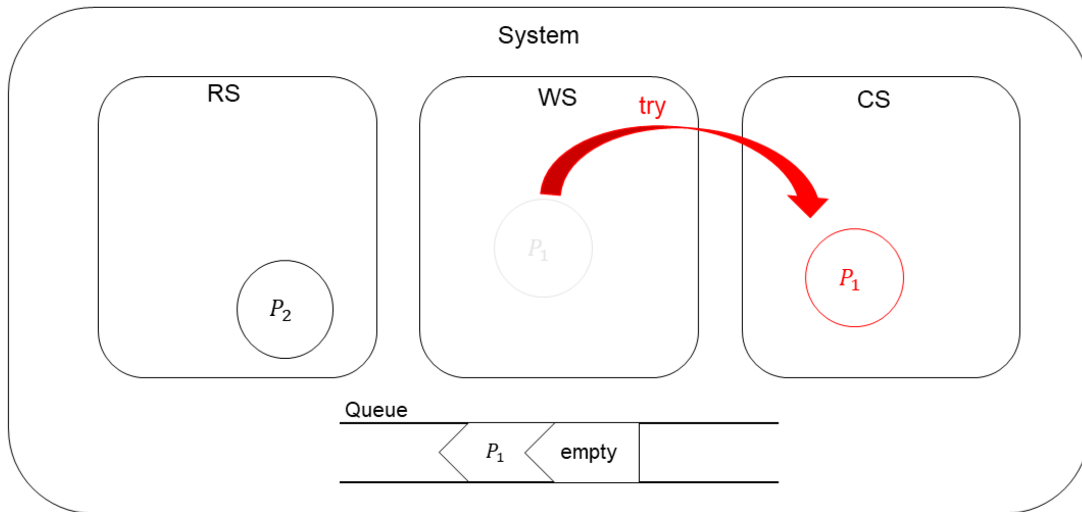

Figure 4.9: Transion from ws to cs of Qlock protocol

If the process in WS and the process at the top of the queue are the same, the process transitions from WS to CS.

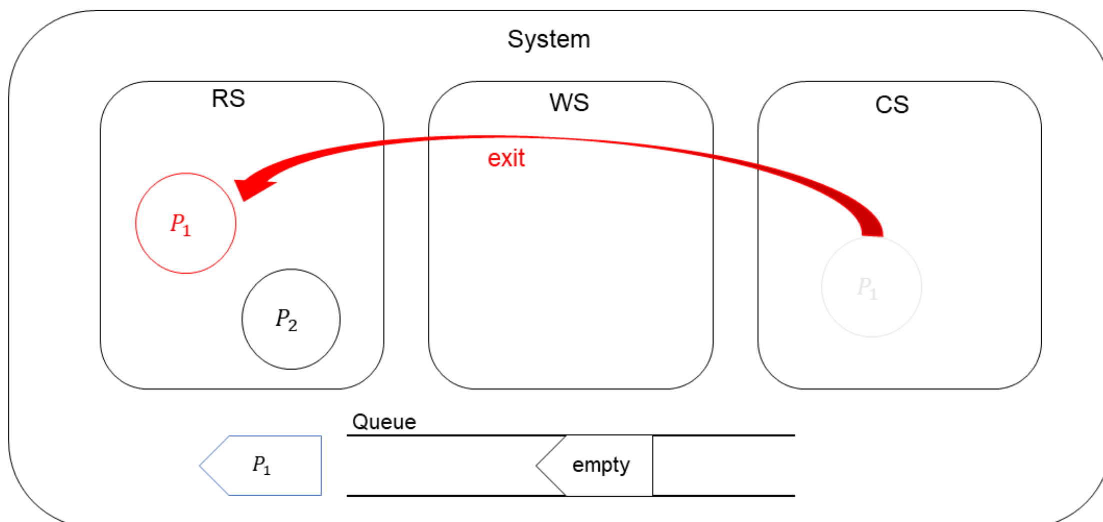

Figure 4.10: Transion from cs to rs of Qlock protocol

When the process transitions from CS to RS, remove the process at the top of the queue.

4.2 Formal verification of Qlock protocol by proof scoring

In verification of mutual exclusion property, it is necessary to distinguish between as many cases as there are transition functions. And in it, we need to consider the case where any process `p`, `q` is equal to `r`, or one of them is different from the other, or each of them is different from the other. The part of The proof scores of the mutual exclusion property of Qlock protocol are described below.

```
-- I) Base case

open QLOCK .
   -- fresh constants
   ops i j : -> Pid .
   -- |-
   red inv1(init,i,j) .
close

   --

-- II) Induction cases

-- 1) want(s,k)

open QLOCK .
   -- fresh constants
   op s : -> Sys .
   ops i j k : -> Pid .
-- IH
   eq [:nonexec] : inv1(s,I:Pid,J:Pid) = true .
-- assumptions
   eq pc(s,k) = rs .
   eq i = k .
-- |-
   red inv1(s,i,j) implies inv1(want(s,k),i,j) .
close
```

(The following is omitted)

