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

Title	An Investigation of Applications of State Machines [課題研究報告書]
Author(s)	Ferdous, Mohammad Farhan
Citation	
Issue Date	2018-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/15470
Rights	
Description	Supervisor:緒方　和博，情報科学研究科，修士

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

An Investigation of Applications of State Machines

By Mohammad Farhan Ferdous

A research project report submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kazuhiro Ogata

An Investigation of Applications of State Machines

By Mohammad Farhan Ferdous (1510215)

A research project report submitted to
School of Information Science,
Japan Advanced Institute of Science and Technology,
in partial fulfillment of the requirements
for the degree of
Master of Information Science
Graduate Program in Information Science

Written under the direction of
Professor Kazuhiro Ogata

and approved by
Professor Kazuhiro Ogata
Professor Kunihiko Hiraishi
Professor Toshiaki Aoki

August, 2018 (Submitted)

Copyright © 2018 by Mohammad Farhan Ferdous

Acknowledgements

This Masters research report would not have been understandable without the proper direction and the help of my main supervisor Professor Kazuhiro Ogata. I might want to offer my genuine thanks to him for the continuous support of my exploration as well as my life. I also grateful to others professors our dean Professor Tojo, Professor Iida and others Professors.

My true thanks additionally go to Nguyen Thi Thanh Tam for making the SMGA for graphical animations of state machine tool which is a free web-based illustration application for originators and designers. The tool has helped me to actualize my thought in this examination.

To wrap things up, I might want to thank my parents for their sacrifice for me and continuous support me. To each one of those individuals especially my all labs members and all JAIST and Bangladeshi friends who helped me to survive the JAIST life.

Contents

1 Introduction **5**
- 1.1 Overview . . . 5
- 1.2 Aim and Contribution . . . 6
- 1.3 Report Outline . . . 6

2 Preliminaries **8**
- 2.1 Mutual Exclusion . . . 8
- 2.2 State Machine and Invariants . . . 8
- 2.3 Kripke Structures and LTL . . . 10
- 2.4 Maude . . . 11
- 2.5 SMGA . . . 12

3 Ticket Mutual Exclusion Protocol **15**
- 3.1 FTicket: A Flawed Version of Ticket Protocol . . . 15
  - 3.1.1 Specification of FTicket in Maude and State Transition Diagram . . . 15
  - 3.1.2 Specification of FTicket as State Machines . . . 16
  - 3.1.3 Model Checking of FTicket . . . 18
  - 3.1.4 Graphical animations of FTicket Counterexamples . . . 19
- 3.2 Ticket Protocol . . . 20
  - 3.2.1 Specification of Ticket in Maude and State Transition Diagrams . . . 22
  - 3.2.2 Specification of Ticket as State Machines . . . 23
  - 3.2.3 Model Checking of Ticket . . . 24
  - 3.2.4 Graphical Animations of Ticket . . . 26
- 3.3 Non-deterministic version of Ticket Protocol . . . 28
  - 3.3.1 Specification of ND-Ticket . . . 30
  - 3.3.2 Specification of ND-Ticket as State Machines . . . 30
  - 3.3.3 Model Checking of ND-Ticket . . . 32

4 Anderson Mutual Exclusion Protocol **33**
- 4.1 FAnderson: A Flawed Version of Anderson Protocol . . . 33
  - 4.1.1 Specification of FAnderson in Maude and State Transition Diagrams 34
  - 4.1.2 Specification of FAnderson as State Machines . . . 34
  - 4.1.3 Model Checking of FAnderson . . . 37
  - 4.1.4 Graphical Animations of FAnderson Counterexamples . . . 37

4.2 Anderson Protocol . . . 39
4.2.1 Specification of Anderson in Maude and State Transition Diagrams 40
4.2.2 Specification of Anderson as State Machines . . . 41
4.2.3 Model Checking of Anderson . . . 43
4.2.4 Graphical Animations of Anderson . . . 45
4.3 Non-deterministic version of Anderson Protocol . . . 46
4.3.1 Specification of ND-Anderson in Maude and State Transition Diagrams . . . 48
4.3.2 Specification of ND-Anderson as State Machines . . . 49
4.3.3 Model Checking of ND-Anderson . . . 51

5 Qlock Mutual Exclusion Protocol 52
5.1 FQlock0: A Flawed Version of Qlock0 Protocol . . . 52
5.1.1 Specification of FQlock0 in Maude and State Transition Diagrams . 52
5.1.2 Specification of FQlock0 as State Machines . . . 53
5.1.3 Model Checking of FQlock0 . . . 55
5.1.4 Graphical Animations of FQlock0 Counterexamples . . . 56
5.2 FQlock1 Protocol . . . 57
5.2.1 Specification of FQlock1 in Maude and State Transition Diagrams . 59
5.2.2 Specification of FQlock1 as State Machines . . . 59
5.2.3 Model Checking of FQlock1 . . . 61
5.2.4 Graphical Animations of FQlock1 . . . 63
5.3 Qlock Mutual Exclusion Protocol . . . 64
5.3.1 Specification of Qlock in Maude . . . 66
5.3.2 Specification of Qlock as State Machines . . . 67
5.3.3 Model Checking of Qlock . . . 68
5.3.4 Graphical Animations of Qlock . . . 71
5.4 Non-deterministic version of Qlock . . . 72
5.4.1 ND-Qlock . . . 73
5.4.2 Specification of ND-Qlock as State Machines . . . 74
5.4.3 Model Checking of ND-Qlock . . . 76

6 Discussion 77
6.1 Summarized diagram of the report . . . 77
6.2 Model checking protocol code testing . . . 77
6.3 Model checking protocol design testing . . . 78

7 Conclusion 80

List of Figures

2.1 Mutual exclusion . . . 9
2.2 State Machine . . . 9
2.3 Invariant . . . 10
2.4 Kripke structures and LTL . . . 11
2.5 A picture of Maude display screen . . . 13
2.6 A picture of QLOCK protocol . . . 13

3.1 State Transition Diagram of FTicket . . . 17
3.2 Counterexample for FTicket of states 0 and 1 . . . 20
3.3 Counterexample for FTicket of states 3 and 6 . . . 21
3.4 Counterexample for FTicket of states 12 and 20 . . . 21
3.5 Counterexample for FTicket of state 28 . . . 21
3.6 State Diagram of Ticket . . . 23
3.7 States 0 and 1 of Ticket . . . 27
3.8 States 3 and 7 of Ticket . . . 27
3.9 States 10 and 12 of Ticket . . . 28
3.10 States 16 and 20 of Ticket . . . 28
3.11 States 22 and 24 of Ticket . . . 28
3.12 States 26 and 28 of Ticket . . . 29
3.13 State 30 of Ticket . . . 29
3.14 State Transition Diagram of ND-Ticket . . . 31

4.1 State Transition Diagram of FAnderson . . . 35
4.2 Counterexample for FAnderson of states 0 and 1 . . . 38
4.3 Counterexample for FAnderson of states 3 and 6 . . . 39
4.4 Counterexample for FAnderson of states 12 and 20 . . . 39
4.5 Counterexample for FAnderson of state 28 . . . 39
4.6 State transition diagram of Anderson . . . 41
4.7 States 0 and 1 of Anderson . . . 46
4.8 States 3 and 7 of Anderson . . . 46
4.9 States 10 and 12 of Anderson . . . 46
4.10 States 16 and 20 of Anderson . . . 47
4.11 States 22 and 24 of Anderson . . . 47
4.12 States 26 and 28 of Anderson . . . 47
4.13 State 30 of Anderson . . . 48

4.14 State transition diagram of ND-Anderson 49

5.1 State Transition Diagram of FQLOCK0 54
5.2 Counterexample for FQlock0 of states 0 and 1 57
5.3 Counterexample for FQlock0 of states 3 and 6 58
5.4 Counterexample for FQlock0 of states 13 and 23 58
5.5 Counterexample for FQlock0 of state 33 58
5.6 State Transition Diagram of FQLOCK1 60
5.7 States 0 and 1 of counterexample of the lockout freedom property for FQLOCK1 64
5.8 States 2 and 3 of counterexample of the lockout freedom property for FQLOCK1 64
5.9 States 4 and 5 of counterexample of the lockout freedom property for FQLOCK1 65
5.10 States 6 and 7 of counterexample of the lockout freedom property for FQLOCK1 65
5.11 States 8 and 9 of counterexample of the lockout freedom property for FQLOCK1 65
5.12 States 10 and 11 of counterexample of the lockout freedom property for FQLOCK1 66
5.13 State 12 of counterexample of the lockout freedom property for FQLOCK1 66
5.14 State Transition Diagram of QLOCK 67
5.15 States 0 and 1 of QLOCK 72
5.16 States 2 and 3 of QLOCK 72
5.17 States 4 and 5 of QLOCK 72
5.18 State 6 of QLOCK 73
5.19 State Transition Diagram of ND-QLOCK 75

6.1 Summarize diagram of the project report 78
6.2 Model checking protocol code testing 79
6.3 Model checking protocol design testing 79

7.1 Summarize results of all protocols 82

Chapter 1

Introduction

1.1 Overview

The world essentially relies upon programming. It is difficult to try and envision our lives without utilization of any software. The societal dependability is nearly the same as that of programming. How many individuals rely upon programming must addition later on. In this manner, we need dependable innovations to make programming genuinely solid.

State machines[1] are spoken to utilizing state diagrams. The output of a state machine is a component of the input and the present state. State machines assume a noteworthy part in regions, for example, electrical engineering, phonetics, software engineering, biology, science, mathematics, and logic. They are best utilized as a part of the displaying of utilization conduct, software engineering, the design of hardware digital systems, network protocols, compilers, and the study of computation and languages.

The state machine consists of a set of states, some of which are initial states and a binary relation over states. Elements of binary relation are called (state) transitions. State machines can be used to formalized various kind of systems, such as mutual exclusion protocols, communication protocols, and authentication protocols.

The mathematical expression of the state machine as follows:

A state machine M is $\langle S, I, T \rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, and here $T \subseteq S \times S$ is a binary relation over S. $(s, s') \in T$ is called a state transition of state machine M and defined as $s \to s'$. Where s' called a successor state of s. The set R of reachable states : (1) $I \subseteq R$ and (2) if $s \subseteq R$ and $s \to s'$, then $s' \in R$. A state machine predicate p is invariant then we can write M iff $(\forall s \in R)$ p(s). We can see from Fig.4 about invariant. A state predicate p can be interpreted as a set P of states where we can write $(\forall s \in P)$p(s) and $(\forall s \notin P) \neg p(s)$.

In the project report, We used Maude which is a rewriting logic-based computer language equipped with model checking facilities. We also conduct SMGA which is a state machine graphical animation tool. We have used as the concrete example of different types of mutual exclusion protocol such as Ticket, Anderson, and Qlock wrong, right and Non-deterministic version respectively. SMGA[2] is a graphical animation tool which used for drawing such a mutual exclusion protocol in each state.

1.2 Aim and Contribution

The aim of the project research is to learn state machines, how to mathematical formalize as the state machine, how to formalize systems as state machines, how to describe state machines in a formal specification language such as Maude, and how to model check that state machine enjoys properties based on such formal specification. How to develop graphical animations of the state machines used in the case studies with SMGA[2] by rebuilding the graphical animations.

There are many possible ways to describe state machines in many formal specification languages. One possible way formal specification language is Maude[3], a direct successor of OBJ3, the most famous algebraic specification language. Maude has been used to describe many state machines formalizing various kinds of systems and model check that such state machines enjoy desired properties formalized as invariants based on such formal specification in Maude.

It is worth learning machines, how to formalize systems as state machines, how to describe state machines in Maude and how to model check that state machines, how to describe state machines in Maude and how to model check that state machines desired in invariant properties based on such formal specifications with Maude. Some mutual exclusion protocols, such as the Flawed version of Ticket protocol, the right version of Ticket, Non-deterministic version of Ticket protocol, Flawed version of Anderson protocol, the right version of Anderson, Non-deterministic version of Anderson protocol. The flawed version of QLOCK0, version of QLOCK1, the right version of QLOCK, Non-deterministic version of QLOCK protocol, used as concrete examples to conduct the project.

A tool called SMGA for graphical animations of state machines has been developed. We used in the different case studies such as FTicket, Ticket, FAnderson, Anderson, FQLOCK0, FQLOCK1 , QLOCK with SMGA by rebuilding the graphical animations.

1.3 Report Outline

The rest of the project report is organized as follows :

- Chapter 2: Preliminaries

  This chapter presents some preliminaries such as Mutual exclusion, State Machine and Invariants, Kripke Structures and LTL, Maude and SMGA.

- Chapter 3: Ticket Mutual Exclusion Protocol

  This chapter presents FTicket: A Flawed Version of Ticket Protocol, Ticket Protocol, Non-deterministic version of Ticket Protocol,

- Chapter 4: Anderson Mutual Exclusion Protocol

  This chapter presents FAnderson: A Flawed Version of Anderson Protocol, Anderson Protocol,Non-deterministic version of Anderson Protocol,

- Chapter 5: Qlock Mutual Exclusion Protocol

  This chapter presents FQlock0: A Flawed Version of Qlock0 Protocol, FQlock1 Protocol, Qlock Mutual Exclusion Protocol, Non-deterministic version of Qlock.

- Chapter 6: Discussion

  This chapter discusses summery of the project work.

- Chapter 7: Conclusion

  This chapter concludes the research project.

Chapter 2

Preliminaries

2.1 Mutual Exclusion

A mutual exclusion (mutex) [4] is a program object that averts synchronous access to a shared resource. This idea is utilized as a part of simultaneous programming with a critical section, a bit of code in which procedures or strings get to a shared resource. Just a single string claims the mutex at once, in this way a mutex with a remarkable name is made when a program begins. At the point when a string holds a resource, it needs to bolt the mutex from different strings to anticipate simultaneous access of the resource. After discharging the asset, the string opens the mutex.

Expect that many agents (or procedures) are viewing for a equipment, however, at any moment of time just a single agent can utilize the equipment. That is, the operators are commonly barred from utilizing the equipment. A convention (component or calculation) which can accomplish the mutual exclusion is called "mutual exclusion protocol".

For example: How to make sure at most one person is given the permission to use the shared bike? A queue may be used to do so. Here suppose Emma, David and Alice enqueue their initials into the queue in this order. Emma is the 1st person who is given the permission. When her use is done, the queue is dequeued. David is the 1st person who is given the permission. Fig. 2.1 shows a mutual exclusion protocol for use of the shared bike [5].

2.2 State Machine and Invariants

A state machine M is $\langle S, I, T \rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, and here $T \subseteq S \times S$ is a binary relation over S. $(s, s') \in T$ is called a state transition of state machine M and defined as $s \rightarrow s'$. Where s' called a successor state of s. Fig. 2.2 shows that the concept of state machine [5]. The state machine and their properties can be used to formalize various system and requirements. Systems verification can then conducted by theorem proving the state machines enjoy properties.

A case of a straightforward system that can be demonstrated by a state machine is a turnstile [6]. We principally utilized a turnstile for control access to metros and event

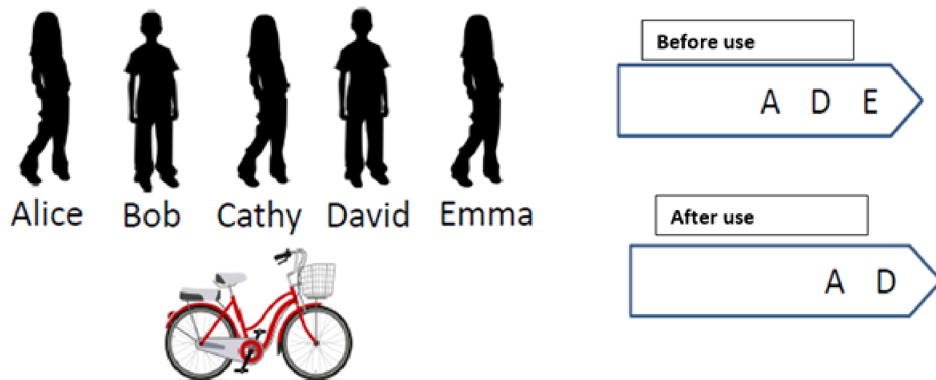

Figure 2.1: Mutual exclusion

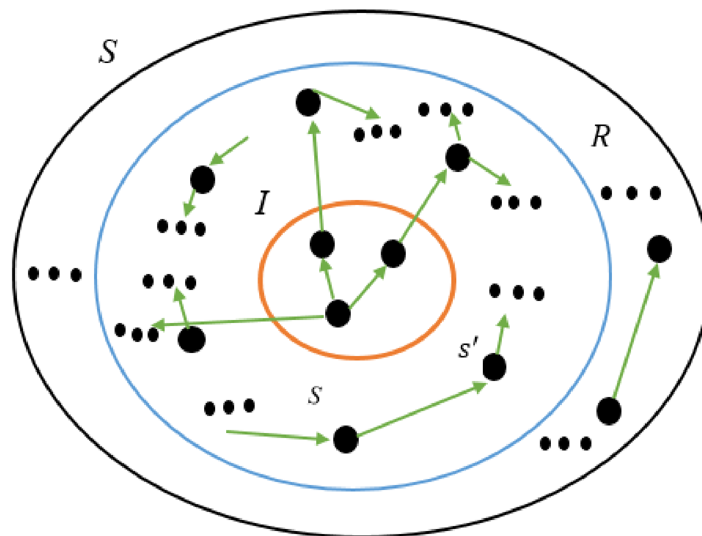

Figure 2.2: State Machine

congregation rides, is a door with three turning arms at midriff tallness, one over the passage. Essentially, arms are bolted at first, obstructing the passage, keeping supporters from going through. In the wake of putting a coin or token in a space on the entryway at that point opens the arms, enabling a solitary client to push through. At the point when the client goes through after arms are bolted again until the point when another coin is embedded.

On the off chance that we considered as a state machine, here are two conceivable conditions of the entryway: Locked and Unlocked. There are two conceivable sources of info that influence its state: putting a coin in the slot (coin) and pushing the arm (push). In the locked state, pushing on the arm has no impact; regardless of how often the info

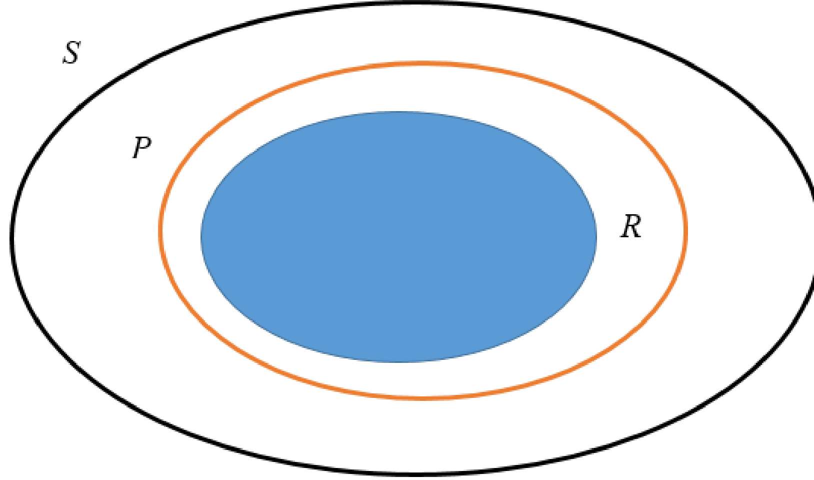

Figure 2.3: Invariant

push is given, it remains in the bolted state. When putting a coin in the machine that is, giving the machine a coin input moves the state from Locked to Unlocked. In the opened state, putting extra coins in has no impact; that is, giving extra coin inputs does not change the state. In any case, a client pushing through the arms, giving a push input, move the state back to Locked.

The set R of reachable states : (1) $I \subseteq R$ and (2) if $s \subseteq R$ and $s \rightarrow s'$, then $s' \in R$. A state machine predicate p is invariant then we can write M iff $(\forall s \in R)$ p(s). We can see from Fig. 2.3 about the invariant. A state predicate p can be interpreted as a set P of states where we can write $(\forall s \in P)$p(s) and $(\forall s \notin P)\neg p(s)$.

2.3 Kripke Structures and LTL

A kripke structure [5] K is $\langle S, I, P, L, T \rangle$, where S, I, T are same as state machine M, although S is total. The set $I \subseteq S$ of initial states. Where $T \subseteq S \times S$ is a binary relation of states. A set $P \subseteq U$ of atomic state propositions. L is a labeling function whose type $S \rightarrow 2^p$. A path π of a kripke structure K is $s_0; .....; s_i; s_{i+1};......$of S such that $(s_i, s_{i+1}) \in T$ for each i. π is defined as a computations if $\pi(0) \in I$. Here $\subseteq P$ is defined set of all the paths of $\subseteq K \subseteq C$ is all computation of $\subseteq K$ and $\subseteq K$ is set of all kripke structures. Here for the kripke structure K is $\langle S, I, P, L, T \rangle$.

The formulas of the linear temporal logic (LTL) is denoted as : $\varphi ::= T|p|\neg\varphi|\varphi \wedge \varphi|O\varphi|\varphi u \varphi$ where $p \in P$. We can in like manner see from Fig. 2.4 of Kripke structures and LTL. Let F be the set of all formulas in LTL for K. An arbitrary path $\pi \in P$ of K and an arbitrary LTL formula $\varphi \subseteq F$ of K, $K, \pi \models \varphi$ is inductively defined as $K, \pi \models \top$,

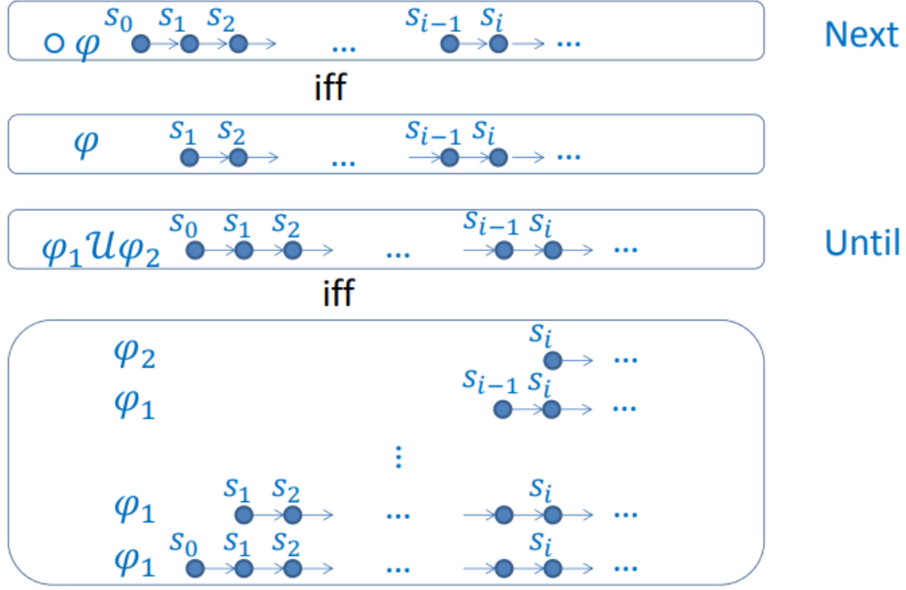

Figure 2.4: Kripke structures and LTL

$K, \pi \models p$ if and only if (iff) $p \in L(\pi(0))$, $K, \pi \models \neg\varphi_1$ iff $K, \pi \not\models \varphi_1$, $K, \pi \models \varphi_1 \wedge \varphi_2$ iff $K, \pi \models \varphi_1$ and $K, \pi \models \varphi_2$, $K, \pi \models \bigcirc\varphi_1$ iff $K, \pi_1 \models \varphi_1$ $K, \pi \models \varphi_1 \cup \varphi_2$ iff there exists a natural number i s.t. $K, \pi_i \models \varphi_2$ and for all natural numbers $j < i$, $K, \pi_j \models \varphi_1$, where φ_1 and φ_2 are LTL formulas. Then, $K \models \varphi$ iff $K, \pi \models \varphi$ for each computation $\pi \in C$ of K. The temporal connectives O and U are called the next operator and the until operator, respectively. The other logical and temporal connectives are defined as usual as follows: $\bot \triangleq \neg\top$, $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\Diamond\varphi \triangleq \top U \varphi$, $\Box\varphi \triangleq \neg(\Diamond\neg\varphi)$, and $\varphi_1 \leadsto \varphi_2 \triangleq \Box(\varphi_1 \Rightarrow \Diamond\varphi_2)$. $\Diamond, \Box$ and $\leadsto$ are called eventually, always and leadsto operators.

2.4 Maude

Maude[3] is a rewriting logic-based based computer languages and framework. It is one of the immediate successors of OBJ3 [7], the most well known mathematical particular dialect and framework for the most part planned by Joseph A. Goguen. Details can be composed in Maude adaptably. Acquainted as well as commutative parallel administrators can be uninhibitedly utilized as a part of details and after that complex simultaneous and circulated frameworks can be concisely indicated. Maude is furnished with numerous functionalities, among which are show checking (the Maude look summon and the Maude LTL display checker), and meta-programming. A metaprogram is a program that takes programs as sources of info and plays out some valuable calculations. Other than that, Maude is outfitted with the pursuit summon that thoroughly crosses the reachable states from an offered state to discover states that match some example and fulfill some condition

in a broadness first way. It is likewise outfitted with a metalevel work that is the partner of the inquiry summon. Fig. 2.5 shows the initial screen of Maude and how to feed the maude file in the system.

Maude underpins in a methodical and effective way legitimate reflection. This makes Maude strikingly extensible and ground-breaking, bolsters an extensible polynomial math of module creation activities, and permits numerous progressed metaprogramming and metalanguage applications. Undoubtedly, the absolute most intriguing uses of Maude are metalanguage applications, in which Maude is accustomed to making executable conditions for various rationales, hypothesis provers, dialects and models of calculation.

2.5 SMGA

On the off chance that the state machine graphical movement apparatus manages state machines inside, we have to outline an interior portrayal of state machines or embrace some current ones. It is cumbersome to request that human clients compose state machines in such an inside portrayal. Subsequently, we have to outline a particular dialect for state machines or receive some current ones. Provided that this is true, it is important to decipher state machines written in a detail dialect into those written in an interior portrayal. We ought to build up various interpreters for numerous particular dialects to make it workable for any state machines to be graphically enlivened. Since numerous detail dialects have been and would be proposed, in any case, it would not be shrewd to build up an interpreter for every particular dialect since it's anything but a minor assignment to grow even one interpreter for one determination dialect. we would like to make is extensible as well as maintainable as much as possible. one of such technologies is Scalable Vector Graphics(SVG) used to define graphics for the web. SVG has several methods for drawing paths, boxes, circles, texts and graphics vector. It is helpful to use SVG for drawing pictures of state machines. Since SVG supported by almost all major web browsers it makes it possible to make the tool available in as many platforms and/or environments possible. Several tools with which SVG animations can be made have been developed. one of them is DRAW-SVG [8], which we have used in this research. DRAW-SVG is a free online drawing application for designers and develops, making it possible to create fully standard compliant SVG. Fig. 2.6 demonstrates the case of Qlock protocol of an initial state which is drawing by SMGA.

We have not planned the state machine graphical movement apparatus to such an extent that it manages state machines inside however composed it with the end goal that it essentially takes a limited calculation of a state machine. This is on the grounds that devices, for example, show checkers, that can manage state machines can create limited calculations of state machines. We have to settle how to speak to each condition of state machines and limited arrangements of states. It would be significantly less demanding, in any case, to change some extraordinary state portrayals to that utilized for the state machine graphical movement device than to decipher state machines written in a determination dialect into those written in another. In addition, it is clear to change some extraordinary portrayals of limited state successions to that utilized for the state machine

```
                   \||||||||||||||||||/
                 --- Welcome to Maude ---
                   /||||||||||||||||||\
           Maude 2.7.1 built: Jun 27 2016 16:43:23
            Copyright 1997-2016 SRI International
                 Thu Jul 19 23:44:12 2018
Maude> in TICKET
==========================================
fmod LABEL
==========================================
fmod PID
==========================================
mod TICKET
==========================================
.
Reading in file: "model-checker.maude"
==========================================
fmod LTL
==========================================
fmod LTL-SIMPLIFIER
==========================================
fmod SAT-SOLVER
==========================================
fmod SATISFACTION
==========================================
fmod MODEL-CHECKER
Done reading in file: "model-checker.maude"
==========================================
mod TICKET-PREDS
==========================================
```

Figure 2.5: A picture of Maude display screen

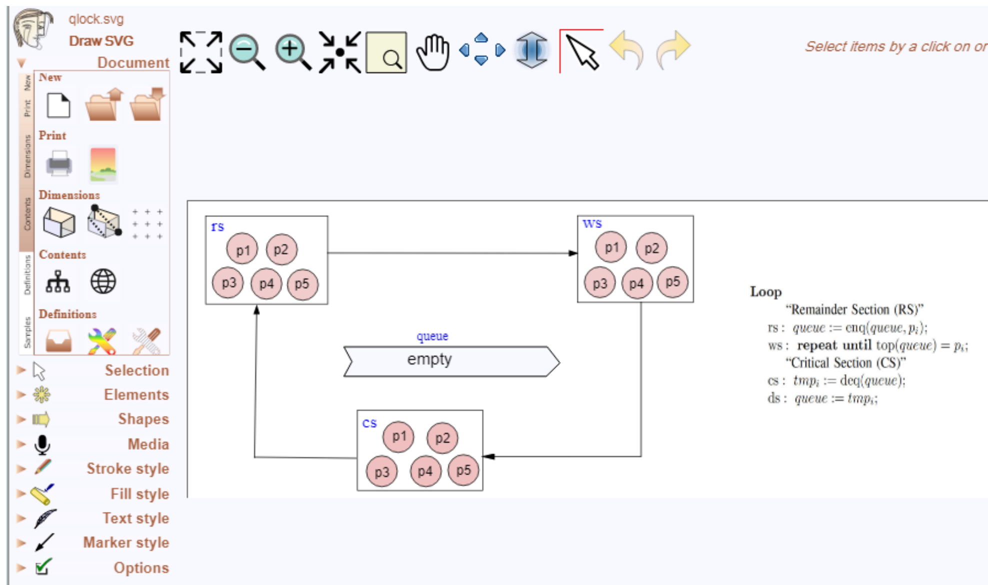

Figure 2.6: A picture of QLOCK protocol

graphical liveliness apparatus once unique state portrayals have been changed into that utilized for the device.

If each state in a finite computation of a state machine is graphically represented, the finite computation is essentially a film of a graphical animation of the state machine. Therefore, it would suffice to allow human users to intuitively design graphical state representations (or images or pictures) of state machines.

We have actualized a state machine graphical activity device [9] mostly in light of the

fact that human clients could perceive some valuable examples in energized limited calculations of state machines and guess helpful lemmas from the examples. Tam Thi Than Nguyen executed an instrument which is open on the site https://tamntt.bitbucket.io /Research/GraphicalAnimation/. The apparatus fundamentally takes a limited calculation and plays its graphical movement. The device enables human clients to configuration pictures or flares of liveliness, change the speed of movements, and select a few expresses that fulfill a few conditions and additionally limitations from a limited calculation.

Chapter 3

Ticket Mutual Exclusion Protocol

3.1 FTicket: A Flawed Version of Ticket Protocol

The FTicket protocol [10] is a mutual exclusion protocol based on issuing tickets to a critical section. next and serve are natural number variables share by all process. $ticket[i]$ is a natural number variable that is local to process i.

FTicket for a process i can be described as follows:

Loop: "Remainder Section"
rs: *ticket*[i] := *next*;
l1: *next*:= (*next*+1) % N;
ws: **repeat until** *ticket*[i] = *serve*;
    "Critical Section"
cs: *serve*:= (*serve*+1) % N;

Here are four locations rs (remainder section), l1 (label 1), ws (waiting section), cs (critical section). We suppose that there are N processes. For each process i, there are two local variables: $ticket_i$ whose value is in $\{0, 1, \ldots, N - 1\}$. Initially, $ticket_i = 0$. Two global variables shared by the N processes: *next* whose value is in $\{0, 1, \ldots, N - 1\}$ and *serve* whose value is in $\{0, 1, \ldots, N - 1\}$. Initially the value of $next = 0$ and $serve = 0$. next represents the next ticket to the critical section that is to be issued to a process, while serve represents the ticket whose owner is in critical or allowed to enter it. When a process i tries to enter the critical section, it takes a ticket. A process's *ticket* is equal to serve, so it enter the critical section. When a process leaves there, it increments *serve* remainder N.

3.1.1 Specification of FTicket in Maude and State Transition Diagram

Here are two processes whose are denoted by p1 and p2. I, X and Y are Maude variables of process IDs, next and serve are process IDs, successively. $ticket[i]$ is a natural number

variable that is local to process i.

The state transitions of FTicket are specified as the following four rewrite rules:

```
rl [setTicket] : (pc[I]: rs) (ticket[I]: X) (next: Y)
=>  (pc[I]: l1) (ticket[I]: Y) (next: Y) .
rl [incTicket] : (pc[I]: l1) (next: Y)
=> (pc[I]: ws) (next: ((Y + 1) rem N)) .
rl [wait] : (pc[I]: ws) (ticket[I]: X) (serve: X)
=>  (pc[I]: cs) (ticket[I]: X) (serve: X) .
rl [incServe] : (pc[I]: cs) (serve: X)
=> (pc[I]: rs) (serve: ((X + 1) rem N)) .
```

setTicket, incTicket, wait, incServe are the names of the four rewrite rules, respectively. The details description of four rewrite rule follows:

rule 1(setTicket) : a process I is located at rs, the content of ticket is X, the content of next is Y. After that a process I is located at l1, the content of ticket is Y, the content of next is Y.

rule 2(incTicket) : a process I is located at l1, the content of next is Y. After that a process I is located at ws, the content of next is increments Y remainder N. Here the number of processes N = 2.

rule 3(wait) : a process I is located at ws, the content of ticket is X, the content of serve is X. After that a process I is located at cs, the content of ticket is X, the content of serve is X.

rule 4(incServe) : a process I is located at cs, the content of serve is X. After that a process I is located at rs, the content of serve is increments X remainder N. Here is the number of processes N = 2.

Fig. 3.1 shows the four state transition $setTicket_I$, $incTicket_I$, $wait_I$ and $incServe_I$ respectively. After the transition from one state to another state, we can indicate the process IDs I.

3.1.2 Specification of FTicket as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, l1, cs, ws} of locations.

Four kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(ticket[i] : X_N)$ - It says that the content of $ticket_i$ is X_N;
- $(serve : Z)$ - It says that the content of $serve$ is Z;
- $(next : Y)$ - It says that the content of $next$ is Y;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, $l \in$ Loc and $X_N \in$ Pid Nat are values and $X_1,\ldots,X_N$, Y, Z $\in$ Nat . We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in FTicket.

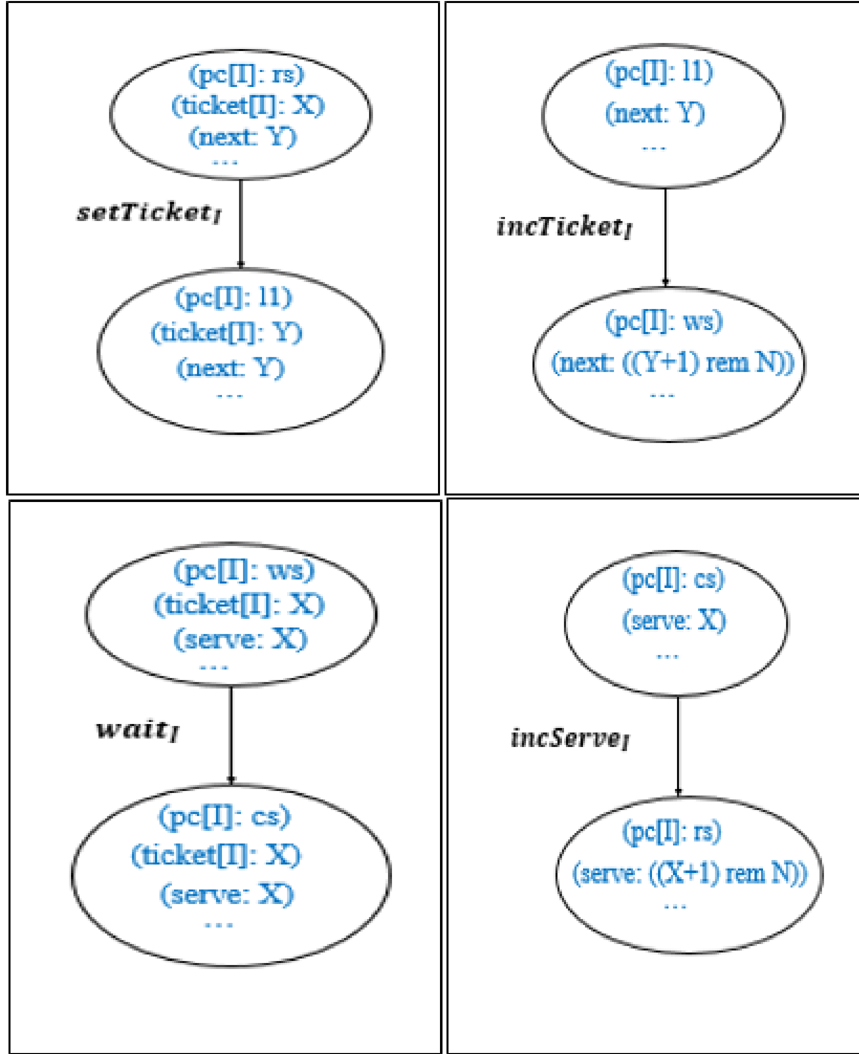

Figure 3.1: State Transition Diagram of FTicket

- Set of States, S = {(pc[1] : L_1) ... (pc[N] : L_N) (ticket[1] : X_1). ... (ticket[N] : X_N) (next : Y) (serve : Z) | L_1 ,..., L_N ∈ Loc, X_1,...,X_N , Y, Z ∈ Nat}.
- Initial State, I = {(pc[1] : rs) ... (pc[N] : rs) (ticket[1] : 0)... (ticket[N] : 0) (next : 0) (serve : 0)}
- $T_{setTicket}$ = {((pc[1] : L_1)... (pc[I] : rs) ... (pc[N] : L_N) (ticket[1] : X_1)... (ticket[I] : X_I)... (ticket[N] : X_N) (next : Y) (serve : Z), (pc[1] : L_1) ... (pc[I] : l1) ... (pc[N] : L_N)

(ticket[1] : X_1)...(ticket[I] : Y)...(ticket[N] : X_N) (next : Y) (serve : Z))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,...,X_N, Y, Z ∈ Nat}

- $T_{incTicket}$ = {((pc[1] : L_1) ...(pc[I] : l1)...(pc[N] : L_N)
  (next : Y) (serve : Z) (ticket[1] : X_1)...(ticket[I] : Y)...(ticket[N] : X_N),
  (pc[1] : L_1) ...(pc[I] : ws) ...(pc[N] : L_N) (serve : Z)
  (next : ((Y + 1)rem N) (ticket[1] : X_1)...(ticket[I] : Y)...(ticket[N] : X_N))
  | I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,...,X_N , Y, Z ∈ Nat}
- T_{wait} = {((pc[1] : L_1) ...(pc[I] : ws) ...(pc[N] : L_N)
  (ticket[1] : X_1)...(ticket[I] : X_I)...(ticket[N] : X_N) (serve : Z) (next : Y),
  (pc[1] : L_1) ...(pc[I] : cs) ...(pc[N] : L_N)
  (ticket[1] : X_1) ...(ticket[I] : X)...(ticket[N] : X_N)
  (next : Y) (serve : Z))
  | I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,...,X_N, Y, Z ∈ Nat}
- $T_{ncServe}$ = {((pc[1] : L_1)...(pc[I] : cs)...(pc[N] : L_N) (serve : Z) (next : Y)
  (ticket[1] : X_1)...(ticket[I] : X_I)...(ticket[N] : X_N),
  (pc[1] : L_1)...(pc[I] : rs)...(pc[N] : L_N) (serve: ((Z + 1)rem N))
  (next : Y) (ticket[1] : X_1)...(ticket[I] : X_I)...(ticket[N] : X_N))
  | I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, Y, Z ∈ Nat}

3.1.3 Model Checking of FTicket

The following search command used for checking the FTicket.

```
search [1] in FTICKET : init =>* (pc[p1]: cs) (pc[p2]: cs) S .
```

Maude finds a solution meaning FTicket does not enjoy the property. Here is also used following path search command : show path 28. This command shows each state of the FTicket version by Maude.