By entering annotated proof scores into CiMPG, the following proof scripts are generated

```
open QLOCK .
   :goal
   eq [inv1 :nonexec] : inv2(S:Sys,P:Pid) = true .
   eq [inv11 :nonexec] : inv1(S:Sys,P:Pid,P0:Pid) = true .

   :ind on (S:Sys)

   :apply(si)

-- for exit

   :apply(tc)

   :def csb1a = :ctf eq pc(S#Sys,P#Pid) = cs .

   :apply(csb1a)

   :def csb2a = :ctf eq P@Pid = P#Pid .

   :apply(csb2a)

   :imp [inv11] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

   :apply (rd)

   :def csb3a = :ctf eq P0@Pid = P#Pid .

   :apply(csb3a)

   :imp [inv11] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

   :apply (rd)

   :imp [inv11] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;
```

```
    :apply (rd)

    :imp [inv11] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

    :apply (rd)

    :def csb4a = :ctf eq pc(S#Sys,P#Pid) = cs .

    :apply(csb4a)

    :def csb5a = :ctf eq P@Pid = P#Pid .

    :apply(csb5a)

    :imp [inv1] by P:Pid <- P@Pid ;

    :apply (rd)

    :def csb6a = :ctf eq pc(S#Sys,P@Pid) = cs .

    :apply(csb6a)

    :imp [inv1] by P:Pid <- P@Pid ;

    :imp [inv11] by P0:Pid <- P#Pid ; P:Pid <- P@Pid ;

    :apply (rd)

    :imp [inv1] by P:Pid <- P@Pid ;

    :apply (rd)

    :imp [inv1] by P:Pid <- P@Pid ;

    :apply (rd)

-- end of proof for exit()
```

(The following is omitted)

In the formal verification of Qlock protocol, two goals were generated because of the need to prove mutual exclusion property and another lemma.

Also, Qlock protocol has four constructors, `init, want, try`, and V `exit`, so the proof script is divided into four parts. They do not necessarily generate proof scripts in the order described in the proof score. Induction case proofs need to describe the proof score for all cases. The details will be discussed later in the section on formal verification of the Anderson protocol.

By inputting the above proof script into CiMPA, we have completed the formal verification of the Qlock protocol.

Chapter 5

Anderson protocol

In this chapter, to prove mutual exclusion property of Anderson protocol[7], we describe the specification of the protocol and verify it using the proof scoring. We also proposed a method to prove the property which was difficult to prove and completed the proof. This chapter describes the specification and formal verification of Anderson. Finally, we were tackled automatic proof using CiMPG and CiMPA.

5.1 Formal specification of Anderson

Anderson protocol written in Algol-like pseudo-code is as follows:

loop{
"Remainder Section"
rs: *place[i]* := fetch&incmode(*next*,N);
"Waiting Section"
ws: **repeat until** *array*[*place*[*i*]] ;
"Critical Section"
cs: *array*[*place*[*i*]] := `false`;
array[(*place*[*i*] + 1) % N] := `true`;
}

Anderson has three sections like Qlock protocol. And it has two arrays and two variables. Two arrays are *place* and *array*. *place* is the waiting place of the process that transits to CS. When a process transitions from RS to WS, a waiting place is allocated to the process. The waiting area to be allocated depends on the value of *next*. *array* is the array used to allocate the right to transition to CS to processes. Therefore, the value of *array* is

always `true` for only one of them and `false` for all others. Two variables are *next* and *count*. *next* is used to allocate a waiting place to a process in order as described above. The minimum value is 0 and the maximum value is the number of all processes. When the process transitions from RS to WS, the value increases by 1. When the maximum value is reached, it returns to 0. *count* is used to count the number of processes in WS and CS. When the process transitions from RS to WS, the value increases by 1. When the process transitions from CS to RS, the value decreases by 1.

The following diagram shows the initial state.

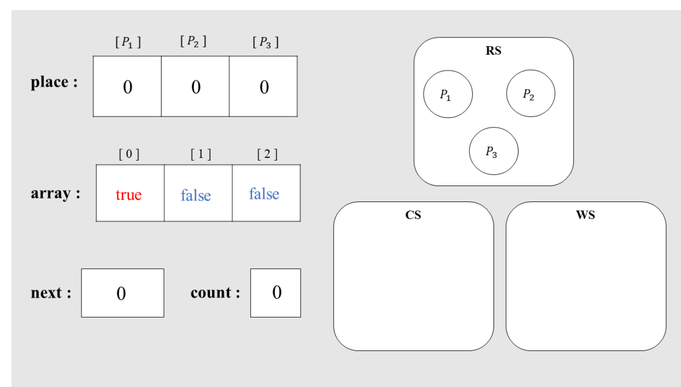

Figure 5.1: Initial state of Anderson protocol

Consider the case where P_2 transitions from RS to WS.

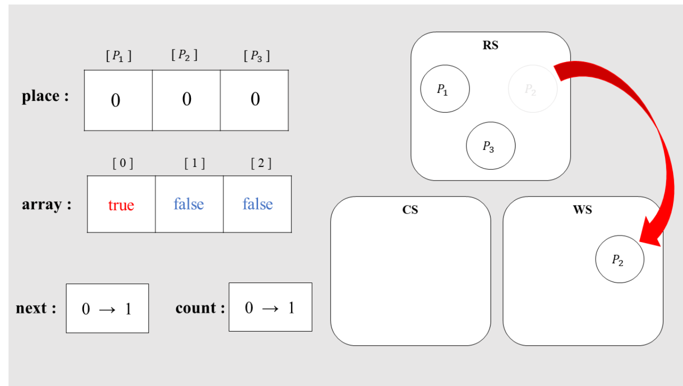

Figure 5.2: Transition from rs to ws of Anderson protocol

At this time, the value of *next* is 0, so the value of *place* corresponding to P_2 is also rewritten to 0. (However, since the initial value of each *place* is 0, it appears to be unchanged.) Then, increase the values of *next* and *count* by 1.

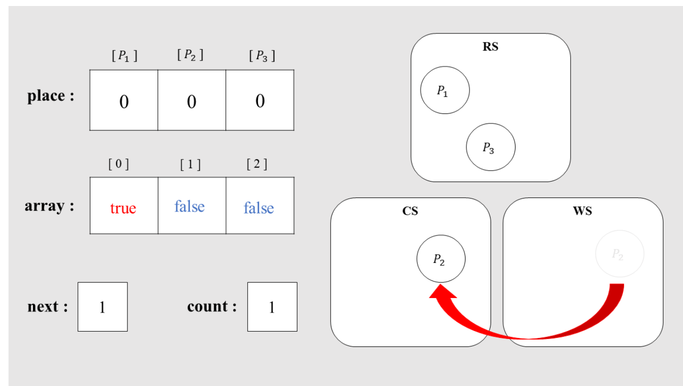

Figure 5.3: Transition from ws to cs of Anderson protocol

Then, consider the case where P_2 transitions from WS to CS. In this time, Since the value of *place* corresponding to P_2 is 0 and the value of *array* corresponding to 0 is *true*, P_2 can transition to CS. each value remains unchanged.

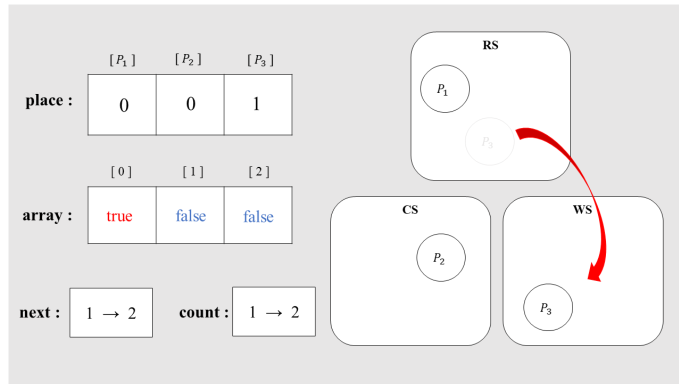

Figure 5.4: Transition from rs to ws of Anderson protocol - 2

Consider the case where P_3 transitions from RS to WS. In this time, The value of *place* corresponding to P_3 is rewritten to the value of *next*. And both the *next* and *count* values are increased by 1.

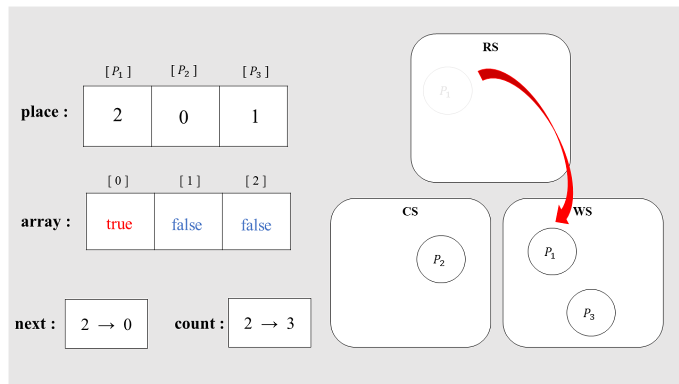

Figure 5.5: Transition from rs to ws of Anderson protocol - 3

Consider the case where P1 transitions from RS to WS. In this time, The value of *place* corresponding to P_1 is rewritten to the value of *next*. the value *count* is increased by 1. The value of *next* is the maximum value, so it returns to 0. Everything else remains the same.

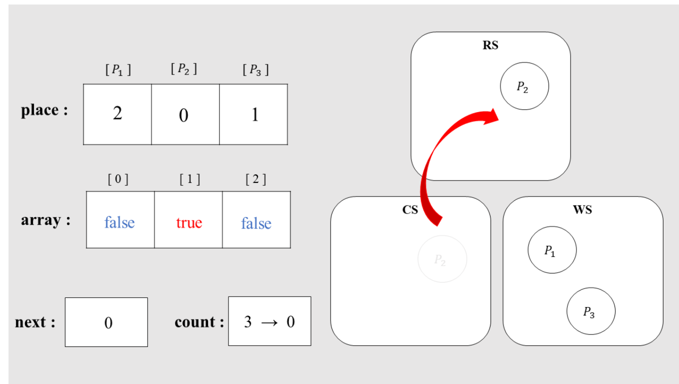

Figure 5.6: Transition from cs to rs of Anderson protocol

Consider the case where P_2 transitions from CS to RS.

5.2 Formal verification of Anderson by proof scoring

In verification of the mutual exclusion property, the reachable states need to be exhaustively verified, but in some cases, they do not need to be verified. In Anderson, there are four sub-goals, (1)`init`, (2)`want`, (3)`try`, and (4)`exit`. The following figure shows the case splitting of (2)`want`.