Where FTICKET is the module in which FTicket is specified and S is a Maude variable of state fragments. The search command finds the following counterexample:

```
state 0, Sys: next: 0 serve: 0 (pc[p1]: rs)
(pc[p2]: rs) (ticket[p1]: 0)  ticket[p2]: 0
===[ rl next: Y (pc[I]: rs) ticket[I]: X =>
(next: Y ticket[I]: Y) pc[I]: l1 [ label setTicket] . ]===>
state 1, Sys: next: 0 serve: 0 (pc[p1]: l1)
(pc[p2]: rs) (ticket[p1]: 0) ticket[p2]: 0
```

```
===[ rl next: Y (pc[I]: rs) ticket[I]: X =>
(next: Y ticket[I]: Y) pc[I]: l1 [label setTicket] . ]===>
state 3, Sys: next: 0 serve: 0 (pc[p1]: l1)
(pc[p2]: l1) (ticket[p1]: 0) ticket[p2]: 0
===[ rl next: Y pc[I]: l1 =>
next: ((Y + 1) rem 2) pc[I]: ws [label incTicket]  . ]===>
state 6, Sys: next: 1 serve: 0 (pc[p1]: ws)
(pc[p2]: l1) (ticket[p1]: 0) ticket[p2]: 0
===[ rl next: Y pc[I]: l1 =>
next: ((Y + 1) rem 2) pc[I]: ws [label incTicket]  . ]===>
state 12, Sys: next: 0 serve: 0 (pc[p1]: ws)
(pc[p2]: ws) (ticket[p1]: 0)  ticket[p2]: 0
===[ rl serve: X (pc[I]: ws) ticket[I]: X =>
serve: X pc[I]: cs [label wait] . ]===>
state 20, Sys: next: 0 serve: 0 (pc[p1]: cs)
(pc[p2]: ws) (ticket[p1]: 0) ticket[p2]: 0
===[ rl serve: X (pc[I]: ws) ticket[I]: X =>
serve: X pc[I]: cs [label wait] .  ]===>
state 28, Sys: next: 0 serve: 0 (pc[p1]: cs) (pc[p2]: cs)
(ticket[p1]: 0) ticket[p2]: 0
```

Maude can generate counterexample without any type of difficulties, which non-experts can not do it. Fig. 3.2 to 3.5 show that counterexample which found by Maude software for FTicket version.

3.1.4 Graphical animations of FTicket Counterexamples

There are three regions: ###keys, ###textDisplay and ###states. In the main region ###keys, the names of the detectable parts are composed. The request in which the names are composed ought to be the same as the request in which the comparing discernible parts are composed in each state. In the second region ###textDisplay, we could think of a few mandates about how to show accumulations, for example, lines and records. For instance, a rundown is on a level plane showed naturally with the end goal that the best component seems left-most and the base component seems acceptable most. We could guide the instrument to show a rundown in the turn around arrange or potentially vertically. In the third area ###states, a limited calculation is composed.

```
###keys
next serve pc[p1] pc[p2] ticket1 ticket2

###textDisplay

###states
(next: 0 serve: 0 (pc[p1]: rs) (pc[p2]: rs) (ticket1: 0) (ticket2: 0) ) ||
```

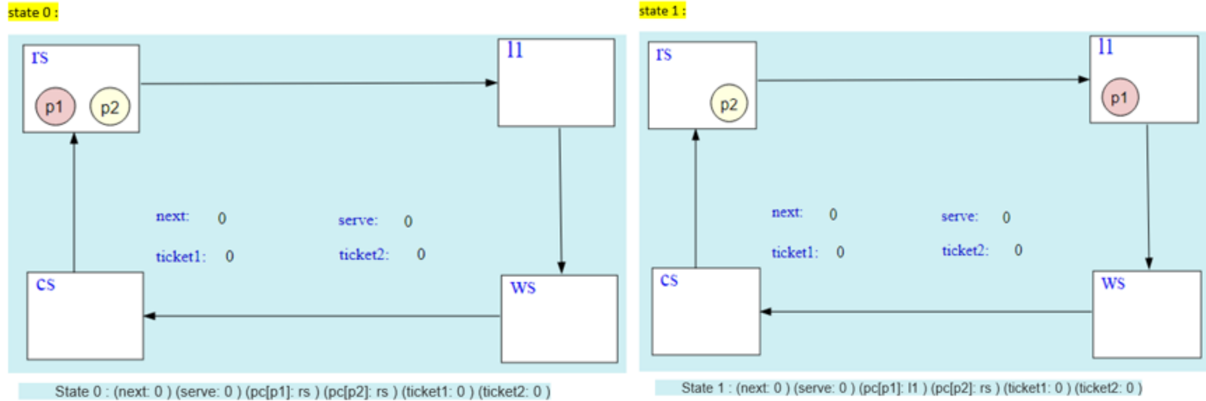

Figure 3.2: Counterexample for FTicket of states 0 and 1

```
(next: 0 serve: 0 (pc[p1]: l1) (pc[p2]: rs) (ticket1: 0) (ticket2: 0) ) ||
(next: 0 serve: 0 (pc[p1]: l1) (pc[p2]: l1) (ticket1: 0) (ticket2: 0) ) ||
(next: 1 serve: 0 (pc[p1]: ws) (pc[p2]: l1) (ticket1: 0) (ticket2: 0) ) ||
(next: 0 serve: 0 (pc[p1]: ws) (pc[p2]: ws) (ticket1: 0) (ticket2: 0) )||
(next: 0 serve: 0 (pc[p1]: cs) (pc[p2]: ws) (ticket1: 0) (ticket2: 0) ) ||
(next: 0 serve: 0 (pc[p1]: cs) (pc[p2]: cs) (ticket1: 0) (ticket2: 0)
```

- keys: This is a list of keys which are names of observable components in a state. The order in which the keys appear must be the same as the order in which the corresponding observable components appear in each state.
- textDisplay: This part specifies how the value of an observable component is displayed. When displaying a queue, if nothing is specified, it is displayed horizontally and its top appears left most. There may be the case, however, where its top should appear right most. Some values, such as stacks, may have to be displayed vertically instead.
- states: This is a finite computation of a state machine, namely a finite sequence of states. The sign || is a separator used to distinguish adjacent states.

We used SMGA for drawing the seven pictures for FTicket version. These pictures make it possible to reorganize at which location the each process is, what the value stored in next and serve and what the value stored in ticket1 and ticket2. The details description of each Fig. written in the discussion section. Here is only next: 1 in the state 6. But for the others states natural numbers variables next, serve, ticket1, ticket2 values are zero.

3.2 Ticket Protocol

The Ticket protocol [10] is a mutual exclusion protocol based on issuing tickets to a critical section. next and serve are natural number variables share by all process. $ticket[i]$ is a

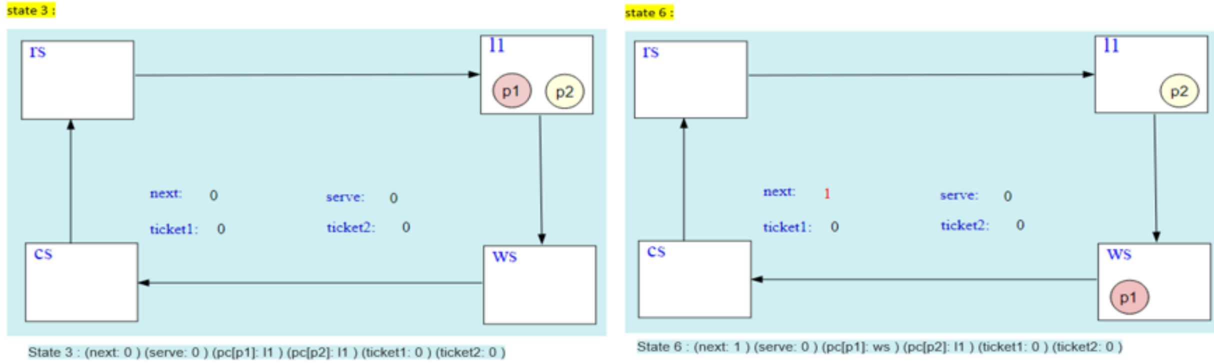

Figure 3.3: Counterexample for FTicket of states 3 and 6

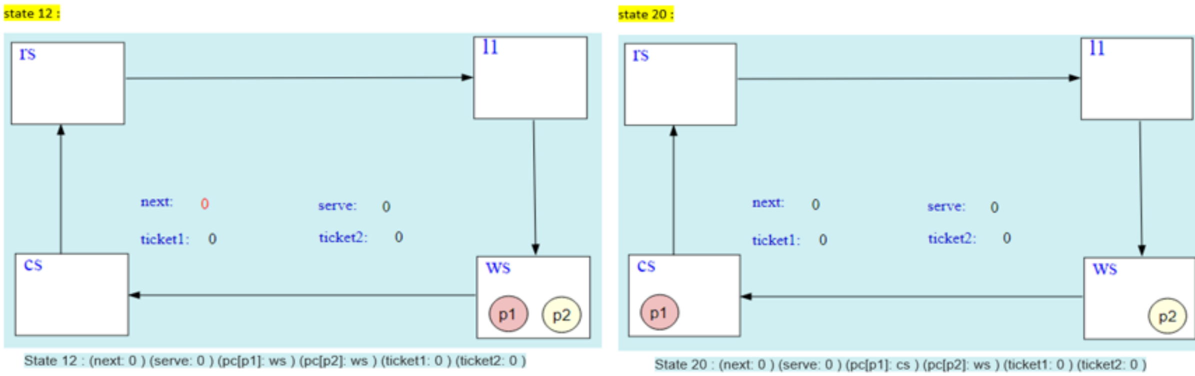

Figure 3.4: Counterexample for FTicket of states 12 and 20

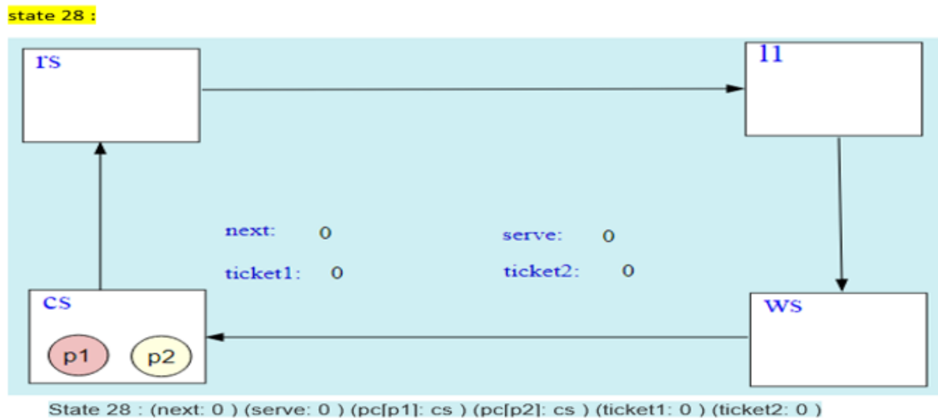

Figure 3.5: Counterexample for FTicket of state 28

natural number variable that is local to process i.

Ticket for a process i can be described as follows:

Loop: "Remainder Section"

rs: *ticket*[*i*] := fetch&incmode(*next*,*N*);
ws: **repeat until** *ticket*[i] = *serve*;
"Critical Section"
cs: *serve*:= (*serve*+1) % *N*;

There are used fetch&incmode atomic operation for implement the protocol. This is atomically reads a memory location, increments the value modulo N, writes the result into the memory location and return the old value. Here are three locations rs (remainder section), ws (waiting section), cs (critical section). We suppose that there are N processes. For each process i, there are two local variables: $ticket_i$ whose value is in $\{0, 1, \ldots, N - 1\}$. Initially, $ticket_i = 0$. Two global variables shared by the N processes: *next* whose value is in $\{0, 1, \ldots, N - 1\}$ and *serve* whose value is in $\{0, 1, \ldots, N - 1\}$. Initially the value of $next = 0$ and $serve = 0$. next represents the next ticket to the critical section that is to be issued to a process, while serve represents the ticket whose owner is in critical or allowed to enter it. When a process i tries to enter the critical section, it takes a ticket, that is, it indivisibly copies into it local variable *ticket* and increments *next* remainder N using $fetch\&incmode$. A process's *ticket* is equal to serve, so it enter the critical section. When a process leaves there, it increments *serve* remainder N.

For the variable x and a constant c whose type of natural numbers
fetch&incmode(x,n) conducts the following atomically (or indivisibly):
t := x; x:= (x + 1)%n; **return** t

3.2.1 Specification of Ticket in Maude and State Transition Diagrams

There are two processes whose are denoted by p1 and p2. I, X and Y are Maude variables of process IDs, next and serve are process IDs, successively. *ticket*[*i*] is a natural number variable that is local to process i.

The state transitions of Ticket are specified as the following three rewrite rules :

```
rl [incNxt&St] : (pc[I]: rs) (ticket[I]: X) (next: Y)
=> (pc[I]: ws) (ticket[I]: Y) (next: ((Y + 1) rem N)) .
rl [wait] : (pc[I]: ws) (ticket[I]: X) (serve: X)
=> (pc[I]: cs) (ticket[I]: X) (serve: X) .
rl [incServe] : (pc[I]: cs) (serve: X)
=> (pc[I]: rs) (serve: ((X + 1) rem N)) .
```

incNxt&St, wait, incServe are the names of the three rewrite rules, respectively. The details description of three rewrite rules follows:

rule 1(incNxt&St) : a process I is located at rs, the content of ticket is X, the content of next is Y. After that a process I is located at ws, the content of ticket is Y, the content of next is incremented Y remainder N. Here is the number of processes N because of N = 2.

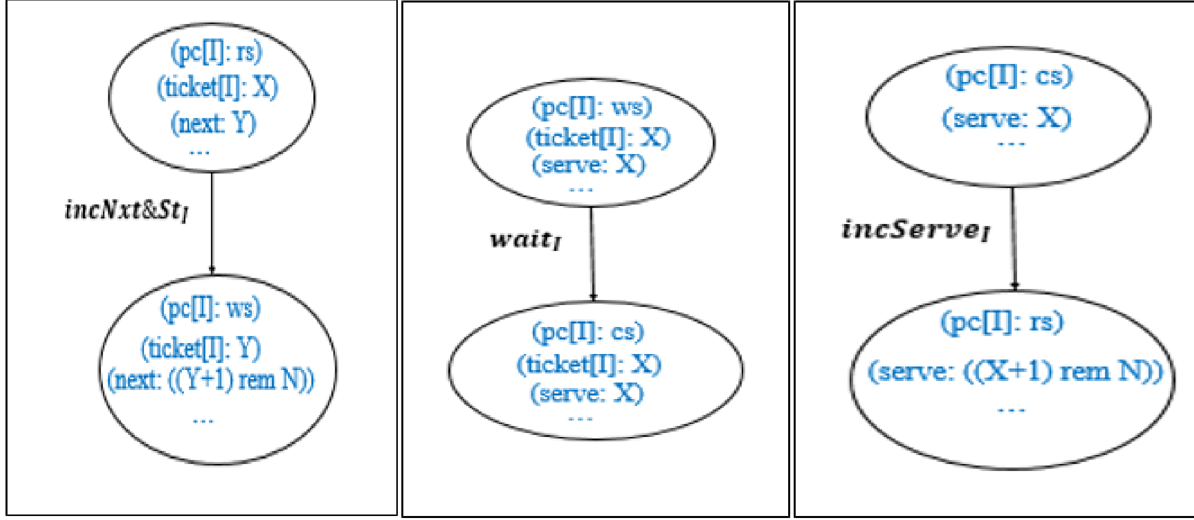

Figure 3.6: State Diagram of Ticket

rule 2(wait) : a process I is located at ws, the content of ticket is X, the content of serve is X. After that a process I is located at cs, the content of ticket is X, the content of serve is X.

rule 3(incServe) : a process I is located at cs, the content of serve is X. After that a process I is located at rs, the content of serve is incremented X remainder N. Here is the number of processes N and N = 2.

Fig. 3.6 shows the three state transition $incNxt\&St_I$, $wait_I$ and $incServe_I$ respectively. After the transition from one state to another state, we can indicate the process IDs I.

3.2.2 Specification of Ticket as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, cs, ws} of locations.

Four kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(ticket[i] : X_N)$ - It says that the content of $ticket_i$ is X_N;
- $(serve : Z)$ - It says that the content of $serve$ is Z;
- $(next : Y)$ - It says that the content of $next$ is Y;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, $l \in$ Loc and $X_N \in$ Pid Nat are values, ticket $\in$ Pid Nat and serve, next $\in$ Nat . We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in Ticket .

- Set of States, S = {(pc[1] : L_1) . . . (pc[N] : L_N)

(ticket[1] : X_1)...(ticket[N] : X_N) (next : Y) (serve : Z)
| L_1,...,L_N ∈ Loc, X_1,...,X_N, Y, Z ∈ Nat}.

- Initial State, I = {(pc[1] : rs)...(pc[N] : rs)
(ticket[1] : 0)...(ticket[N] : 0) (next : 0) (serve : 0)}.

- $T_{incNxt\&St}$ = {((pc[1] : L_1)...(pc[I] : rs) ...(pc[N] : L_N)
(ticket[1] : X_1) ...(ticket[I] : X_I) ...(ticket[N] : X_N) (next : Y) (serve : Z),
(pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N) (serve : Z)
(ticket[1] : X_1)...(ticket[I] : Y_I)...(ticket[N] : X_N) (next : ((Y + 1)rem N)))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, Y, Z ∈ Nat}

- T_{wait} = {((pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N)
(ticket[1] : X_1)...(ticket[I] : X_I) ...(ticket[N] : X_N) (serve : Z) (next : Y),
(pc[1] : L_1) ...(pc[I] : cs)...(pc[N] : L_N)
(ticket[1] : X_1)...(ticket[I] : X)...(ticket[N] : X_N) (serve : Z) (next : Y))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, Y, Z ∈ Nat}

- $T_{ncServe}$ = {((pc[1] : L_1)...(pc[I] : cs)...(pc[N] : L_N) (serve : Z) (next : Y)
(ticket[1] : X_1)...(ticket[I] : X_I)...(ticket[N] : X_N),
(pc[1] : L_1)...(pc[I] : rs)...(pc[N] : L_N) (serve: ((Z + 1)rem N))
(next : Y) (ticket[1] : X_1)...(ticket[I] : X_I)...(ticket[N] : X_N))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, Y, Z ∈ Nat}

3.2.3 Model Checking of Ticket

- **Maude search command**

The following search command used for checking the Ticket.

```
search [1] in TICKET : init =>* (pc[p1]: cs) (pc[p2]: cs) S .
```

Maude finds no solution meaning Ticket likely to enjoy the mutual exclusion property. Here is also used following path search command: show path 30 . This command shows each states of the Ticket version by Maude.