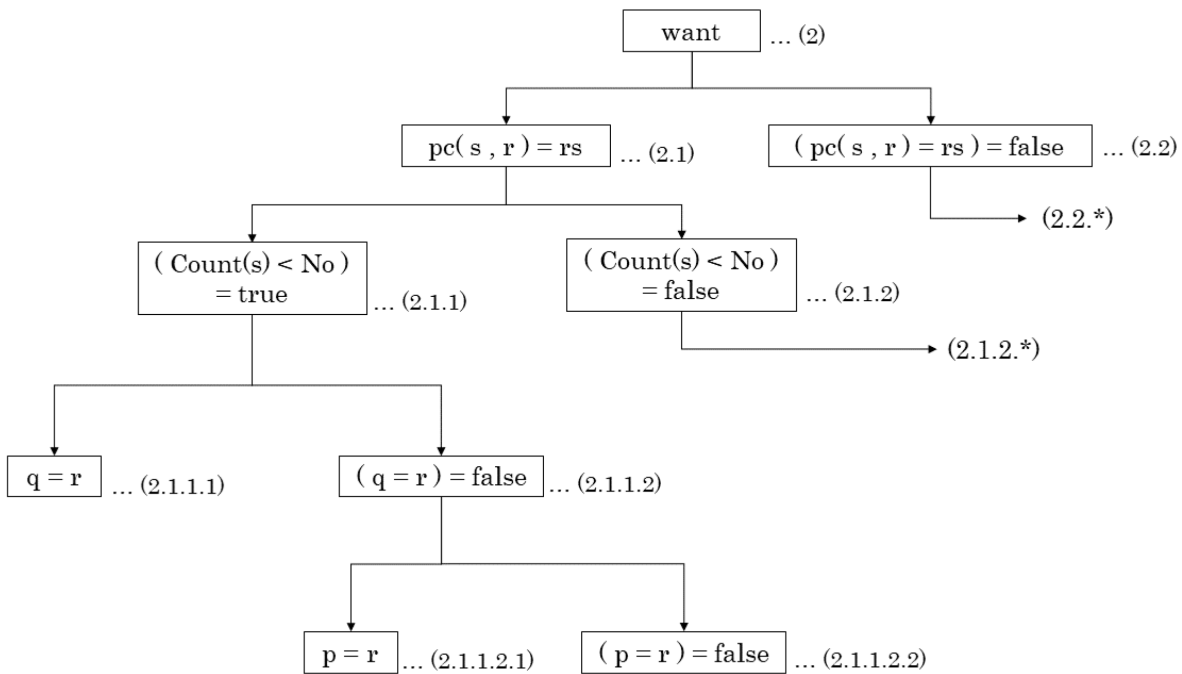

Figure 5.7: Example of case splitting

In case(2.2.*), if the process is not located in RS, the state does not transition, so there is no need to prove it further. In case(2.1.2.*), since the value of *count* can never be greater than the total number of processes, no further proof is needed.

The part of the proof scores of the mutual exclusion property of Anderson protocol are described below. where `INV` is the name of the module that describes the specification of the invariants of the Anderson protocol.

```
--> init
open INV .
   ops i, j : -> Pid .
   red mutex(init, i, j) .
close

--> exit
open INV .
   op s : -> Sys .
   ops p q r : -> Pid .
   eq [:nonexec] : mutex(s, P:Pid, Q:Pid) = true .
   eq pc(s, r) = cs .
   eq q = r .
   red mutex(s, p, q) implies mutex(exit(s, r), p, q) .
close
```

(The following is omitted)

5.3 Formal verification by introducing an auxiliary variable

`inv7` cannot complete the proof as it is[8]. We also need `inv7-2` to prove `inv7`.

```
eq inv7-2(S,P,Q) = ((pc(S,P) = ws or pc(S,P) = cs) and (pc(S,Q) =
ws or pc(S,Q) = cs) and (P = Q) = false) implies (s(0) < count(S))
.
```

`inv7-2` states that if there exists 2 different processes `P` and Q located at ws or cs, then `count` is greater than 1 (which is `s(0)`). Again, the proof of `inv7-2` needs to use another lemma `inv7-3`, which states that if there exist 3 different processes located at ws or cs, then count is greater than 2.

```
eq inv7-3(S,P,Q,R) = ((pc(S,P) = ws or pc(S,P) = cs) and (pc(S,Q)
= ws or pc(S,Q) = cs) and (pc(S,R) = ws or pc(S,R) = cs) and (P =
Q) = false and (P = R) = false and (Q = R) = false) implies (s(s(0))
< count(S)) .
```

And more, we have to define an infinite number of lemma for 3 processes, 4 processes, and so on. The problem comes from we cannot observe the full

numbers processes located at ws or cs. In each `inv7, inv7-2. inv7-3,` etc., we only know that there exists one or 2 or 3 number of processes located at ws or cs. We could also generalize the lemma, but it is impossible to deal with such an operator with a variable number of parameters in CafeOBJ.

```
   eq inv7-k( S,P1,...,Pk ) = ( (pc(S,P1) = ws or pc(S,P1) = cs)
and ...  and ( pc( S,Pk) = ws or pc(S,Pk) = cs ) and ( P1 = P2 )
= false and ...  and ( P(k-1) = Pk) = false ) implies ((s)*(k-1)
(0) < count(S) ) .
```

To complete the proof, we need to add a new observers. [9]They are shown below.

```
op # : Set -> SNat .
op \in_ : Elt Set -> Bool .
op _-_ : Set Set -> Set .
op psInWsCs : Sys -> Set .
```

Given a set `c`, `#(c)` is the number of elements in `c`. Given a set `c1, c2`, `c1 - c2` is the set obtained by deleting each element in `c2` from *c1*. `\in` is a membership predicate of sets. `psInWsCs(s)` is the set of processes that are in WS and CS.

Any process `P` is in WS or CS means that `P` is in the set `psInWsCs`, so the number of elements of `psInWsCs` is at least 1. Then, the value of `count` is at least 1. So, it is clear that the lemma is correct.

For the proof of `inv7` using the proof score, we need to consider each of the following cases:

- `pc(p) = cs`
  `pc(p) = ws`
  `pc(p) = rs`

- `p \in psInWsCs(s)`
  `p not \in psInWsCs(s)`

- `(0 < #(psInWsCs(s))) = true`
  `(0 < #(psInWsCs(s))) = false`

- #(psInWsCs(s)) = count(s)
  #(psInWsCs(s)) not = count(s)

```
open INV .
   op p : -> Pid .
   op s : -> Sys .
   eq pc(s,p) = cs .
   eq (p \in psInWsCs(s)) = true .
   eq (0 < #(psInWsCs(s))) = true .
   eq count(s) = #(psInWsCs(s)) .
   red inv7(s,p) .
close

open INV .
   op p : -> Pid .
   op s : -> Sys .
   eq pc(s,p) = cs .
   eq (p in psInWsCs(s)) = true .
   eq (0 < #(psInWsCs(s))) = false .
   red inv-s2(psInWsCs(s), p) implies inv7(s,p) .
close

open INV .
   op p : -> Pid .
   op s : -> Sys .
   eq pc(s,p) = cs .
   eq (p \in psInWsCs(s)) = false .
   red inv9(s,p) implies inv7(s,p) .
close

open INV .
   ops p r : -> Pid .
   op s : -> Sys .
   eq pc(s,p) = cs .
   eq (#(psInWsCs(s)) = count(s)) = false .
   red inv8(s) implies inv7(s,p) .
close

open INV .
   op p : -> Pid .
```

```
   op s : -> Sys .
   eq pc(s,p) = ws .
   eq (p \in psInWsCs(s)) = true .
   eq (0 < #(psInWsCs(s))) = true .
   eq count(s) = #(psInWsCs(s)) .
   red inv7(s,p) .
close

open INV .
   op p : -> Pid .
   op s : -> Sys .
   eq pc(s,p) = ws .
   eq (p \in psInWsCs(s)) = true .
   eq (0 < #(psInWsCs(s))) = false .
   red inv-s2(psInWsCs(s), p) implies inv7(s,p) .
close

open INV .
   op p : -> Pid .
   op s : -> Sys .
   eq pc(s,p) = ws .
   eq (p \in psInWsCs(s)) = false .
   red inv9(s,p) implies inv7(s,p) .
close

open INV .
   ops p r : -> Pid .
   op s : -> Sys .
   eq pc(s,p) = ws .
   eq (#(psInWsCs(s)) = count(s)) = false .
   red inv8(s) implies inv7(s,p) .
close

open INV .
   op p : -> Pid .
   op s : -> Sys .
   eq (pc(s,p) = ws) = false .
   eq (pc(s,p) = cs) = false .
   red inv7(s,p) .
close
```

The following cases are omitted because it is clear from the supplementary title that they are not valid.

- p not \in psInWsCs(s)
  By inv9: inv9(S,P) = (pc(S,P) = ws or pc(S,P) = cs) implies P \in psInWsCs(S) .
- (0 < #(psInWsCs(s))) = false
  By inv-s2: inv-s2(S,E) = E \in S implies 0 < #(S) .
- #(psInWsCs(s)) not = count(s)
  By inv8: inv8(S) = #(psInWsCs(S)) = count(S) .

5.4 Formal verification of mutual exclusion property of Anderson using CiMPA and CiMPG

We conducted formal verification of Anderson using CiMPG and CiMPA. To use CiMPG, it was necessary to write a proof score using induction. As is usually the case, It is easy to write proof scores by induction from proof scores written by case-splitting, but `inv7` is not easy, and a new lemma `inv14` had to be defined. And `inv7` written by induction are shown below.

```
open INV .
  :id(inv7)
  op p : -> Pid .
  red inv7(init, p) .
close

open INV .
  :id(inv7)
  ops p r : -> Pid .
  op s : -> Sys .
  eq pc(s,r) = cs .
  eq p = r .
  red inv7(s,p) implies inv7(exit(s,r),p) .
close
```

```
open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
    eq pc(s,p) = cs .
    red mutex(s,r,p) implies inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
    eq pc(s,p) = rs .
    red inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
    eq pc(s,p) = ws .
    eq r \in psInWsCs(s) = true .
    eq p \in psInWsCs(s) = true .
    eq count(s) = #(psInWsCs(s)) .
    eq #(psInWsCs(s)) = s(#(psInWsCs(s) - r)) .
    eq #(psInWsCs(s) - r) = s(#((psInWsCs(s) - r) - p)) .
    red inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
```

```
    eq pc(s,p) = ws .
    eq r \in psInWsCs(s) = true .
    eq p \in psInWsCs(s) = true .
    eq count(s) = #(psInWsCs(s)) .
    eq #(psInWsCs(s)) = s(#(psInWsCs(s) - r)) .
    eq ( #(psInWsCs(s) - r) = s(#((psInWsCs(s) - r) - p)) ) = false .
    red inv14(psInWsCs(s) - r,p) implies inv7(s,p) implies inv7(exit(s,r),p) .
close
open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
    eq pc(s,p) = ws .
    eq r \in psInWsCs(s) = true .
    eq p \in psInWsCs(s) = true .
    eq count(s) = #(psInWsCs(s)) .
    eq ( #(psInWsCs(s)) = s(#(psInWsCs(s) - r)) ) = false .
    red inv14(psInWsCs(s),r) implies inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
    eq pc(s,p) = ws .
    eq r \in psInWsCs(s) = true .
    eq p \in psInWsCs(s) = true .
    eq (count(s) = #(psInWsCs(s))) = false .
    red inv10(s) implies inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
```

```
    eq pc(s,p) = ws .
    eq r \in psInWsCs(s) = true .
    eq p \in psInWsCs(s) = false .
    red inv11(s,p) implies inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = cs .
    eq (p = r) = false .
    eq pc(s,p) = ws .
    eq r \in psInWsCs(s) = false .
    red inv11(s,r) implies inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq (pc(s,r) = cs) = false .
    red inv7(s,p) implies inv7(exit(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = ws .
    eq array(s,place(s,r)) = true .
    eq p = r .
    red inv7(s,p) implies inv7(try(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = ws .
    eq array(s,place(s,r)) = true .
```

```
    eq (p = r) = false .
    red inv7(s,p) implies inv7(try(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = ws .
    eq array(s,place(s,r)) = false .
    red inv7(s,p) implies inv7(try(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq (pc(s,r) = ws) = false .
    red inv7(s,p) implies inv7(try(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = rs .
    eq count(s) < No = true .
    red inv7(s,p) implies inv7(want(s,r),p) .
close

open INV .
    :id(inv7)
    ops p r : -> Pid .
    op s : -> Sys .
    eq pc(s,r) = rs .
    eq count(s) < No = false .
    red inv7(s,p) implies inv7(want(s,r),p) .
close

open INV .
    :id(inv7)
```

```
    ops p r : -> Pid .
    op s : -> Sys .
    eq (pc(s,r) = rs) = false .
    red inv7(s,p) implies inv7(want(s,r),p) .
close
```

If you compare the above proof score with the previous proof score from case splitting, you can see the difference between the two approaches.