Where TICKET is the module in which Ticket is specified and S is a maude variable of state fragments. The search command finds the follows:

```
state 0, Sys: next: 0 serve: 0 (pc[p1]: rs) (pc[p2]: rs) (ticket[p1]: 0)
ticket[p2]: 0
===[ rl next: Y (pc[I]: rs) ticket[I]: X => (next: ((Y + 1) rem 2)
ticket[I]: Y) pc[I]: ws [label incNxt&St] . ]===>
state 1, Sys: next: 1 serve: 0 (pc[p1]: ws) (pc[p2]: rs) (ticket[p1]: 0)
ticket[p2]: 0
===[ rl next: Y (pc[I]: rs) ticket[I]: X => (next: ((Y + 1) rem 2)
ticket[I]: Y) pc[I]: ws [label incNxt&St] . ]===>
state 3, Sys: next: 0 serve: 0 (pc[p1]: ws) (pc[p2]: ws) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl serve: X (pc[I]: ws) ticket[I]: X => (serve: X ticket[I]: X)
pc[I]: cs [label wait] . ]===>
state 7, Sys: next: 0 serve: 0 (pc[p1]: cs) (pc[p2]: ws) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl serve: X pc[I]: cs => serve: ((X + 1) rem 2)
pc[I]: rs [label incServe]. ]===>
state 10, Sys: next: 0 serve: 1 (pc[p1]: rs) (pc[p2]: ws) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl next: Y (pc[I]: rs) ticket[I]: X => (next: ((Y + 1) rem 2)
ticket[I]: Y) pc[I]: ws [label incNxt&St] . ]===>
state 12, Sys: next: 1 serve: 1 (pc[p1]: ws) (pc[p2]: ws) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl serve: X (pc[I]: ws) ticket[I]: X => (serve: X ticket[I]: X)
pc[I]: cs [label wait] . ]===>
state 16, Sys: next: 1 serve: 1 (pc[p1]: ws) (pc[p2]: cs) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl serve: X pc[I]: cs => serve: ((X + 1) rem 2)
pc[I]: rs [label incServe] . ]===>
state 20, Sys: next: 1 serve: 0 (pc[p1]: ws) (pc[p2]: rs) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl serve: X (pc[I]: ws) ticket[I]: X => (serve: X ticket[I]: X)
pc[I]: cs [label wait] . ]===>
state 22, Sys: next: 1 serve: 0 (pc[p1]: cs) (pc[p2]: rs) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl serve: X pc[I]: cs => serve: ((X + 1) rem 2)
pc[I]: rs [label incServe] . ]===>
state 24, Sys: next: 1 serve: 1 (pc[p1]: rs) (pc[p2]: rs) (ticket[p1]: 0)
ticket[p2]: 1
===[ rl next: Y (pc[I]: rs) ticket[I]: X => (next: ((Y + 1) rem 2)
ticket[I]: Y) pc[I]: ws [label incNxt&St] . ]===>
state 26, Sys: next: 0 serve: 1 (pc[p1]: ws) (pc[p2]: rs) (ticket[p1]: 1)
ticket[p2]: 1
===[ rl serve: X (pc[I]: ws) ticket[I]: X => (serve: X ticket[I]: X)
```

```
pc[I]: cs [label wait] . ]===>
state 28, Sys: next: 0 serve: 1 (pc[p1]: cs) (pc[p2]: rs) (ticket[p1]: 1)
ticket[p2]: 1
===[ rl serve: X pc[I]: cs => serve: ((X + 1) rem 2)
 pc[I]: rs [label incServe] . ]===>
state 30, Sys: next: 0 serve: 0 (pc[p1]: rs) (pc[p2]: rs) (ticket[p1]: 1)
ticket[p2]: 1
```

- **LTL Model Checking**

The following LTL search command used for checking the Ticket.

```
red in TICKET-CHECK : modelCheck(init,lofree) .
```

We get the following result :
rewrites: 365 in 2ms cpu (8ms real) (169530 rewrites/second) result Bool: true

To use the Maude LTL model checker to check if Ticket enjoys the lockout freedom property, we need two kinds of atomic propositions **wait(P)** and **crit(P)**, where **P** is a process ID. Users are also supposed to specify a labeling function. For our purpose, we declare the three equations : **eq(pc[P] : ws) S | = want(P) = true., eq (pc[P] : cs) S | = crip(P) = true., and eq S | = PROP = false** [**owise**] ., where **P** is a Maude variable of process IDs, **S** is a Maude variable of atomic propositions. The three equations say a state **s** satisfies **want(P)** if and only if **(pc[P] : ws)** appears in **s** and **s** satisfies **crit(P)** if and only if **(pc[P] : cs)** appears in **s**. Then, users are supposed to specify LTL formulas to check. The lockout freedom property is expressed as **want(P)** $\leadsto$ **crit(P)**, where $\leadsto$ is the LTL leadsto operator. In Maude, the formula is specified as **eq lofree = (wait(p1)** $\mapsto$ **crit(p1))** /\ **(wait(p2)** $\mapsto$ **crit(p2))**, where the operator $\mapsto$ denotes the leadsto $\leadsto$. The model checking is conducted by reducing **modelCheck(init,lofree(p1))**, finding true, that means the ticket protocol likely enjoy the lockout freedom property.

3.2.4 Graphical Animations of Ticket

The Fig. 3.7 to 3.13 show that each state which found by Maude software for Ticket version. We used SMGA for drawing the thirteen pictures. These pictures make it possible to reorganize at which location of located each process is, what the value stored in next and serve and what the value stored in ticket1 and ticket2.

```
###keys
next serve pc[p1] pc[p2] ticket1 ticket2

###textDisplay
```

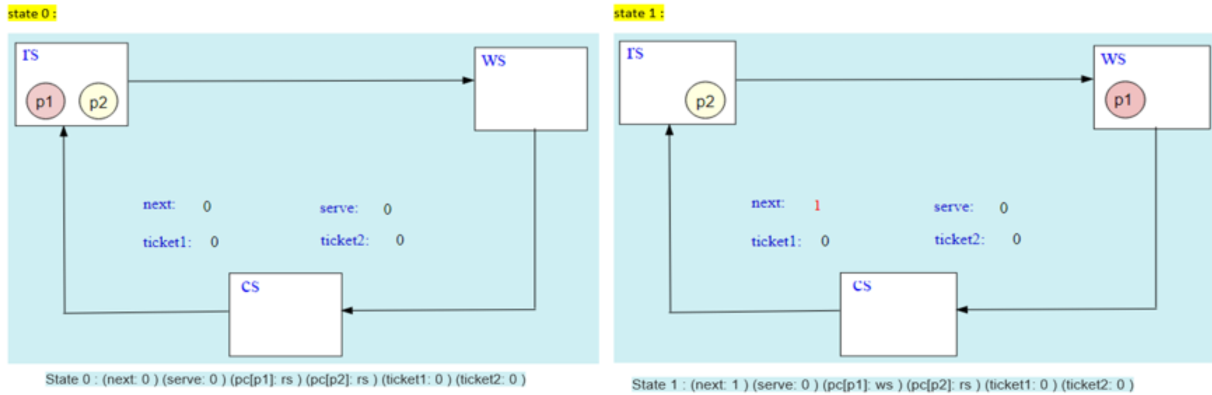

Figure 3.7: States 0 and 1 of Ticket

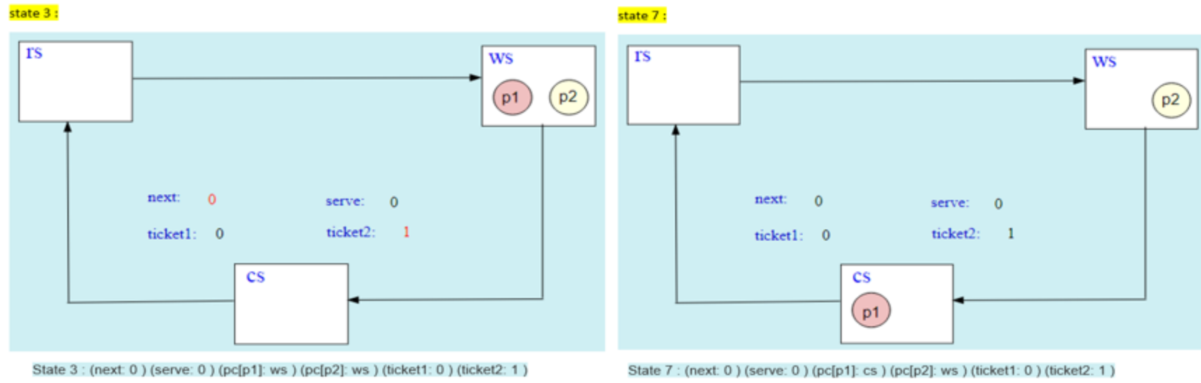

Figure 3.8: States 3 and 7 of Ticket

```
###states
(next: 0 serve: 0 (pc[p1]: rs) (pc[p2]: rs) (ticket1: 0) (ticket2: 0) ) ||
(next: 1 serve: 0 (pc[p1]: ws) (pc[p2]: rs) (ticket1: 0) (ticket2: 0) ) ||
(next: 0 serve: 0 (pc[p1]: ws) (pc[p2]: ws) (ticket1: 0) (ticket2: 1) ) ||
(next: 0 serve: 0 (pc[p1]: cs) (pc[p2]: ws) (ticket1: 0) (ticket2: 1) ) ||
(next: 0 serve: 1 (pc[p1]: rs) (pc[p2]: ws) (ticket1: 0) (ticket2: 1) ) ||
(next: 1 serve: 1 (pc[p1]: ws) (pc[p2]: ws) (ticket1: 0) (ticket2: 1) ) ||
(next: 1 serve: 1 (pc[p1]: ws) (pc[p2]: cs) (ticket1: 0) (ticket2: 1) ) ||
(next: 1 serve: 0 (pc[p1]: ws) (pc[p2]: rs) (ticket1: 0) (ticket2: 1) ) ||
(next: 1 serve: 0 (pc[p1]: cs) (pc[p2]: rs) (ticket1: 0) (ticket2: 1) ) ||
(next: 1 serve: 1 (pc[p1]: rs) (pc[p2]: rs) (ticket1: 0) (ticket2: 1) ) ||
(next: 0 serve: 1 (pc[p1]: ws) (pc[p2]: rs) (ticket1: 1) (ticket2: 1) ) ||
(next: 0 serve: 1 (pc[p1]: cs) (pc[p2]: rs) (ticket1: 1) (ticket2: 1) ) ||
(next: 0 serve: 0 (pc[p1]: rs) (pc[p2]: rs) (ticket1: 1) (ticket2: 1)
```

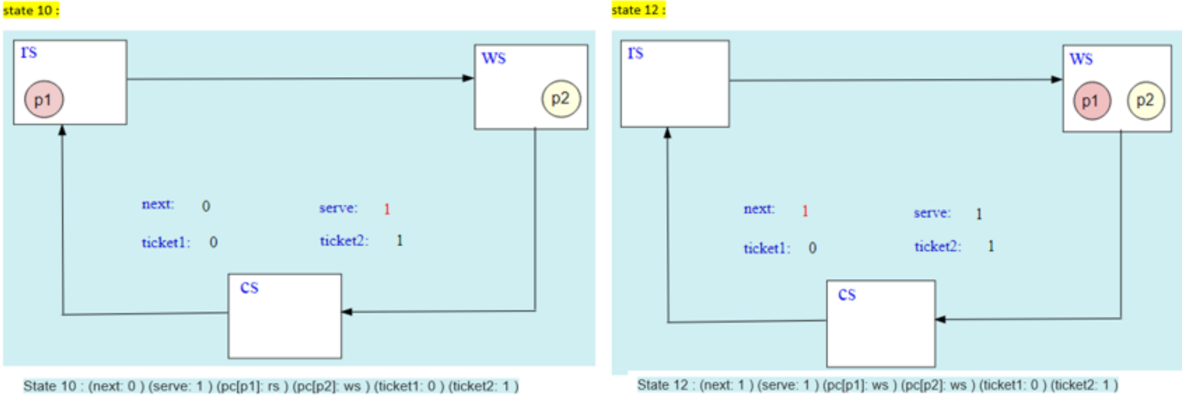

Figure 3.9: States 10 and 12 of Ticket

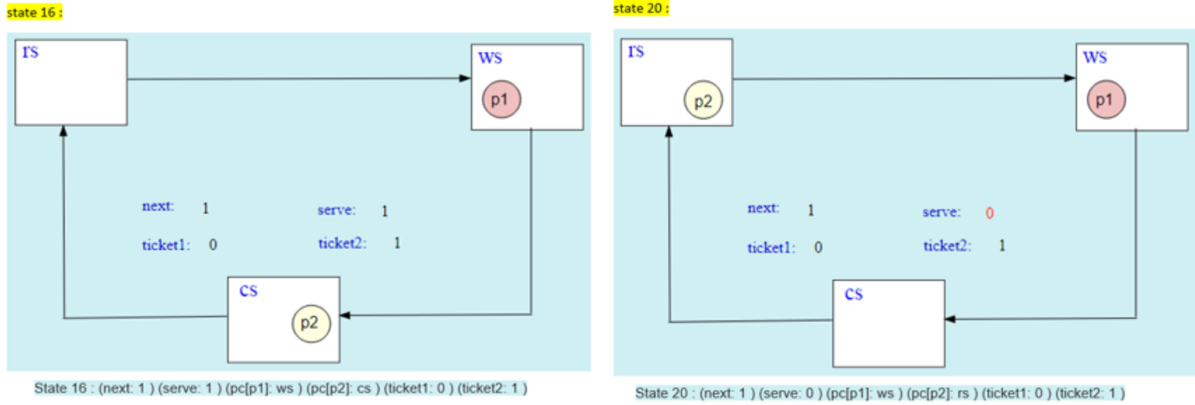

Figure 3.10: States 16 and 20 of Ticket

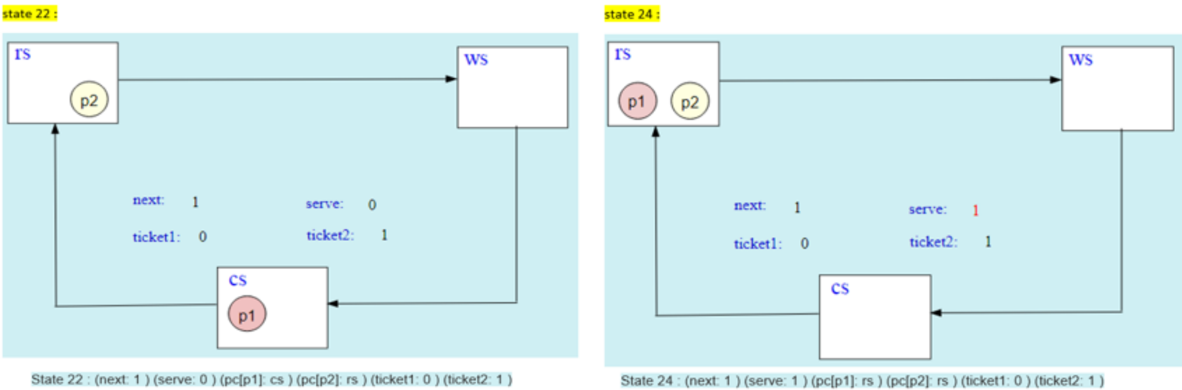

Figure 3.11: States 22 and 24 of Ticket

3.3 Non-deterministic version of Ticket Protocol

The ND-Ticket protocol is a mutual exclusion protocol based on issuing tickets to a critical section.next and serve are natural number variables share by all process. $ticket[i]$

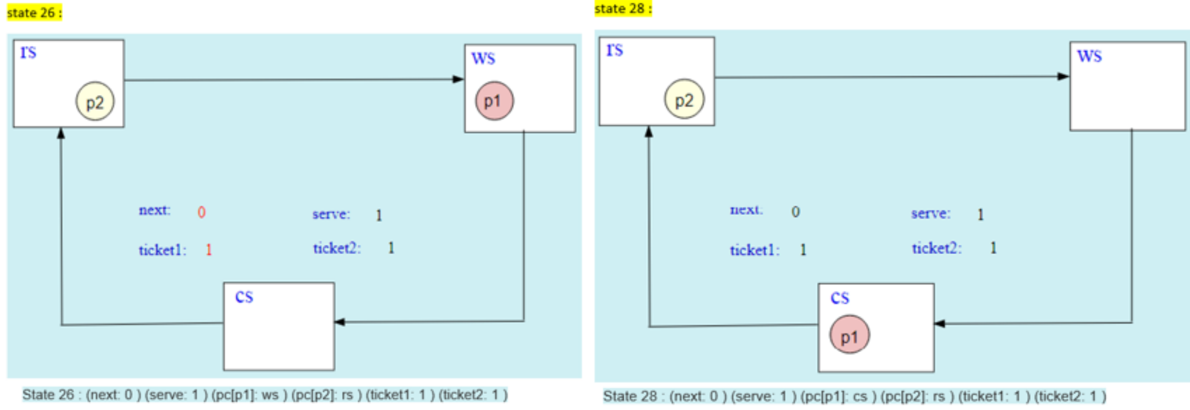

Figure 3.12: States 26 and 28 of Ticket

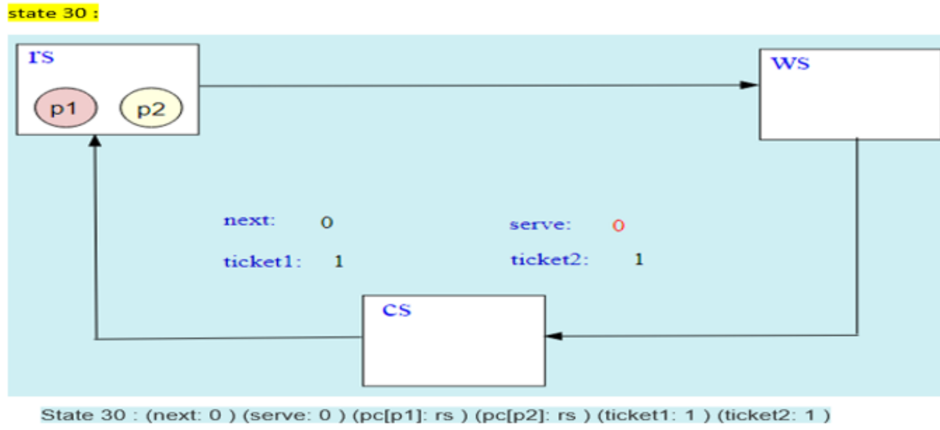

Figure 3.13: State 30 of Ticket

is a natural number variable that is local to process i. Where $Stm_1 \mid Stm_2$ is a non-deterministic choice statement s.t either Stm_1 or Stm_2 is non-deterministically chosen. This version is called ND-ticket.

ND-Ticket for a process i can be described as follows:

Loop: "Remainder Section"
rs: $ticket[i]$:= fetch&incmode($next$, N) | goto rs;
ws: **repeat until** $ticket$[i] = $serve$;
    "Critical Section"
cs: $serve$:= ($serve$+1) % N;

3.3.1 Specification of ND-Ticket

Here are two processes whose are denoted by p1 and p2. I, X and Y are Maude variables of process IDs, next and serve are process IDs, successively. $ticket[i]$ is a natural number variable that is local to process i. From some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times.

The state transitions of ND-Ticket are specified as the following three rewrite rules :

```
rl [incNxt&St] : (pc[I]: rs) (ticket[I]: X) (next: Y)
=> (pc[I]: ws) (ticket[I]: Y) (next: ((Y + 1) rem N)) .
rl [ds] : (pc[I]: rs)  => (pc[I]: rs) .
rl [wait] : (pc[I]: ws) (ticket[I]: X) (serve: X)
=> (pc[I]: cs) (ticket[I]: X) (serve: X) .
rl [incServe] : (pc[I]: cs) (serve: X) => (pc[I]: rs) (serve: ((X + 1) rem N)) .
```

incNxt&St, wait, incServe are the names of the three rewrite rules, respectively. The details description of three rewrite rules follows:

rule 1(incNxt&St) : a process I is located at rs, the content of ticket is X, the content of next is Y. After that a process I is located at ws, the content of ticket is Y, the content of next is increments Y remainder N. Here is rem N because processes N = 2.

rule 2(ds) : From some time on , a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times.

rule 3(wait) : a process I is located at ws, the content of ticket is X, the content of serve is X. After that a process I is located at cs, the content of ticket is X, the content of serve is X.

rule 4(incServe) : a process I is located at cs, the content of serve is X. After that a process I is located at rs, the content of serve is increments X remainder N. Here is the number of processes N = 2.

Fig. 3.14 shows the four state transition $incNxt\&St_I$, ds_I, $wait_I$ and $incServe_I$ respectively. After the transition from one state to another state we can indicate the process IDs I.

3.3.2 Specification of ND-Ticket as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, cs, ws} of locations.

Four kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(ticket[i] : X_N)$ - It says that the content of $ticket_i$ is X_N;
- $(serve : X)$ - It says that the content of $serve$ is X;

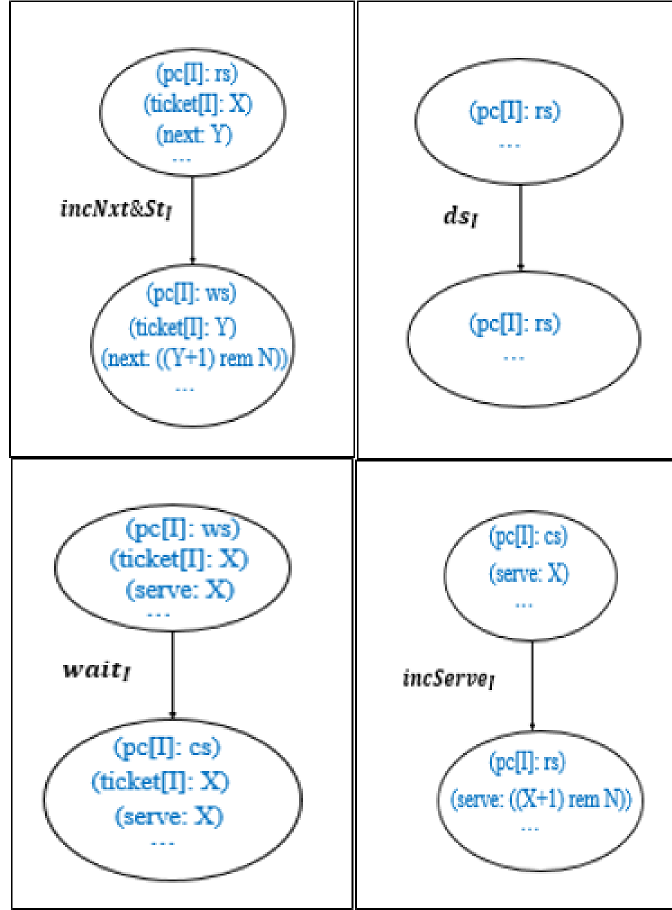

Figure 3.14: State Transition Diagram of ND-Ticket

- $(next : Y)$ - It says that the content of $next$ is Y;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, $l \in$ Loc and $X_N \in$ Pid Nat are values, tickett $\in$ Pid Nat and $X_1,\ldots,X_N$, X, Y $\in$ Nat . We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in ND-Ticket.