The proof score for each invariant was entered into CiMPG to generate a proof script. As an example, a part of the proof script for `mutex` is shown below.

```
:proven(inv1(S:Sys, P:Pid, Q:Pid))
:goal{
eq [mutex :nonexec] : mutex(S:Sys, P:Pid, P0:Pid) = true .
}

    :ind on (S:Sys)

    :apply(si)

    :sel(2)
    :apply(tc)

    :apply (rd)

:sel(1)
    :apply(tc)

    :def csb1#1 = :ctf eq pc(S#Sys, P#Pid) = cs .

    :apply(csb1#1)

    :def csb2#1 = :ctf eq P0@Pid = P#Pid .

    :apply(csb2#1)

    :imp [mutex] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;
```

:apply (rd)

:def csb3#1 = :ctf eq P@Pid = P#Pid .

:apply(csb3#1)

:imp [mutex] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

:apply (rd)

:imp [mutex] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

:apply (rd)

:imp [mutex] by P0:Pid <- P0@Pid ; P:Pid <- P@Pid ;

:apply (rd)

(The following is omitted)

The structure of the proof script for the Anderson protocol does not differ from that of the QlocK protocol, but because the Anderson protocol has many invariants, a proof script was generated for each invariant, unlike the other two protocols. For this reason, the first line of the proof script generates a prove command. (This means that `inv1` is used as a lemma for `mutex` proofs.)

By inputting the above proof script into CiMPA, we have completed the formal verification of the Anderson.

Chapter 6

Lessons Learned

In this chapter, I will describe what I have learned through this study.

6.1 Functional programming

- CafeOBJ used in this study is a functional programming language. Unlike general programming languages, functional programming languages cannot use control structures such as ”for”.Therefore, it is necessary to implement a recursive function to perform the iterative process. Implementing recursive iterative processing requires more thought and ingenuity than iterative processing using control structures.
- CafeOBJ treats equations as rewrite rules. Therefore, you need to be a little careful when writing expressions. For example, there are equations such as `A=B` and `(A=B) = true`. In general, these two are interpreted to mean that they are equal. However, in CafeOBJ, the former means that `A` will be rewritten as `B`, and the latter means that `A=B` will be rewritten as true, so do not confuse the two.
- functional programming languages have referential transparency. Referential transparency is the property that if the same argument is given to a function, the return value will always be the same. In other words, it is a guarantee that the function does not have any state. Therefore, most of the code in CafeOBJ is written in the form of equations. So, in functional programming, the expression ”evaluate an expression” is often used instead of ”call a function”.
- Unlike procedural programming languages, programming with the above restrictions was difficult, but it was interesting to write the code with ingenuity.

- Learning for functional programming is often done by constructing small, simple programming languages. In the course on functional programming that I took to work on this research, I also used CafeOBJ to create a programming language called ”Minila” (strictly speaking, its processor, compiler, and virtual machine). We usually use a variety of programming languages to create a variety of programs. So, it is not that difficult to answer the questions of what is a programming language, what is a processor, what is a compiler, and what is a virtual machine. However, a programming language is also a program. Not many people can answer yes to the question, ”Can you create a programming language, or have you ever created one? To create a program needs programming knowledge and skills, and that a deeper understanding of how programming languages work will enable you to create better programs.

6.2 Date structure

In doing functional programming, of course, programming knowledge and skills are required, but there were many situations where knowledge of data structures was necessary. Typical data structures include queues and stacks. In applying formal methods to mutual exclusion protocols, knowledge of sets is essential; the Qlock protocol, as the name suggests, uses queues for mutual exclusion. Understanding data structures is not difficult, but implementing data structures in functional programming is not always easy. In the formal verification of the Anderson protocol, the data structure used in the protocol specification made the proof using CiMPA and CiMPG extremely difficult, and the data structure had to be changed. In implementing various programs, the only way to find out which data structure is appropriate to use is through trial and error.

6.3 Formal method

I was wondering why formal methods were not widely used in the software development field, even though they are considered to be a promising technology to assure the quality of software. However, I realized the difficulty of formal methods when I worked on formal verification.

- I used CafeOBJ in the formal specification because it is a language developed in my graduate school. But CafeOBJ is not the only formal

specification language (or tool) used for formalisms. Some of the more famous ones are Z and VDM. They are not the same thing, and you have to use the one that is appropriate for your problem. Especially for new cases, it is rare to get the right approach from the beginning, so trial and error is necessary.

- Formal methods are not a panacea, and not all systems can be modeled. As a matter of course, the larger the system in issue, the more difficult it is to model and the more difficult it is to verify. The target of the formal verification I worked on in this research is the mutual exclusion protocol. It is a piece of software in an information system. It is difficult for me to think how difficult it would be to target the whole software. There is no confirmed case to date where formal verification of a large-scale information system has been completed.

6.4 Proof scoring

Proof scoring is a method of proving that a required property described as a state predicate from a formal specification is satisfied. That was sufficient for the TAS and QloCK case studies, but for the Anderson protocol, the formal specification alone was not sufficient to complete the proof. An auxiliary variable was needed to complete the proof. An auxiliary variable is just a variable that changes the type that the originally defined variable holds, and it does not affect the behavior of the protocol. Therefore, it was easy to understand the role of auxiliary variables. However, it was very difficult for me to solve questions such as why we need auxiliary variables, when we need auxiliary variables, and whether it is impossible to complete a proof without auxiliary variables.