- Set of States, S = $\{$(pc[1] : L_1) ... (pc[N] : L_N)
  (ticket[1] : X_1)... (ticket[N] : X_N) (next : Y) (serve : X)
  | $L_1,\ldots,L_N \in$ Loc, $X_1,\ldots,X_N$, X, Y $\in$ Nat$\}$.
- Initial State, I = $\{$(pc[1] : rs)... (pc[N] : rs)
  (ticket[1] : 0)... (ticket[N] : 0) (next : 0) (serve : 0)$\}$.
- $T_{incNxt\&St}$ = $\{$((pc[1] : L_1)... (pc[I] : rs) ... (pc[N] : L_N)
  (ticket[1] : X_1) ... (ticket[I] : X_I) ... (ticket[N] : X_N) (next : Y) (serve : X),

(pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N) (serve : X)
(ticket[1] : X_1)...(ticket[I] : Y)...(ticket[N] : X_N) (next : ((Y + 1)rem N)))
| I ∈ {1, ..., N}, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, X, Y ∈ Nat}

- T_{ds} = {((pc[1] : L_1)...(pc[I] : rs) ...(pc[N] : L_N)
(next : Y) (serve : X) (ticket[1] : X_1)...(ticket[I] : X_I) ...(ticket[N] : X_N),
(pc[1] : L_1)...(pc[I] : rs)...(pc[N] : L_N)
(next : Y) (serve : X) (ticket[1] : X_1)...(ticket[I] : X_I) ...(ticket[N] : X_N))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, X, Y ∈ Nat}
- T_{wait} = {((pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N)
(ticket[1] : X_1)...(ticket[I] : X_I) ...(ticket[N] : X_N) (serve : X) (next : Y),
(pc[1] : L_1) ...(pc[I] : cs)...(pc[N] : L_N)
(ticket[1] : X_1)...(ticket[I] : X)...(ticket[N] : X_N) (serve : X) (next : Y))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, X, Y ∈ Nat}
- $T_{ncServe}$ = {((pc[1] : L_1)...(pc[I] : cs)...(pc[N] : L_N) (serve : X) (next : Y)
(ticket[1] : X_1)...(ticket[I] : X_I)...(ticket[N] : X_N),
(pc[1] : L_1)...(pc[I] : rs)...(pc[N] : L_N) (serve: ((X + 1)rem N))
(next : Y) (ticket[1] : X_1)...(ticket[I] : X_I)...(ticket[N] : X_N))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N, X, Y ∈ Nat}

3.3.3 Model Checking of ND-Ticket

The following search command used for checking the ND-Ticket.

```
search [1] in ND-TICKET : init =>* (pc[p1]: cs) (pc[p2]: cs) S .
```

Maude finds no solution meaning ND-Ticket likely to enjoy the mutual exclusion property.

Where ND-TICKET is the module in which ND-Ticket is specified and S is a maude variable of state fragments.

```
red in ND-Ticket-CHECK : modelCheck(init,lofree) .
```

A counter example is found.

The LTL found the following counter example:

```
counterexample({next: 0 serve: 0 (pc[p1]: rs) (pc[p2]:rs) (ticket[p1]: 0)
ticket[p2]: 0,'incNxt&St}, {next: 1 serve: 0 (pc[p1]:ws) (pc[p2]: rs)
(ticket[p1]: 0) ticket[p2]: 0,'ds})
```

Althogh p1 is ready to entering the critical section, p2 is always chosen and the rewrite rule ds for p2 is taken, which is not fair for p1.

Chapter 4

Anderson Mutual Exclusion Protocol

4.1 FAnderson: A Flawed Version of Anderson Protocol

The FAnderson is an array-based queuing mutual exclusion protocol. This is a wrong version of Anderson protocol. It might be viewed as a change of Fticket calculation. next and array are number variables share by the N processes. $place[i]$ is a natural number variable that is local to process i.

FAnderson for a process i can be described as follows:

Loop: "Remainder Section"
rs: $place[i]$:= $next$;
l1: $next$:= ($next$+1) % N;
ws: **repeat until** array[place[i]];
    "Critical Section"
cs: array[place[i]], array[(place[i]+1) % N:= false, true;

Here are four locations rs (remainder section), l1 (label 1), ws (waiting section), cs (critical section). We suppose that there are N processes. For each process i, there are two local variables: $place_i$ whose value is in {0, 1,. . . ,N - 1} to process whose ID is i in {0, 1,. . . , N - 1}, initially $place_i$ = 0. Two global variables shared by the N processes: *next* whose value is in {0, 1,. . . , N - 1} and *array* whose value is in {0, 1,. . . , N - 1}. Initially the value of *next* = 0, *array[0]* = true and *array[i]* = false. next represents the next to the critical section that is to be issued to a process, while array represents the Boolean array whose size is N *array[0]*, *array[1]*,..,*array[N - 1]* and *array[j]* = 0 for each j. The place of the procedure is set to the next. At that point, the next is computed expanded first and discover update when isolated by the number of procedures. In waiting section up area, it will rehash until the point when the cluster of the procedure put. In the basic segment, the variety of process put is false and an array of the expanded process put partitioned the quantity of process genuine.

4.1.1 Specification of FAnderson in Maude and State Transition Diagrams

Here are two processes whose are denoted by p1 and p2. I, X and Y, X1 are Maude variables of process IDs and natural number and B1, B2 are boolean successively. $place[i]$ is a natural number variable that is local to process i.

The state transitions of FAnderson are specified as the following four rewrite rules:

```
rl [setPlace] : (pc[I]: rs) (place[I]: X) (next: Y)
=> (pc[I]: l1) (place[I]: Y) (next: Y) .
rl [incNxt] : (pc[I]: l1) (next: Y)
=> (pc[I]: ws) (next: ((Y + 1) rem N)) .
rl [wait] : (pc[I]: ws) (place[I]: X) (array[X]: true)
=> (pc[I]: cs) (place[I]: X) (array[X]: true) .
crl [chArray] : (pc[I]: cs) (place[I]: X) (array[X]: B1) (array[X1]: B2)
=> (pc[I]: rs) (place[I]: X) (array[X]: false) (array[X1]: true)
if X1 = (X + 1) rem N .
```

setPlace, incNxt, wait, chArray are the names of the four rewrite rules, respectively. The details description of four rewrite rule follows:

rule 1(setPLace) : a process I is located at rs, the content of place is X, the content of next is Y. After that a process I is located at l1, the content of place is Y, the content of next is Y.

rule 2(incNxt) : a process I is located at l1, the content of next is Y. After that a process I is located at ws, the content of next is increments Y remainder N. Here is N = 2.

rule 3(wait) : a process I is located at ws, the content of place is X, the content of array is true. After that a process I is located at cs, the content of place is X, the content of array is true.

Conditional rule 4(chArray) : a process I is located at cs, the content of place is X and the contains of array are B1 and B2 . After that a process I is located at rs, the content of place is X, the contains of array are false and true, increments X remainder N. Here is N = 2.

Fig. 4.1 shows the four state transition $setPlace_I$, $incNxt_I$, $wait_I$ and $chArray_I$ respectively. After the transition from one state to another state we can indicate the process IDs I.

4.1.2 Specification of FAnderson as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, l1, cs, ws} of locations.

Five kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(array[0] : true)$ - It says that the content of $array[0]$ is true;

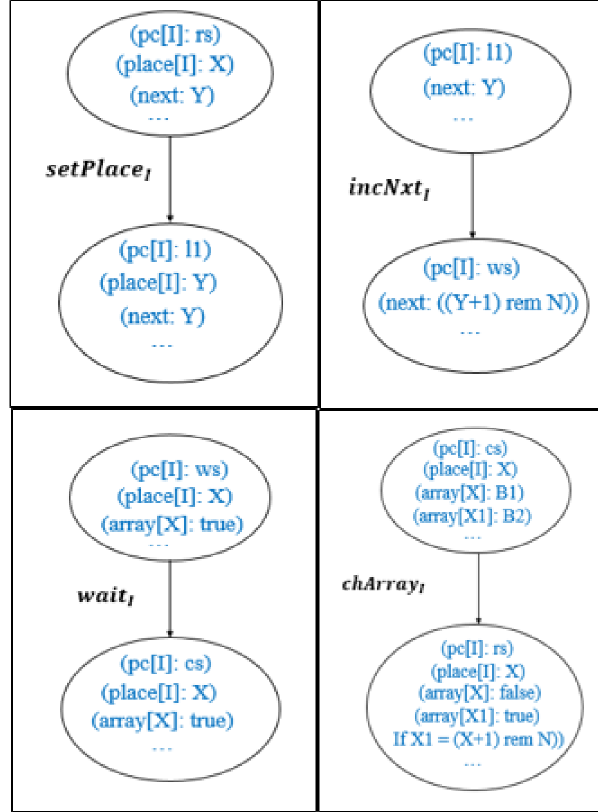

Figure 4.1: State Transition Diagram of FAnderson

- $(array[1] : false)$ - It says that the content of $array[1]$ is false;
- $(place : X)$ - It says that the content of $place$ is X;
- $(next : Y)$ - It says that the content of $next$ is Y;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, $l \in$ Loc and $X_N \in$ Pid Nat are values, X $\in$ Pid Nat and X, Y, B1, B2 $\in$ Nat Bool. We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in FAnderson.

- Set of States, S = {(pc[1] : L_1) . . . (pc[N] : L_N) (next : Y)
  (array[X] : B1) (array[X1] : B2)
  (place[1] : X_1) . . . (place[N] : X_N)
  | $L_1, \ldots, L_N \in$ Loc, $X_1, \ldots, X_N$, X, Y $\in$ Nat, B1, B2 $\in$ Bool}.
- Initial State, I = {(pc[1] : rs) . . . (pc[N] : rs)
  (place[1] : 0) . . . (place[N] : 0) (next : 0) (array[0] : true) (array[1] : false)}.

- $T_{SetPlace}$ = {((pc[1] : L_1) ... (pc[I] : rs) ... (pc[N] : L_N)
(place[1] : X_1) ... (place[I] : X_I) ... (place[N] : X_N)
(next : Y) (array[X] : B1) (array[X1] : B2),
(pc[1] : L_1) ... (pc[I] : l_1) ... (pc[N] : L_N)
(place[1] : X_1) ... (place[I] : Y) ... (place[N] : X_N)
(next : Y)) (array[X] : B1) (array[X1] : B2))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc, X_1,..., X_N , X, Y ∈ Nat, B1, B2 ∈ Bool}

- T_{incNxt} = {((pc[1] : L_1) ... (pc[I] : l_1) ... (pc[N] : L_N)
(place[1] : X_1) ... (place[I] : Y) ... (place[N] : X_N)
(next : Y) (array[X] : B1) (array[X1] : B2),
(pc[1] : L_1) ... (pc[I] : ws) ... (pc[N] : L_N) (next : ((Y + 1)rem N))
(place[1] : X_1) ... (place[I] : Y) ... (place[N] : X_N)
(array[X] : B1) (array[X1] : B2))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc,
X_1,..., X_N, X, Y ∈ Nat, B1, B2 ∈ Bool}

- T_{wait} = {((pc[1] : L_1) ... (pc[I] : ws) ... (pc[N] : L_N)
(place[1] : X_1) ... (place[I] : X_I) ... (place[N] : X_N) (array[X] : true),
(pc[1] : L_1) ... (pc[I] : cs) ... (pc[N] : L_N) (array[X] : true)
(place[1] : X_1) ... (place[I] : X) ... (place[N] : X_N))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc,
X_1,..., X_N, X, Y ∈ Nat, B1, B2 ∈ Bool}

- $T_{ChArray}$ = {((pc[1] : L_1) ... (pc[I] : cs) ... (pc[N] : L_N) (array[X] : B1) (array[X1] : B2)
(place[1] : X_1) ... (place[I] : X_I) ... (place[N] : X_N),
(pc[1] : L_1) ... (pc[I] : rs) ... (pc[N] : L_N) (array[X] : false) (array[X1] : true) if X1 = (X +1)rem N
(place[1] : X_1) ... (place[I] : X) ... (place[N] : X_N))
| I ∈ {1, ..., N }, L_1 ,..., L_N ∈ Loc,
X_1,..., X_N, X, Y ∈ Nat, B1, B2 ∈ Bool}

4.1.3 Model Checking of FAnderson

The following search command used for checking the FAnderson.

```
search [1] in FAnderson : init =>* (pc[p1]: cs) (pc[p2]: cs) S .
```

Maude finds a solution meaning FAnderson does not enjoy the property. Here is also used following path search command : show path 28 . This command shows each states of the FAnderson version by Maude.

Where FAnderson is the module in which FAnderson is specified and S is a Maude variable of state fragments. The search command finds the following counterexample:

```
state 0, Sys: next: 0 (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 0)
(place[p2]: 0) (array[0]: true) array[1]: false
===[ rl next: Y (pc[I]: rs) place[I]: X => (next: Y place[I]: Y)
pc[I]: l1 [label setPlace] . ]===>
state 1, Sys: next: 0 (pc[p1]: l1) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 0)
(
array[0]: true) array[1]: false
===[ rl next: Y (pc[I]: rs) place[I]: X => (next: Y place[I]: Y)
pc[I]: l1 [label setPlace] . ]===>
state 3, Sys: next: 0 (pc[p1]: l1) (pc[p2]: l1) (place[p1]: 0) (place[p2]: 0)
 (array[0]: true) array[1]: false
===[ rl next: Y pc[I]: l1 => next: ((Y + 1) rem 2) pc[I]: ws [label incNxt] .
]===>
state 6, Sys: next: 1 (pc[p1]: ws) (pc[p2]: l1) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) array[1]: false
===[ rl next: Y pc[I]: l1 => next: ((Y + 1) rem 2) pc[I]: ws [label incNxt] .
]===>
state 12, Sys: next: 0 (pc[p1]: ws) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) array[1]: false
===[ rl (pc[I]: ws) (place[I]: X) array[X]: true => ((place[I]: X) array[X]:
true) pc[I]: cs [label wait] . ]===>
state 20, Sys: next: 0 (pc[p1]: cs) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) array[1]: false
===[ rl (pc[I]: ws) (place[I]: X) array[X]: true => ((place[I]: X) array[X]:
true) pc[I]: cs [label wait] . ]===>
state 28, Sys: next: 0 (pc[p1]: cs) (pc[p2]: cs) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) array[1]: false
```

4.1.4 Graphical Animations of FAnderson Counterexamples

Maude can generate counterexample without any type of difficulties, which non-experts cannot do it. The Fig. 4.2 to 4.5 show that counterexample which found by Maude software for FAnderson version. We used SMGA for drawing the seven pictures. These

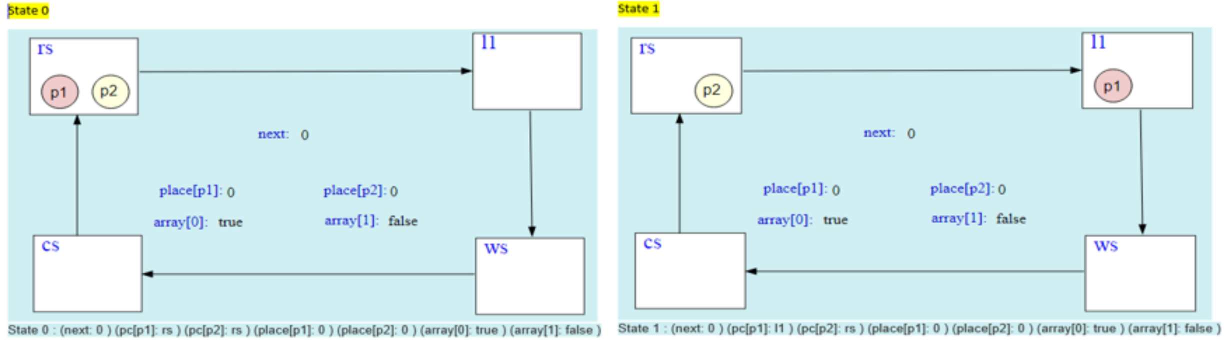

Figure 4.2: Counterexample for FAnderson of states 0 and 1

pictures make it possible to reorganize at which location each process is, what the value stored in next, what the value stored in place1 and place2, array a Boolean array whose size is array[0] and array[1]. Here is only next: 1 in the state 6. But for the others states natural numbers variables content of next, place1, place2 values are zero and array0 and array1 contain true, false respectively.

```
###keys
next pc[p1] pc[p2] place[p1] place[p2] array[0] array[1]


###textDisplay

###states
(next: 0  (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: l1) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: l1) (pc[p2]: l1) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 1  (pc[p1]: ws) (pc[p2]: l1) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: ws) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: cs) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: cs) (pc[p2]: cs) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false)
```

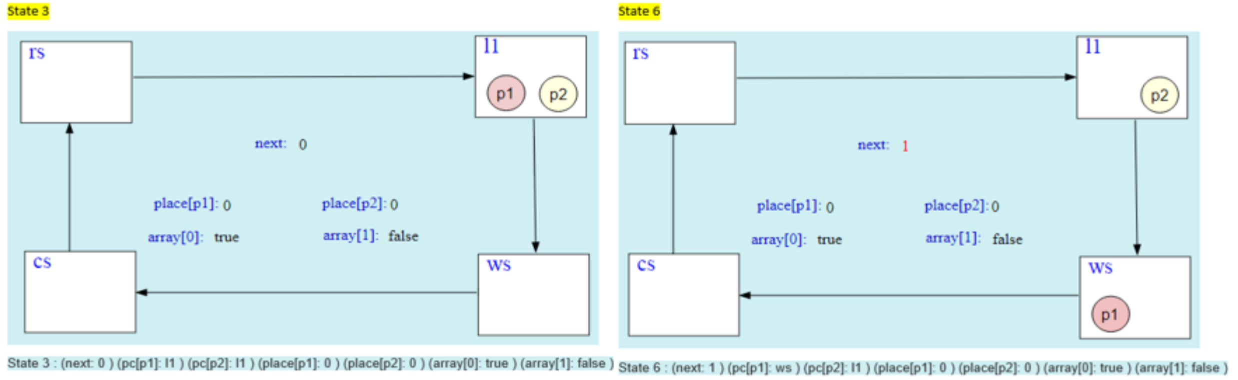

Figure 4.3: Counterexample for FAnderson of states 3 and 6

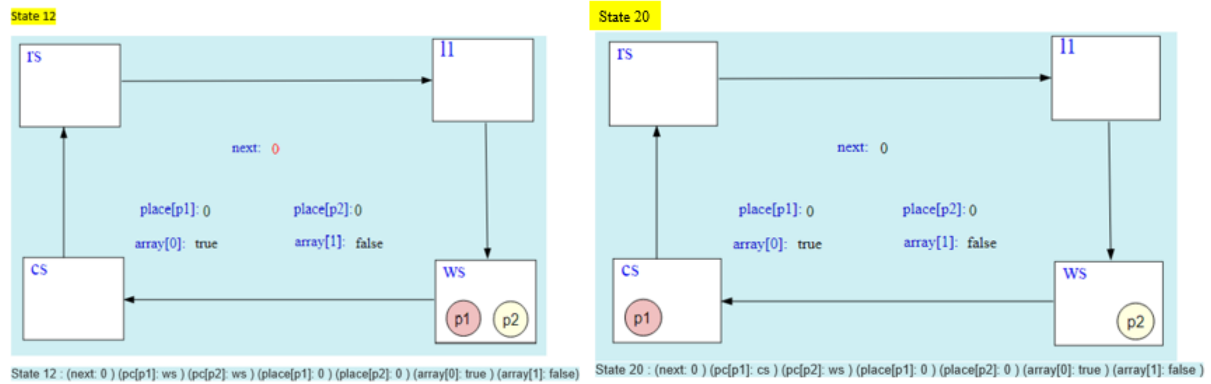

Figure 4.4: Counterexample for FAnderson of states 12 and 20

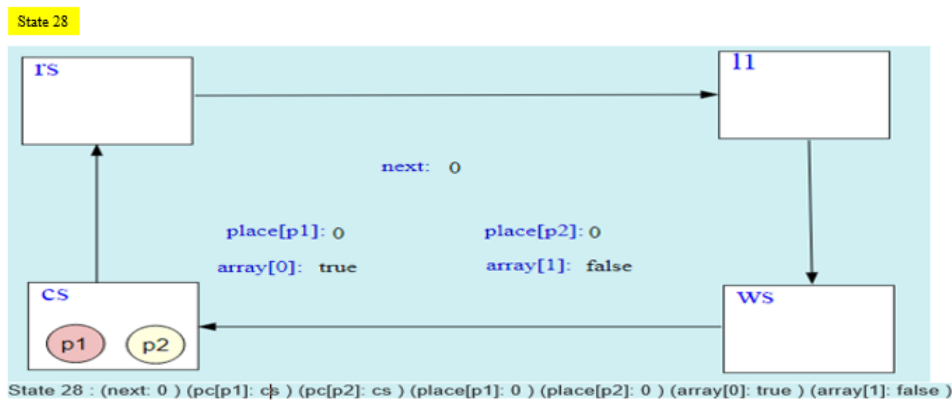

Figure 4.5: Counterexample for FAnderson of state 28

4.2 Anderson Protocol

The Anderson is an array-based queuing mutual exclusion protocol. This is a right version of Anderson protocol. It might be viewed as a change of Ticket calculation. next and

array are number variables share by the N processes. $place[i]$ is a natural number variable that is local to process i. In the Anderson protocol , each process is waiting on a different location, in a different cache line, if some process is in critical section.

Anderson for a process i can be described as follows:

Loop: "Remainder Section"
rs: $place[i]$:= featch&incmode($next$,N) ;
ws: **repeat until** $a$rray[place[i]];
"Critical Section"
cs: $a$rray[$p$lace[i]], $a$rray[($p$lace[i]+1) % N:= false, true;

There are used fetch&incmode atomic operation for implement the protocol. This is atomically reads a memory location, increments the value modulo N. Here are four locations rs (remainder section), l1 (label 1), ws (waiting section), cs (critical section). We suppose that there are N processes. For each process i, there are two local variables: $place_i$ whose value is in $\{0, 1, \ldots, N - 1\}$ to process whose ID is i in $\{0, 1, \ldots, N - 1\}$, initially $place_i = 0$. Two global variables shared by the N processes: *next* whose value is in $\{0, 1, \ldots, N - 1\}$ and *array* whose value is in $\{0, 1, \ldots, N - 1\}$. Initially the value of *next* = 0, *array[0]* = true and *array[i]* = false. next represents the next to the critical section that is to be issued to a process, while array represents the Boolean array whose size is N *array[0]*, *array[1]*, ...,*array[N - 1]* and *array[j]* = 0 for each j. When a process i tries to enter the critical section, it indivisibly copies into it local variable *place* and increments *next* remainder N using $fetch\&incmode$.

For the variable x and a constant c whose type of natural numbers
fetch&incmode(x,n) conducts the following atomically (or indivisibly):
t := x; x:= (x + 1)%n; **return** t

4.2.1 Specification of Anderson in Maude and State Transition Diagrams

Here are two processes whose are denoted by p1 and p2. I, X and Y are Maude variables of process IDs, next and serve are process IDs, successively. $ticket[i]$ is a natural number variable that is local to process i.

The state transitions of Anderson are specified as the following three rewrite rules :

```
 rl [setPlace] : (pc[I]: rs) (next: X) (place[I]: Y)
 => (pc[I]: ws) (next: ((X + 1) rem N))  (place[I]: X) .
rl [wait] : (pc[I]: ws) (place[I]: X) (array[X]: true)
 => (pc[I]: cs) (place[I]: X) (array[X]: true) .
crl [chArray] : (pc[I]: cs) (place[I]: X) (array[X]: B1) (array[X1]: B2)
=> (pc[I]: rs) (place[I]: X) (array[X]: false) (array[X1]: true)
if X1 = (X + 1) rem N .
```

setPlace, wait, chArray are the names of the three rewrite rules, respectively. The details description of three rewrite rule for Anderson follows:

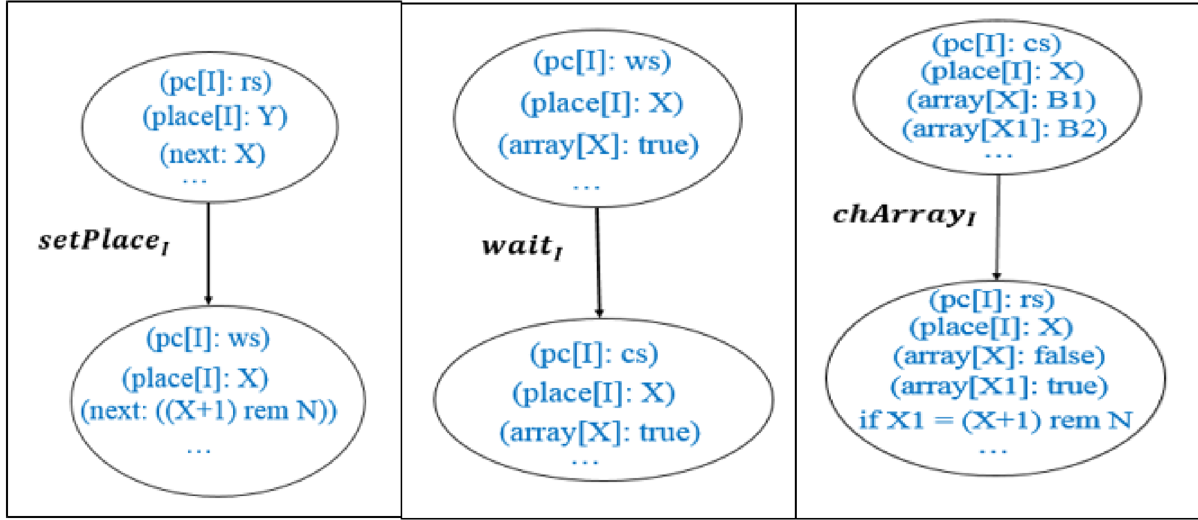

Figure 4.6: State transition diagram of Anderson

rule 1(setPLace) : a process I is located at rs, the content of place is Y, the content of next is X. After that a process I is located at ws, the content of place is Y, the content of next is increments X remainder N. Here is number of processes N = 2.

rule 2(wait) : a process I is located at ws, the content of place is X, the content of array is true. After that a process I is located at cs, the content of place is X, the content of array is true.

conditional rule 3(chArray) : a process I is located at cs, the content of place is X and the contains of array are B1 and B2 . After that a process I is located at rs, the content of place is X, the contains of array are false and true, increments X remainder N. Here is number of processes N = 2.

Fig. 4.6 shows the three state transition $setPlace_I$, $wait_I$ and $chArray_I$ respectively. After the transition from one state to another state we can indicate the process IDs I.

4.2.2 Specification of Anderson as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, cs, ws} of locations.

Five kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(array[0] : true)$ - It says that the content of $array[0]$ is true;
- $(array[1] : false)$ - It says that the content of $array[1]$ is false;
- $(place : Y)$ - It says that the content of $place$ is Y;

- $(next : X)$ - It says that the content of $next$ is X;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, $l \in$ Loc and $X_N \in$ Pid Nat are values, next $\in$ Pid Nat and array $\in$ Nat Bool . We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in Anderson.

- Set of States, S = {(pc[1] : L_1) ... (pc[N] : L_N) (next : X)

  (array[X] : B1) (array[X1] : B2)

  (place[1] : X_1) ... (place[N] : X_N)

  | $L_1, \ldots, L_N \in$ Loc, $X_1, \ldots, X_N$, X, Y $\in$ Nat, B1, B2 $\in$ Bool}.

- Initial State, I = {(pc[1] : rs) ... (pc[N] : rs)

  (place[1] : 0) ... (place[N] : 0) (next : 0) (array[0] : true) (array[1] : false)}.

- $T_{SetPlace}$ = {((pc[1] : L_1) ... (pc[I] : rs) ... (pc[N] : L_N)

  (place[1] : X_1) ... (place[I] : Y_I) ... (place[N] : X_N) (next : X) (array[X] : B1) (array[X1] : B2),

  (pc[1] : L_1) ... (pc[I] : ws) ... (pc[N] : L_N)

  (place[1] : X_1) ... (place[I] : X) ... (place[N] : X_N) (next : ((X + 1) rem N))

  (array[X] : B1) (array[X1] : B2))

  | I $\in$ {1, ..., N }, L_1 ,..., $L_N \in$ Loc , X_1,..., X_N, X, Y $\in$ Nat, B1, B2 $\in$ Bool}

- T_{wait} = {((pc[1] : L_1) ... (pc[I] : ws) ... (pc[N] : L_N)

  (place[1] : X_1). ... (place[I] : X_I)... (place[N] : X_N) (array[X] : true) (next : X),

  (pc[1] : L_1) ... (pc[I] : cs) ... (pc[N] : L_N) (array[X] : true)

  (place[1] : X_1) ... (place[I] : X)... (place[N] : X_N)) (next : X))

  | I $\in$ {1, ..., N }, L_1 ,..., $L_N \in$ Loc , X_1,..., X_N, X, Y $\in$ Nat, B1, B2 $\in$ Bool}

- $T_{ChArray}$ = {((pc[1] : L_1) ... (pc[I] : cs) ... (pc[N] : L_N) (array[X] : B1) (array[X1] : B2) (place[1] : X_1) ... (place[I] : X_I) ... (place[N] : X_N) (next : X),

  (pc[1] : L_1) ... (pc[I] : rs) ... (pc[N] : L_N) (array[X] : false) (array[X1] : true) if X1 = (X + 1) rem N

  (place[1] : X_1) ... (place[I] : X)... (place[N] : X_N)) (next : X))

  | I $\in$ {1, ..., N }, L_1 ,..., $L_N \in$ Loc , X_1,..., X_N, X, Y $\in$ Nat, B1, B2 $\in$ Bool}

4.2.3 Model Checking of Anderson

- **Maude Search Command**

The following search command used for checking the Anderson:

```
search [1] in Anderson: init =>* (pc[p1]: cs) (pc[p2]: cs) S .
```

Maude finds no solution meaning Anderson likely to enjoy the mutex property. Here is also used following path search command: show path 30 . This command shows each states of the Anderson version by Maude.

Where Anderson is the module in which Anderson is specified and S is a maude variable of state fragments. The search command finds the follows:

```
state 0, Sys: next: 0 (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 0)
(place[p2]: 0) (array[0]: true) array[1]: false
===[ rl next: X (pc[I]: rs) place[I]: Y => (next: ((X + 1) rem 2)
place[I]: X) pc[I]: ws [label setPlace] . ]===>
state 1, Sys: next: 1 (pc[p1]: ws) (pc[p2]: rs) (place[p1]: 0)
(place[p2]: 0) (array[0]: true) array[1]: false
===[ rl next: X (pc[I]: rs) place[I]: Y => (next: ((X + 1) rem 2)
place[I]: X) pc[I]: ws [label setPlace] . ]===>
state 3, Sys: next: 0 (pc[p1]: ws) (pc[p2]: ws) (place[p1]: 0)
(place[p2]: 1) (array[0]: true) array[1]: false
===[ rl (pc[I]: ws) (place[I]: X) array[X]: true => ((place[I]: X)
array[X]:true) pc[I]: cs [label wait] . ]===>
state 7, Sys: next: 0 (pc[p1]: cs) (pc[p2]: ws) (place[p1]: 0)
(place[p2]: 1) (array[0]: true) array[1]: false
===[ crl (pc[I]: cs) (place[I]: X) (array[X]: B1) array[X1]: B2 =>
(((array[X]: false) array[X1]: true) place[I]: X) pc[I]: rs if X1
= (X + 1) rem 2 [label chArray] . ]===>
state 10, Sys: next: 0 (pc[p1]: rs) (pc[p2]: ws) (place[p1]: 0)
(place[p2]: 1) (array[0]: false) array[1]: true
===[ rl next: X (pc[I]: rs) place[I]: Y => (next: ((X + 1) rem 2)
place[I]: X) pc[I]: ws [label setPlace] . ]===>
state 12, Sys: next: 1 (pc[p1]: ws) (pc[p2]: ws) (place[p1]: 0)
(place[p2]: 1) (array[0]: false) array[1]: true
===[ rl (pc[I]: ws) (place[I]: X) array[X]: true => ((place[I]: X)
array[X]: true) pc[I]: cs [label wait] . ]===>
state 16, Sys: next: 1 (pc[p1]: ws) (pc[p2]: cs) (place[p1]: 0)
(place[p2]: 1)(array[0]: false) array[1]: true
===[ crl (pc[I]: cs) (place[I]: X) (array[X]: B1) array[X1]: B2 =>
(((array[X]: false) array[X1]: true) place[I]: X) pc[I]: rs if X1
= (X + 1) rem 2 [label chArray] . ]===>
```

```
state 20, Sys: next: 1 (pc[p1]: ws) (pc[p2]: rs) (place[p1]: 0)
(place[p2]: 1) (array[0]: true) array[1]: false
===[ rl (pc[I]: ws) (place[I]: X) array[X]: true => ((place[I]: X)
array[X]: true) pc[I]: cs [label wait] . ]===>
state 22, Sys: next: 1 (pc[p1]: cs) (pc[p2]: rs) (place[p1]: 0)
 (place[p2]: 1) (array[0]: true) array[1]: false
===[ crl (pc[I]: cs) (place[I]: X) (array[X]: B1) array[X1]: B2 =>
(((array[X]: false) array[X1]: true) place[I]: X) pc[I]: rs if X1
= (X + 1) rem 2 [label chArray] . ]===>
state 24, Sys: next: 1 (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 0)
 (place[p2]: 1) (array[0]: false) array[1]: true
===[ rl next: X (pc[I]: rs) place[I]: Y => (next: ((X + 1) rem 2)
place[I]: X) pc[I]: ws [label setPlace] . ]===>
state 26, Sys: next: 0 (pc[p1]: ws) (pc[p2]: rs) (place[p1]: 1)
 (place[p2]: 1) (array[0]: false) array[1]: true
===[ rl (pc[I]: ws) (place[I]: X) array[X]: true => ((place[I]: X)
array[X]: true) pc[I]: cs [label wait] . ]===>
state 28, Sys: next: 0 (pc[p1]: cs) (pc[p2]: rs) (place[p1]: 1)
(place[p2]: 1) (array[0]: false) array[1]: true
===[ crl (pc[I]: cs) (place[I]: X) (array[X]: B1) array[X1]: B2 =>
(((array[X]: false) array[X1]: true) place[I]: X) pc[I]: rs if X1 =
(X + 1) rem 2 [label chArray] . ]===>
state 30, Sys: next: 0 (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 1)
(place[p2]: 1) (array[0]: true) array[1]: false
```

- **LTL Model Checking**

The following LTL search command used for checking the Ticket.

```
red in Anderson-CHECK : modelCheck(init,lofree) .
```

We get the following result:
rewrites: 323 in 1ms cpu (2ms real) (162067 rewrites/second) result Bool: true

To use the Maude LTL model checker to check if Anderson enjoys the lockout freedom property, we need two kinds of atomic propositions **wait(P)** and **crit(P)**, where **P** is a process ID. Users are also supposed to specify a labeling function. For our purpose, we declare the three equations : **eq(pc[P] : ws) S | = want(P) = true., eq (pc[P] : cs) S | = crip(P) = true., and eq S | = PROP = false** [**owise**] ., where **P** is a Maude variable of process IDs, **S** is a Maude variable of atomic propositions. The three equations say a state **s** satisfies **want(P)** if and only if **(pc[P] : ws)** appears in **s** and **s** satisfies **crit(P)** if and only if **(pc[P] : cs)** appears in **s**. Then, users are supposed to specify LTL formulas to check. The lockout freedom property is expressed as **want(P)** $\leadsto$ **crit(P)**, where $\leadsto$ is the LTL leadsto operator. In Maude, the formula

is specified as **eq lpfree = (wait(p1) $\mapsto$ crit(p1)) /\ (wait(p2) $\mapsto$ crit(p2))**, where the operator $\mapsto$ denotes the leadsto $\leadsto$. The model checking is conducted by reducing **modelCheck(init,lofree(p1))**, finding true, that means the Anderson protocol likely enjoy the lockout freedom property.

4.2.4 Graphical Animations of Anderson

The Fig. 4.7 to 4.13 show that the each state which found by Maude software for Anderson correct version. We used SMGA for drawing the thirteen pictures. These pictures make it possible to reorganize at which location of located the each process is, what the value stored in next and array[0], array[1] and what the value stored in place1 and place2.

```
###keys
next pc[p1] pc[p2] place[p1] place[p2] array[0] array[1]

###textDisplay

###states
(next: 0  (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 1  (pc[p1]: ws) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 0)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: ws) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 1)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: cs) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 1)
(array[0]: true) (array[1]: false) ) ||
(next: 0  (pc[p1]: rs) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 1)
(array[0]: false)(array[1]: true) ) ||
(next: 1  (pc[p1]: ws) (pc[p2]: ws) (place[p1]: 0) (place[p2]: 1)
 (array[0]: false) (array[1]: true) ) ||
(next: 1  (pc[p1]: ws) (pc[p2]: cs) (place[p1]: 0) (place[p2]: 1)
(array[0]: false) (array[1]: true) ) ||
(next: 1  (pc[p1]: ws) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 1)
 (array[0]: true) (array[1]: false) ) ||
(next: 1  (pc[p1]: cs) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 1)
 (array[0]: true) (array[1]: false) ) ||
(next: 1  (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 1)
(array[0]: false) (array[1]: true) ) ||
(next: 0  (pc[p1]: ws) (pc[p2]: rs) (place[p1]: 1) (place[p2]: 1)
 (array[0]: false) (array[1]: true) ) ||
(next: 0  (pc[p1]: cs) (pc[p2]: rs) (place[p1]: 1) (place[p2]: 1)
(array[0]: false)(array[1]: true) ) ||
(next: 0  (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 1) (place[p2]: 1)
 (array[0]: true) (array[1]: false)
```