6.5 Visualization of the system

Conjecturing the lemma is one of the most difficult tasks in theorem proving. One of the effective technologies for this purpose is SMGA, which is a graphical animation of state machines. For formal verification of mutual exclusion protocols, it is essential to understand the behavior of the protocol, but it is not easy to intuitively understand the behavior of the protocol from its formal specification or pseudocode. I have never found any useful lemma in SMGA, but being able to see the state transitions of the processes in the protocol was very helpful in understanding the protocol specifications.

6.6 Difference between manual and automatic

- I approached formal verification in two ways. One is proof scoring, The other is using CiMPG and CiMPA.Proof scoring is described as ”flexible” compared to automatic methods. I was faced with a situation where the proof scores were correct but could not be proven in CiMPA and CiMPG in formal verification of Anderson. This was due to the data structure used in Anderson’s specification, so it was necessary to change the specification. This helped me to understand the meaning of ”flexible”. Theorem proving is considered to be more difficult to automate than model checking. The reason is that it is difficult to find and prove a lemma. In addition to that, I found that the flexibility of each method mentioned above was one of the reasons.

- CiMPA can automatically perform proofs but is less flexible than pull scoring, pull scoring is more flexible than CiMPA but can be mixed with human error. Handwritten proof with a pen is less practical but is useful for developing the ability to conjecture lemmas. There are three methods based on theorem proving: proof scoring, using CiMPG and CiMPA, and handwriting with a pen. I found that automatic methods are not necessarily better than other methods and that each method has its strengths and weaknesses.

- One problem that formal methods have is ”the inspection mechanism of the inspection mechanism”. The problem is that if there is a mechanism for inspecting the quality of something, the quality of that inspection mechanism is not certain, so there may be a need for an additional inspection mechanism to verify it. Even in formal verification by theorem proving, human error may occur in the proof by hand. The combined method of CiMPG and CIMPA, which overcomes this shortcoming, is revolutionary.

Chapter 7

Conclusion

This chapter summarizes the contents of the previous chapters and discusses future work.

7.1 Summary

This report aims to address the verification of the mutual exclusion property of several mutual exclusion protocols and describes the following.

- Chapter 1 : Introduction
  This chapter described recent trends in information systems (especially software) and formal methods. Today, many things are controlled by information systems. Information systems can be broadly divided into three elements: hardware, software, and networks. In recent years, the role of software has become particularly important, and the quality required of software has also become higher. One of the techniques to guarantee the quality of software systems is formal methods. Formal methods are considered to be one of the most promising techniques to guarantee the quality of software, but they have not yet penetrated the software development field. One of the possible reasons for this is that there are few examples of formal methods.

- Chapter 2: Preliminaries
  It is not possible to write a description of the formal methods used in carrying out this research. Formal methods can be divided into two main categories. They are formal specification (description) and formal verification. Formal specification is to describe the target system or problem in a formal specification language, and formal verification is to verify whether the target system or problem satisfies the properties

to be verified based on the formal specification. Formal verification is further divided into model checking and theorem proving. Theorem proving can be further divided into several methods, but in this report, we mainly focus on proof by pull-scoring and automatic proof by combining CiMPG and CIMPA.

In this study, we used the language CafeOBJ for both formal specification and formal verification, which is a formal specification language classified as an algebraic specification language designed and developed to support formal methods. To describe a formal specification, the subject must be accurately understood and modeled. There are several frameworks for modeling, such as process algebra and state-transition systems, and there is no general-purpose approach. In this study, we consider a collection of observable typed values that characterize a snapshot of protocol execution as a state of the protocol, and model the behavior of the protocol as a state transition system, an approach called observation transition system.

- Chapter 3 : Test & Set protocol
  This chapter explained the specification and the proof score of the protocol, also automatic proof using CiMPA and CiMAG.

  TAS protocol is the simplest of the mutual exclusion protocols; it has two sections, a reminder section, and a critical section, and uses a boolean variable called ”looked” to perform mutual exclusion. Specifically, when any process is selected, if the value of ”looked” is ”false”, the process is moved to the critical section, and at the same time, the value of ”looked” is set to ”true If the value of ”looked” is ”false,” the process will be moved to the critical section and the value of ”looked” will be rewritten to ”true,” and the value of ”looked” will be rewritten to ”false” when the process leaves the critical section. Other processes will not be able to transition to the critical section while the value of ”looked” is ”true”.In the main text, the behavior of the protocol is explained in detail using pseudo-code and diagrams.

  For the formal verification of the TAS protocol, the proof scores and the proof scripts generated by entering them into CiMPG are shown.

- Chapter 4: Qlock protocol
  This chapter explained the specification and the proof score of the protocol, also automatic proof using CiMPA and CiMAG.

  Qlock protocol is a mutual exclusion protocol that uses a queue, a kind of data structure, and unlike TAS protocol, it has three sections:

a reminder section, a critical section, and waiting section. Unlike TAS protocol, there are three sections: the Remainder Section, the Critical Section, and the Waiting Section. Any selected process located in the Remainder Section first transits to the Waiting Section, where its ID is inserted into the queue. If the ID of the selected arbitrary process located in the waiting section matches the top element of the queue, the process can transition to the critical section. When a process located in the critical section leaves the critical section, the first element of the queue (the ID of the process located in the critical section) is removed. In the main text, the behavior of the protocol is explained in detail using pseudo-code and diagrams.

For the formal verification of the Qlock protocol, the proof scores and the proof scripts generated by entering them into CiMPG are shown.

- Chapter 5: Anderson protocol
This chapter explained the specification and the proof score of the protocol, also automatic proof using CiMPA and CiMAG.Not only that, but it also described the cause of the failure to complete the proof and the solution to it.