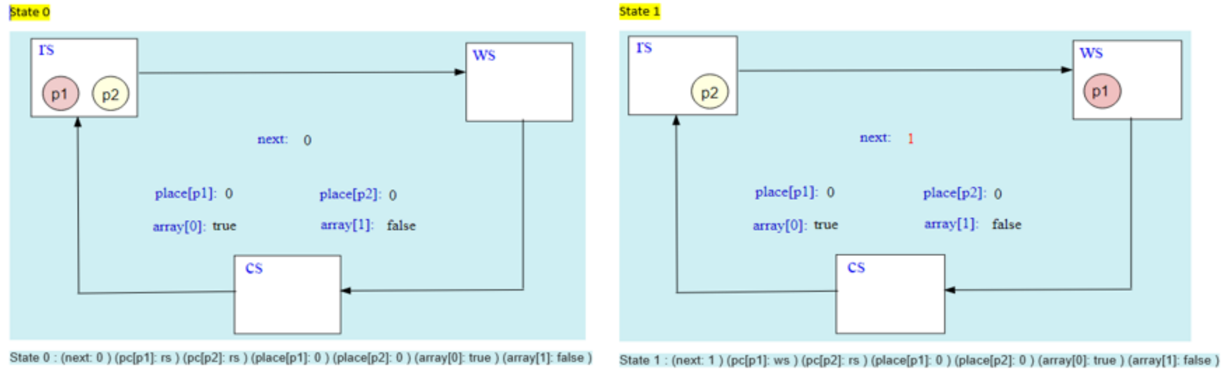

Figure 4.7: States 0 and 1 of Anderson

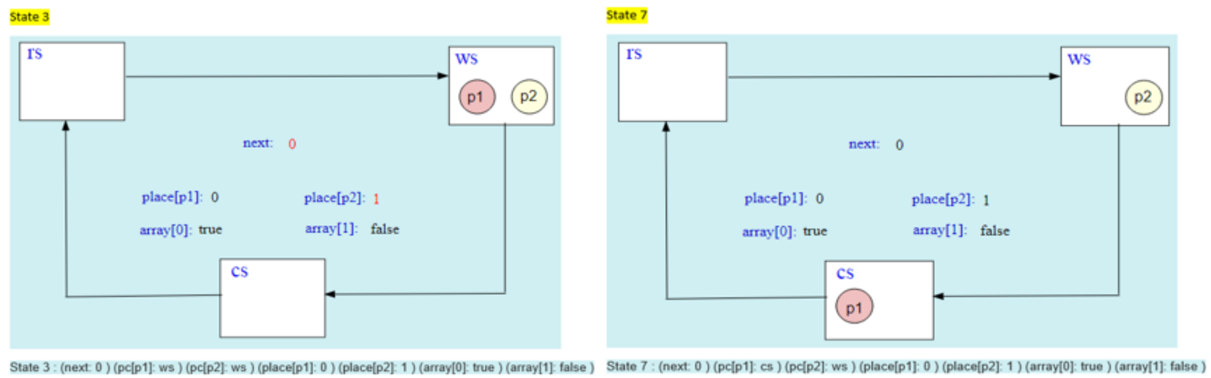

Figure 4.8: States 3 and 7 of Anderson

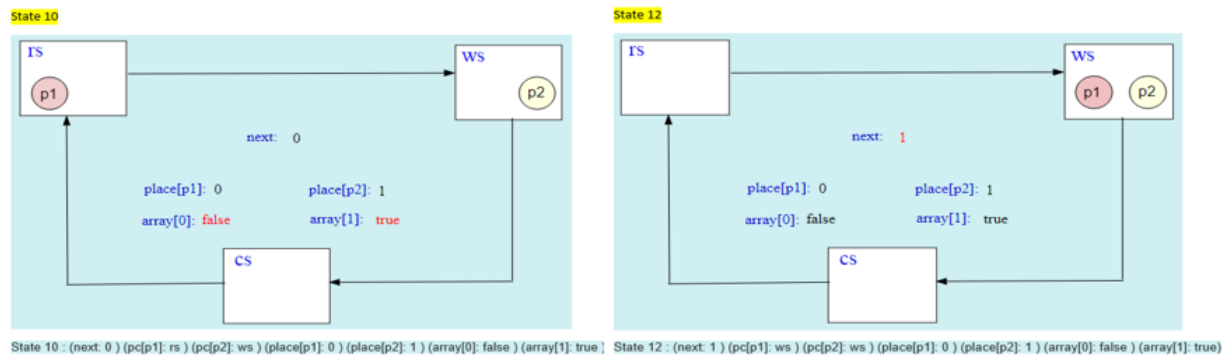

Figure 4.9: States 10 and 12 of Anderson

4.3 Non-deterministic version of Anderson Protocol

The ND-Anderson protocol is a mutual exclusion protocol and Non-deterministic version of Anderson protocol. next and array are natural number variables share by all process. $place[i]$ is a natural number variable that is local to process i. Where $Stm_1 \mid Stm_2$ is a non-deterministic choice statement s.t either Stm_1 or Stm_2 is non-deterministically chosen.

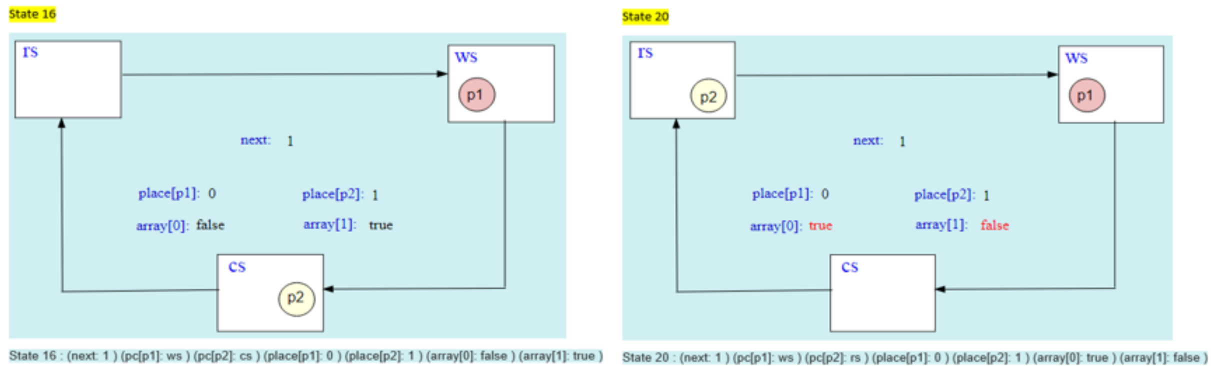

Figure 4.10: States 16 and 20 of Anderson

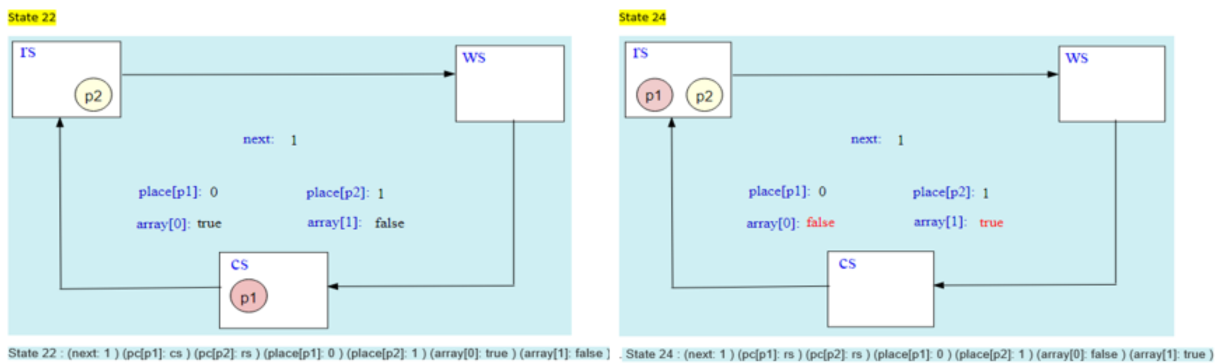

Figure 4.11: States 22 and 24 of Anderson

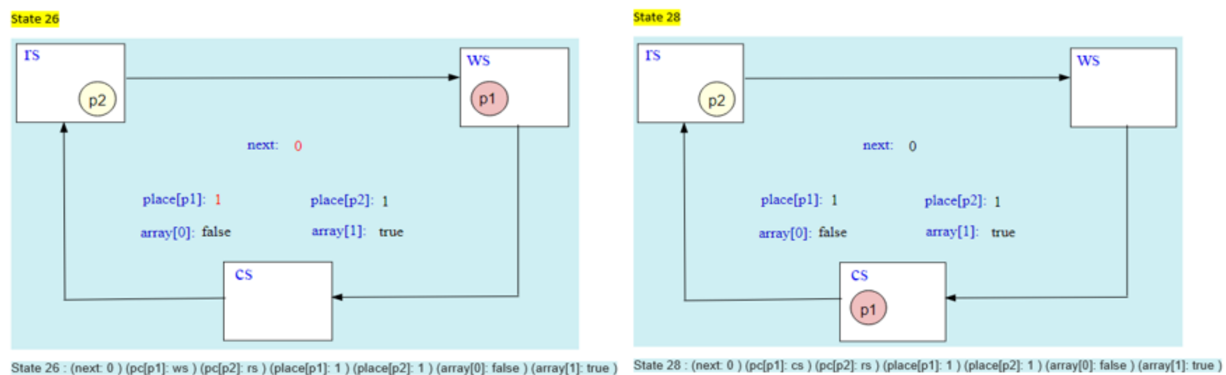

Figure 4.12: States 26 and 28 of Anderson

This version is called ND-Anderson. ND-Anderson for a process i can be described as follows:

Loop: "Remainder Section"
rs: $place[i]$:= fetch&incmode($next$,N) | goto rs;

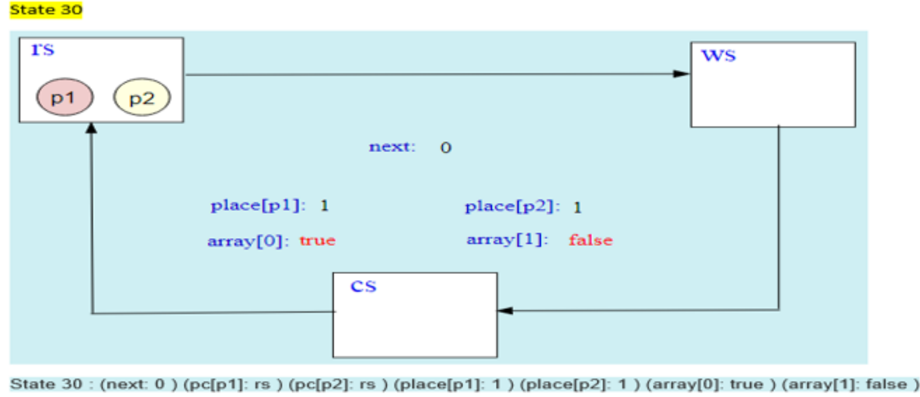

Figure 4.13: State 30 of Anderson

ws: **repeat until** *a*rray[place[i]];
"Critical Section"
cs: *a*rray[*p*lace[i]], *a*rray[(*p*lace[i]+1) % N:= false, true;

4.3.1 Specification of ND-Anderson in Maude and State Transition Diagrams

Here are two processes whose are denoted by p1 and p2. I, X and Y, X1 are Maude variables of process IDs, natural numbers and boolean, successively. $place[i]$ is a natural number variable that is local to process i. From some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times.

The state transitions of ND-Anderson are specified as the following four rewrite rules :

```
rl [setPlace] : (pc[I]: rs) (next: X) (place[I]: Y)
=> (pc[I]: ws) (next: ((X + 1) rem N))  (place[I]: X) .
rl [ds] : (pc[I]: rs)  => (pc[I]: rs) .
rl [wait] : (pc[I]: ws) (place[I]: X) (array[X]: true)
=> (pc[I]: cs) (place[I]: X) (array[X]: true) .
crl [chArray] : (pc[I]: cs) (place[I]: X) (array[X]: B1) (array[X1]: B2)
 => (pc[I]: rs) (place[I]: X) (array[X]: false) (array[X1]: true)
if X1 = (X + 1) rem N .
```

incNxt&St, ds, wait, chArray are the names of the four rewrite rules, respectively. The details description of four rewrite rules follows:

rule 1(setPlace) : a process I is located at rs, the content of place is Y, the content of next is X. After that a process I is located at ws, the content of place is X, the content of next is increments X remainder N. Here is rem N and the number of processes N = 2.

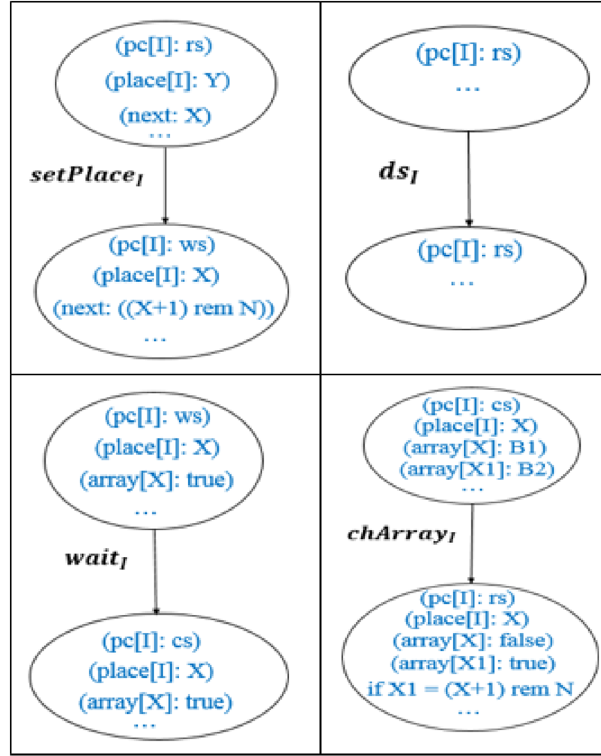

Figure 4.14: State transition diagram of ND-Anderson

rule 2(ds) : From some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times.

rule 3(wait) : a process I is located at ws, the content of place is X, the content of array is true. After that a process I is located at cs, the content of place is X, the content of array is true.

conditional rule 4(chArray) : a process I is located at cs, the content of place is X, the content of array[X] is B1 and array[X1] is B2. After that a process I is located at rs, the content of place is X, Here is rem N and the number of processes N = 2.

Fig. 4.14 shows the four state transition $setPlace_I$, ds_I, $wait_I$ and $chArray_I$ respectively. After the transition from one state to another state we can indicate the process IDs I.

4.3.2 Specification of ND-Anderson as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, cs, ws} of locations.

Three kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(array[0] : true)$- It says that the content of $array[0]$ is true;

- ($array[1] : false$) - It says that the content of $array[1]$ is false;
- ($place : Y$) - It says that the content of $place$ is Y;
- ($next : X$) - It says that the content of $next$ is X;

Where ($pc[p_i]$) is the parametrized name in which $p_i \in$ Pid is a parameter, $l \in$ Loc and $X_N \in$ Pid Nat are values, next $\in$ Pid Nat and array $\in$ Nat Bool . We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in ND-Anderson.

- Set of States, S = {(pc[1] : L_1) . . . (pc[N] : L_N) (next : X)

  (array[X] : B1) (array[X1] : B2) (place[1] : X_1) . . . (place[N] : X_N)

  | $L_1, \ldots, L_N \in$ Loc , $X_1, \ldots, X_N$, X, Y $\in$ Nat, B1, B2 $\in$ Bool}.

- Initial State, I = {(pc[1] : rs) . . . (pc[N] : rs) (place[1] : 0) . . . (place[N] : 0) (next : 0) (array[0] : true) (array[1] : false)}.

- $T_{SetPlace}$ = {((pc[1] : L_1) . . . (pc[I] : rs) . . . (pc[N] : L_N)

  (place[1] : X_1) . . . (place[I] : Y_I) . . . (place[N] : X_N) (next : X) (array[X] : B1) (array[X1] : B2),

  (pc[1] : L_1) . . . (pc[I] : ws) . . . (pc[N] : L_N)

  (place[1] : X_1) . . . (place[I] : X) . . . (place[N] : X_N) (next : ((X + 1) rem N))

  (array[X] : B1) (array[X1] : B2))

  | I $\in$ {1, . . . , N }, L_1 ,. . . , $L_N \in$ Loc , X_1,. . . , X_N, X, Y $\in$ Nat, B1, B2 $\in$ Bool}

- T_{ds} = {((pc[1] : L_1) . . . (pc[I] : rs) . . . (pc[N] : L_N) (next : X) (place[1] : X_1). . . . (place[I] : X_I). . . (place[N] : X_N) (array[X] : B1) (array[X1] : B2),

  (pc[1] : L_1) . . . (pc[I] : rs) . . . (pc[N] : L_N) (next : X) (place[1] : X_1) . . . (place[I] : X_I). . . (place[N] : X_N) (array[X] : B1) (array[X1] : B2))

  | I $\in$ {1, . . . , N }, L_1 ,. . . , $L_N \in$ Loc , X_1,. . . , X_N, X, Y $\in$ Nat, B1, B2 $\in$ Bool}

- T_{wait} = {((pc[1] : L_1) . . . (pc[I] : ws) . . . (pc[N] : L_N)

  (place[1] : X_1). . . . (place[I] : X_I). . . (place[N] : X_N) (array[X] : true) (next : X),

  (pc[1] : L_1) . . . (pc[I] : cs) . . . (pc[N] : L_N) (array[X] : true)

  (place[1] : X_1) . . . (place[I] : X). . . (place[N] : X_N)) (next : X))

  | I $\in$ {1, . . . , N }, L_1 ,. . . , $L_N \in$ Loc , X_1,. . . , X_N, X, Y $\in$ Nat, B1, B2 $\in$ Bool}

- $T_{ChArray}$ = {((pc[1] : L_1) . . . (pc[I] : cs) . . . (pc[N] : L_N) (array[X] : B1) (array[X1] : B2) (place[1] : X_1) . . . (place[I] : X_I) . . . (place[N] : X_N) (next : X),

  (pc[1] : L_1) . . . (pc[I] : rs) . . . (pc[N] : L_N) (array[X] : false) (array[X1] : true) if X1 = (X + 1) rem N

  (place[1] : X_1) . . . (place[I] : X). . . (place[N] : X_N)) (next : X))

  | I $\in$ {1, . . . , N }, L_1 ,. . . , $L_N \in$ Loc , X_1,. . . , X_N, X, Y $\in$ Nat, B1, B2 $\in$ Bool}

4.3.3 Model Checking of ND-Anderson

The following search command used for checking the ND-Anderson.

```
search [1] in ND-Anderson : init =>* (pc[p1]: cs) (pc[p2]: cs) S .
```

No solution. states: 31 rewrites: 169 in 1ms cpu (3ms real) (122641 rewrites/second).

Maude finds no solution meaning ND-Anderson likely to enjoy the mutual exclusion property.

Where ND-Anderson is the module in which ND-Anderson is specified and S is a maude variable of state fragments.

Maude can be confirmed by reducing the following terms :

```
red in ND-Anderson-CHECK : modelCheck(init,lofree) .
```

A counter example is found. The LTL found the following counter example :

```
 counterexample({next: 0 (pc[p1]: rs) (pc[p2]: rs) (place[p1]: 0)
 (place[p2]: 0) (array[0]: true) array[1]: false,'setPlace},
 {next: 1 (pc[p1]: ws) (pc[p2]: rs) (place[p1]: 0) (place[p2]: 0)
  (array[0]: true) array[1]: false,'ds})
```

Althogh p1 is ready to entering the critical section, p2 is always chosen and the rewrite rule ds for p2 is taken, which is not fair for p1.

Chapter 5

Qlock Mutual Exclusion Protocol

5.1 FQlock0: A Flawed Version of Qlock0 Protocol

The FQlock0 protocol [11] is a mutual exclusion protocol. Where *queue* is a queue of process IDs shared by all process. The standard function of queues are enq, top and deq. While tmp_i is a local to each process i. In FQLOCK0, it is not atomic to enqueue a process ID i in *queue* and it is not atomic to dequeue *queue*. The processes are located at five labels rs (Remainder Section), es (Enqueuing Section), ws (Waiting Section), ds (Dequeuing Section) and cs (Critical Section). Initially each process is at rs (Remainder Section) and *queue* is empty.

FQlock0 for a process i can be described as follows:

Loop: "Remainder Section"
rs: *queue* := enq(*queue*,i) ;
es: $q$ueue := tmp_i
ws: **repeat until** top ($q$ueue) = i;
    "Critical Section"
cs: tmp_i := deq ($($queue)
ds: $q$ueue := tmp_i

5.1.1 Specification of FQlock0 in Maude and State Transition Diagrams

There are two processes whose are denoted by p1 and p2. Each state is exposed as $(pc[p1] : l_1)(pc[p1] : l_2)(queue : q)$ $(tmp[p1] : q_1)(tmp[p2] : q_2)$, where l_i is the situated where process i is, q is the queue stored in *queue* and q_i is the queue stored in tmp_i. Initially, l_i is rs, q is empty and q_i is empty. Let init be the term console the initial state.

The state transitions of FQlock0 are specified in maude as the following five rewrite rules:

```
rl [eq1] : (pc[I]: rs) (queue: Q) (tmp[I]: R)
```

```
 => (pc[I]: es) (queue: Q) (tmp[I]: enq(Q,I)) .
rl [eq2] : (pc[I]: es) (queue: Q) (tmp[I]: R)
=> (pc[I]: ws) (queue: R) (tmp[I]: R) .
rl [wt] : (pc[I]: ws) (queue: (I Q))
=> (pc[I]: cs) (queue: (I Q)) .
rl [dq1] : (pc[I]: cs) (queue: Q) (tmp[I]: R)
=> (pc[I]: ds) (queue: Q) (tmp[I]: deq(Q)) .
rl [dq2] : (pc[I]: ds) (queue: Q) (tmp[I]: R)
=> (pc[I]: rs) (queue: R) (tmp[I]: R) .
```

I, Q and R are Maude variables of process IDs and queues of process IDs, respectively. There are five rewrite rules eq1 (Enqueuing 1), eq2 (Enqueuing 2), wt (Waiting), dq1 (dequeuing 1) and dq1 (dequeuing 2) respectively. I Q represents the queue such that I is the top element and Q is the queue obtained by deleting the top.

eq1, eq2, wt, dq1 and dq2 are the names of the five rewrite rules, respectively. The details description of five rewrite rule follows:

rule 1(eq1) : a process I is located at rs, the content of queue is Q, the content of tmp is R. After that a process I is located at es, the content of queue is Q, the content of tmp is enq(Q, I).

rule 2(eq2) : a process I is located at es, the content of queue is Q, the content of tmp is R. After that a process I is located at ws, the content of queue is R, the content of tmp is R.

rule 3(wt) : a process I is located at ws, the content of queue is (I Q). After that a process I is located at cs, the content of queue is (I Q).

rule 4(dq1) : a process I is located at cs, the content of queue is Q, the content of tmp is R. After that a process I is located at ds, the content of queue is Q, the content of tmp is deq(Q).

rule 5(dq2) : a process I is located at ds, the content of queue is Q, the content of tmp is R. After that a process I is located at rs, the content of queue is R, the content of tmp is R.