  Anderson protocol, like Qlock protocol, has three sections: the Remainder section, the Waiting section, and the Critical section. This protocol performs mutual exclusion by allocating a waiting area to a process that is about to transition to the critical section. This protocol is more complex than the two protocols described above and handles more variables than them. It also had a larger number of invariants that needed to be proven. In the main text, the behavior of the protocol is explained in detail using pseudo-code and diagrams. The formal verification of the Anderson protocol was not easy to complete, unlike the formal verification of the other two mutual exclusion protocols. In the formal verification by proof scoring, the proof of one of the invariants could not be completed in any way. Therefore, we introduced a variable called the auxiliary variable. This is a variable that does not affect the behavior (specification) of the protocol. By introducing this variable, we were able to prove the invariant. We then used CiMPA and CiMPG for automatic proofs and were able to complete the formal verification of mutual exclusion property and other invariants of the Anderson protocol. In the main text, the proof scores and the proof scripts generated by entering them into CiMPG are shown.

- Chapter 6: Lessons Learned
This chapter described what learned through this study(or course re-

lated to the study).

Since it was my first time using a functional programming language, it needs to be more creative than in procedural programming languages, which was a challenge, but also a lot of fun. Understanding of programming was deepened through the use of functional programming languages.

In this study, several approaches based on theorem proving were applied to the mutual exclusion protocol to clarify the issues that formal methods face. We were able to realize the shortcomings and difficulties of formal methods by actually working on them.

7.2 Future work

- The target of the formal verification in this study was the mutual exclusion protocol. The previous chapter explained that this is a mechanism to limit (mutual exclusion) the number of processes using a shared resource, such as shared memory, to at most one process at any given time in a system. But the mutual exclusion protocol is not the only mechanism to perform mutual exclusion. An example is semaphore, which is a variable or abstract data type that provides a simple and convenient abstraction to control access to shared resources by multiple processes in a parallel programming environment. Semaphore is simply a record of how many resources are available, coupled with operations that rewrite the record of using and releasing the resource, and wait for the resource to become available. Semaphore that handles an arbitrary number of resources is called counting semaphore, while a semaphore whose value is limited to 0 and 1 is called binary semaphore. When semaphores are used to control the exclusion of critical sections, semaphores allow multiple tasks to enter the critical section (if the initial value is not 1), while mutual exclusion protocols allow only one task to enter the critical section at a time. In other words, binary semaphores and mutual exclusion protocols have the same function. And the most important property required for binary semaphores is also mutual exclusivity. Therefore, we will also conduct formal verification of the mutual exclusion property of binary semaphores.

- The most important property that a mutual exclusion algorithm must satisfy is mutual exclusion, but it is not the only property that is required. Fairness is one of them. Fairness is the property that ”a process that wants to use a shared resource can use the shared resource at some

time. As an example of a case where fairness is not satisfied, if there are three cash registers in a supermarket and there is a queue at each register, and if the queue is slow, the strategy is to queue back up at the end of the other queue, then if the customer is unlucky, he or she may not be able to pay the bill for some time. To make such other properties subject to formal verification, it is necessary to define a new state predicate.

- We have formally verified that Anderson protocol enjoys the mutual exclusion property by introducing an auxiliary variable, and we Completed formal verification of it. Since the proof of one of the invariants of the Anderson protocol was extremely difficult, we introduced it as a way to overcome the difficulty. But it is unclear whether it is possible to prove that Anderson enjoys the mutual exclusion property without introducing auxiliary variables, so we want to disclose this point. And in the future, we want to complete the proof without introducing any auxiliary variables, or if it is necessary, to disclose theoretically why it is necessary. To clarify this, we should also tackle the formal verification of mutual exclusion property of other mutual exclusion protocols.

- Mutual exclusion protocols are not the only protocols that are subject to formal verification. For example, communication protocols and authentication protocols. Since the properties to be verified vary depending on the type of protocol, it can expect to gain new knowledge by working on case studies of protocols other than mutual exclusion protocols.

Bibliography

[1] K.Futatsugi, K.Ogata, and M.Nakamura. ”Introducting CafeOBJ(1) Formal method and CafeOBJ,” *JSSST Computer Software*, vol. 25, no. 2, pp.1_13, 2008

[2] K.Futatsugi, K.Ogata, and M.Nakamura. ”Introducting CafeOBJ(2) Syntax and Semantics,” *JSSST Computer Software*, vol. 25, no. 2, pp.14_27, 2008

[3] K.Futatsugi, K.Ogata, and M.Nakamura. ”Introducing CafeOBJ(3) : Equational Reasoning and Term Rewriting Systems,” *JSSST Computer Software*, vol. 25, no. 3, pp.69_80, 2008

[4] K.Futatsugi, K.Ogata, and M.Nakamura. ”Introducing CafeOBJ (4) : Verification with Proof Scores,” *JSSST Computer Software*, vol. 25, no. 4, pp.68_84, 2008

[5] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer and C. Talcott, All About Maude - A High-Performance Logical Framework. Berlin and Heidelberg, Springer, 2007.

[6] A. Riesco and K. Ogata, ”Prove it! Inferring formal proof scripts from CafeOBJ proof scores,” *ACM Trans. Softw. Eng. Methodol.*, vol. 27, no. 2, pp. 6:1–6:32, 2018.

[7] T. E. Anderson, ”The performance of spin lock alternatives for shared-memory multiprocessors,” *IEEE Trans. Parallel Distrib. Syst.*, vol. 1, no. 1, pp. 6–16, 1990.

[8] D. D. Tran and K. Ogata, ”Formal verification of an abstract version of Anderson protocol with CafeOBJ, CiMPA and CiMPG,” in *SEKE 2020*, 2020, pp. 287–292.

[9] N. Asae, K. Ogata and D. D. Tran, ”Formal verification of Anderson mutual exclusion protocol by introducing an auxiliary variable ,” in *SEKE 2021*, 2021, pp. 126–131.