Figure 5.1 shows the five state transition diagram $eq1_I$, $eq2_I$, wt_I, $dq1_I$ and $dq2_I$ respectively. The every transition of one state to another state we can indicate the process IDs I.

5.1.2 Specification of FQlock0 as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, ds, es, cs, ws} of locations, and PidQueue be the set of queues of process identifiers. empty $\in$ PidQueue is the empty queue. if $p \in$ Pid and $q \in$ PidQueue, then p|q $\in$ PidQueue. The two function enq and deq for PidQueue are defined as follows: for each $q \in$ PidQueue and each p, $p' \in$ Pid, enq(empty,p) = p | empty, enq($p' \mid q, p$) = p' | enq(q, p), deq(empty) = empty, and deq($p \mid q$) = q.

Three kinds of observable components are used:

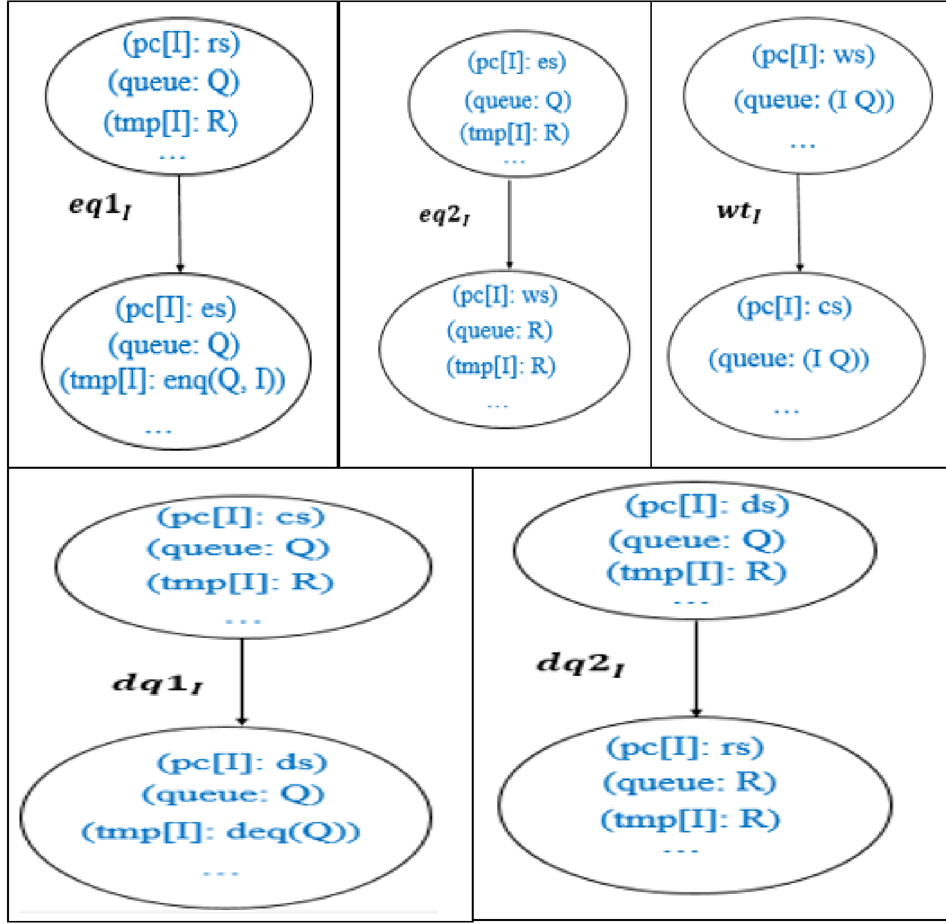

Figure 5.1: State Transition Diagram of FQLOCK0

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(tmp[p] : q_p)$- It says that the content of tmp_p is q_p;
- $(queue : q)$ - It says that the content of $queue$ is q;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, queue is a name, and $l \in$ Loc and $q \in$ PidQueue are values. We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in FQlock0.

- Set of States, S = {(pc[1] : L_1)...(pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)...(tmp[N] : R_N)
  | $L_1 \ldots L_N \in$ Loc, Q, R, $R_1 \ldots R_N \in$ Queue}.
- Initial State, I = {(pc[1] : rs) ...(pc[N] : rs) (queue : empty)
  (tmp[1] : empty)...(tmp[N] : empty)}.

- T_{eq1} = {((pc[1] : L_1) ... (pc[I] : rs)... (pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)... (tmp[I]: R_I)... (tmp[N] : R_N),
  (pc[1] : L_1)... (pc[I] : es)... (pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)... (tmp[I]: enq(Q,I))... (tmp[N] : R_N))
  | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Label, Q, R, R_1 ... R_N ∈ Queue}

- T_{eq2} = {((pc[1] : L_1) ... (pc[I] : es)... (pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)... (tmp[I]: R_I)... (tmp[N] : R_N),
  (pc[1] : L_1)... (pc[I] : ws)... (pc[N] : L_N)
  (tmp[1] : R_1)... (tmp[I]: R_I)... (tmp[N] : R_N) (queue : R))
  | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Label, Q, R, R_1 ... R_N ∈ Queue}

- T_{wt} = {((pc[1] : L_1)... (pc[I] : ws)... (pc[N] : L_N)
  (queue : (I,Q)) (tmp[1] : R_1)... (tmp[I]: R_I)... (tmp[N] : R_N),
  (pc[1] : L_1)... (pc[I] : cs) ... (pc[N] : L_N) (queue : (I,Q))
  (tmp[1] : R_1)... (tmp[I]: R_I... (tmp[N] : R_N))
  | I ∈{1, ..., N}, L_1, ..., L_N ∈ Label, Q, R, R_1 ... R_N ∈ Queue}

- T_{dq1} = {((pc[1] : L_1)... (pc[I] : cs)... (pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)... (tmp[I]: R_I)... (tmp[N] : R_N),
  (pc[1] : L_1)... (pc[I] : ds)... (pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)... (tmp[I]: deq (Q))... (tmp[N] : R_N))
  | I ∈1, ..., N, L_1, ..., L_N ∈ Label, Q, R, R_1 ... R_N ∈ Queue}

- T_{dq2} = {((pc[1] : L_1)... (pc[I] : ds)... (pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)... (tmp[I]: R_I)... (tmp[N] : R_N),
  (pc[1] : L_1)... (pc[I] : rs)... (pc[N] : L_N) (queue : R)
  (tmp[1] : R_1)... (tmp[I]: R)... (tmp[N] : R_N))
  | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Label, Q, R, R_1 ... R_N ∈ Queue}

5.1.3 Model Checking of FQlock0

The following search command used for checking the FQlock0.

```
search [1] in FQlock0 : init =>* (pc[p1]: cs) (pc[p2]: cs) S .
```

Maude finds a solution meaning FQlock0 does not enjoy the property. Here is also used following path search command : show path 28 . This command shows each states of the FQlock0 version by Maude.

Where FQlock0 is the module in which FQlock0 is specified and S is a Maude variable of state fragments. The search command finds the following counterexample:

```
state 0, Sys: queue: empty (pc[p1]: rs) (pc[p2]: rs) (tmp[p1]: empty)
 tmp[p2]: empty===[ rl queue: Q (pc[I]: rs) tmp[I]: R =>
 (queue: Q tmp[I]: enq(Q, I)) pc[I]:es [label eq1] . ]===>
state 1, Sys: queue: empty (pc[p1]: es) (pc[p2]: rs) (tmp[p1]: p1 empty)
tmp[p2]: empty===[ rl queue: Q (pc[I]: rs) tmp[I]: R => (queue: Q
tmp[I]: enq(Q, I)) pc[I]:es [label eq1] . ]===>
state 3, Sys: queue: empty (pc[p1]: es) (pc[p2]: es) (tmp[p1]: p1 empty)
tmp[p2]: p2 empty===[ rl queue: Q (pc[I]: es) tmp[I]: R =>
(queue: R tmp[I]: R) pc[I]: ws [label eq2] . ]===>
state 6, Sys: queue: (p1 empty) (pc[p1]: ws) (pc[p2]: es)
(tmp[p1]: p1 empty)tmp[p2]: p2 empty===[ rl queue: (I Q) pc[I]: ws =>
 queue: (I Q) pc[I]: cs [label wt] . ]===>
state 13, Sys: queue: (p1 empty) (pc[p1]: cs) (pc[p2]: es)
 (tmp[p1]: p1 empty) tmp[p2]: p2 empty===[ rl queue: Q (pc[I]: es)
 tmp[I]: R => (queue: R tmp[I]: R) pc[I]: ws [label eq2] . ]===>
state 23, Sys: queue: (p2 empty) (pc[p1]: cs) (pc[p2]: ws)
 (tmp[p1]: p1 empty) tmp[p2]: p2 empty===[ rl queue: (I Q) pc[I]: ws =>
 queue: (I Q) pc[I]: cs [label wt] . ]===>
state 33, Sys: queue: (p2 empty) (pc[p1]: cs) (pc[p2]: cs)
(tmp[p1]: p1 empty) tmp[p2]: p2 empty
```

Maude can generate counterexample without any type of difficulties, which non-experts can not do it.

5.1.4 Graphical Animations of FQlock0 Counterexamples

We used SMGA for drawing the seven pictures. The Fig. 5.2 to 5.5 show that counterexample which found by Maude software for FQlock0 version. These pictures make it possible to reorganize at which location the each process is, what the value stored in queue and what the value stored in tmp1 and tmp2.

```
###keys
queue pc[p1] pc[p2] tmp[p1] tmp[p2]

###textDisplay
queue::::REV::::_ _
tmp[p1]::::REV::::_ _
tmp[p2]::::REV::::_ _
```

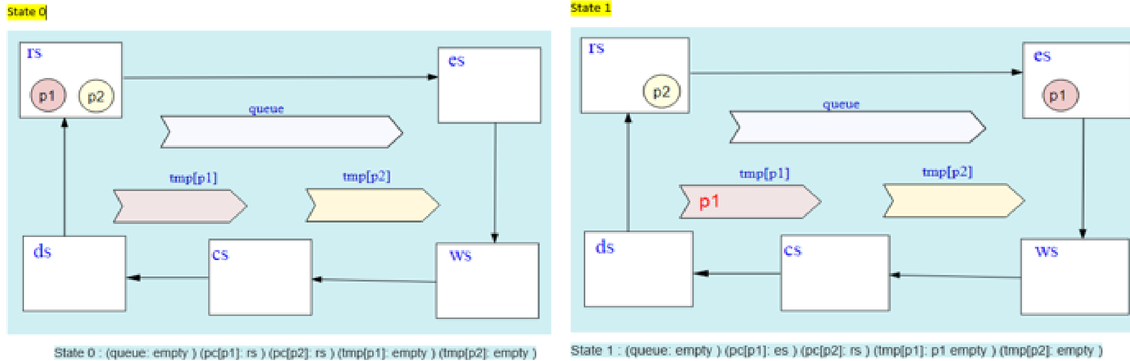

Figure 5.2: Counterexample for FQlock0 of states 0 and 1

```
###states
(queue: (empty) (pc[p1]: rs) (pc[p2]: rs) (tmp[p1]: empty)
tmp[p2]: empty) ||
(queue: (empty) (pc[p1]: es) (pc[p2]: rs) (tmp[p1]: p1 empty)
tmp[p2]: empty) ||
(queue: (empty) (pc[p1]: es) (pc[p2]: es) (tmp[p1]: p1 empty)
tmp[p2]: p2 empty) ||
(queue: (p1 empty) (pc[p1]: ws) (pc[p2]: es) (tmp[p1]: p1 empty)
 tmp[p2]: p2 empty) ||
(queue: (p1 empty) (pc[p1]: cs) (pc[p2]: es) (tmp[p1]: p1 empty)
tmp[p2]: p2 empty) ||
(queue: (p2 empty) (pc[p1]: cs) (pc[p2]: ws) (tmp[p1]: p1 empty)
tmp[p2]: p2 empty) ||
(queue: (p2 empty) (pc[p1]: cs) (pc[p2]: cs) (tmp[p1]: p1 empty)
tmp[p2]: p2 empty)
```

5.2 FQlock1 Protocol

The FQlock1 protocol [11] is a mutual exclusion protocol. Where *queue* is a queue of process IDs shared by all process. The standard function of queues are enq, top and deq. While tmp_i is a local to each process i. In FQLOCK1, it is not atomic to enqueue a process ID i in *queue* and it is not atomic to dequeue *queue*. The processes are located at four labels rs (Remainder Section), ws (Waiting Section), ds (Dequeuing Section) and cs (Critical Section). Initially each process is at rs(Remainder Section) and *queue* is empty.

FQlock1 for a process i can be described as follows:

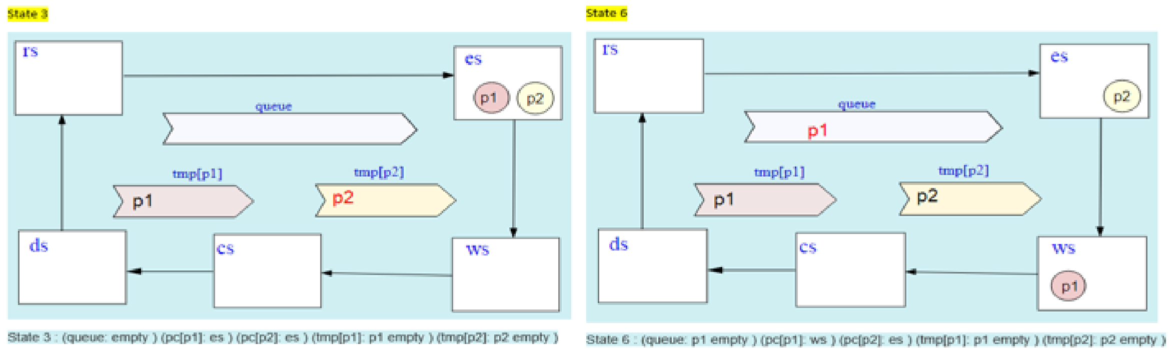

Figure 5.3: Counterexample for FQlock0 of states 3 and 6

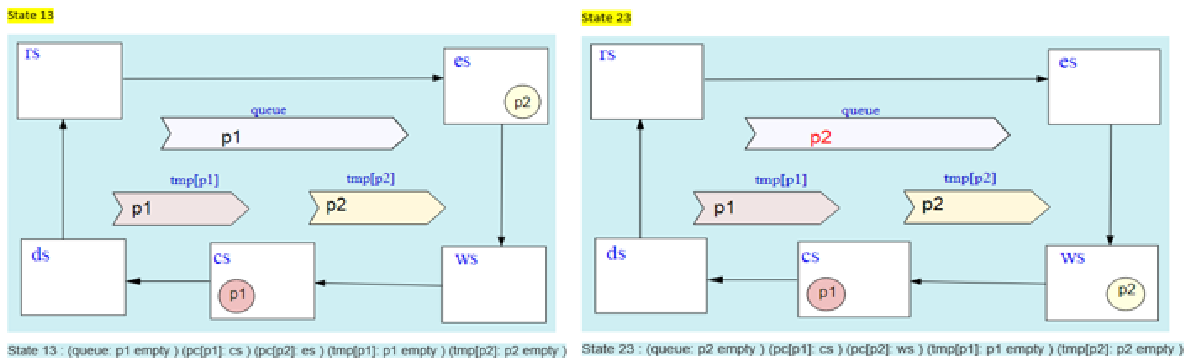

Figure 5.4: Counterexample for FQlock0 of states 13 and 23

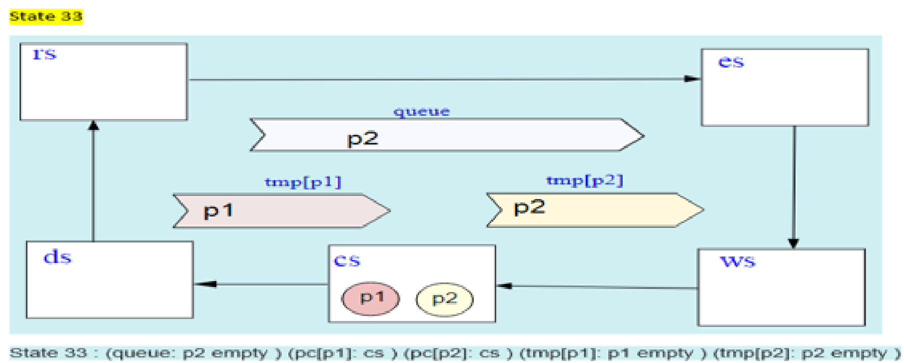

Figure 5.5: Counterexample for FQlock0 of state 33

Loop: "Remainder Section"
rs: $queue$:= enq($queue$,i) ;
ws: **repeat until** top (queue) = i;
  "Critical Section"
cs: tmp_i := deq (queue)
ds: queue := tmp_i

5.2.1 Specification of FQlock1 in Maude and State Transition Diagrams

There are two processes whose are denoted by p1 and p2. I, Q and R are Maude variables of process IDs and queues of process IDs, respectively. There are four rules eq1 (Enqueuing), wt (Waiting), dq1 (dequeuing 1) and dq2 (dequeuing 2) respectively. I Q represents the queue such that I is the top element and Q is the queue obtained by deleting the top. There are only difference between FQlock0 and FQlock1 is that it is atomic to enqueue a process ID i into queue in FQlock1, while it is not in FQlock0.

The state transitions of FQlock1 are specified as the following four rewrite rules :

```
rl [eq] : (pc[I]: rs) (queue: Q)
=> (pc[I]: ws) (queue: enq(Q,I)) .
rl [wt] : (pc[I]: ws) (queue: (I Q))
=> (pc[I]: cs) (queue: (I Q)) .
rl [dq1] : (pc[I]: cs) (queue: Q) (tmp[I]: R)
=> (pc[I]: ds) (queue: Q) (tmp[I]: deq(Q)) .
rl [dq2] : (pc[I]: ds) (queue: Q) (tmp[I]: R)
=> (pc[I]: rs) (queue: R) (tmp[I]: R) .
```

eq, wt, dq1 and dq2 are the names of the four rewrite rules, respectively. The details description of four rewrite rule follows:

rule 1(eq) : a process I is located at rs, the content of queue is Q. After that a process I is located at ws, the content of queue is enq(Q, I).

rule 2(wt) : a process I is located at ws, the content of queue is queue(I Q). After that a process I is located at cs, the content of queue is queue(I Q).

rule 3(dq1) : a process I is located at cs, the content of queue is Q, the content of tmp is R. After that a process I is located at ds, the content of queue is Q, the content of tmp is deq(Q).

rule 4(dq2) : a process I is located at ds, the content of queue is Q, the content of tmp is R. After that a process I is located at rs, the content of queue is R, the content of tmp is R.

Fig. 5.6 shows the four state transition eq_I, wt_I, $dq1_I$ and $dq2_I$ respectively. After the transition from one state to another state we can indicate the process IDs I.

5.2.2 Specification of FQlock1 as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, ds, cs, ws}, of locations, and PidQueue be the set of queues of process identifiers. empty $\in$ PidQueue is the empty queue. if $p \in$ Pid and $q \in$ PidQueue, then p|q $\in$ PidQueue. The two function enq and deq for PidQueue are defined as follows: for each $q \in$ PidQueue and each p, $p' \in$ Pid, enq(empty,p) = p | empty, enq(p' | q, p) = p' | enq(q, p), deq(empty) = empty, and deq(p | q) = q.

Three kinds of observable components are used:

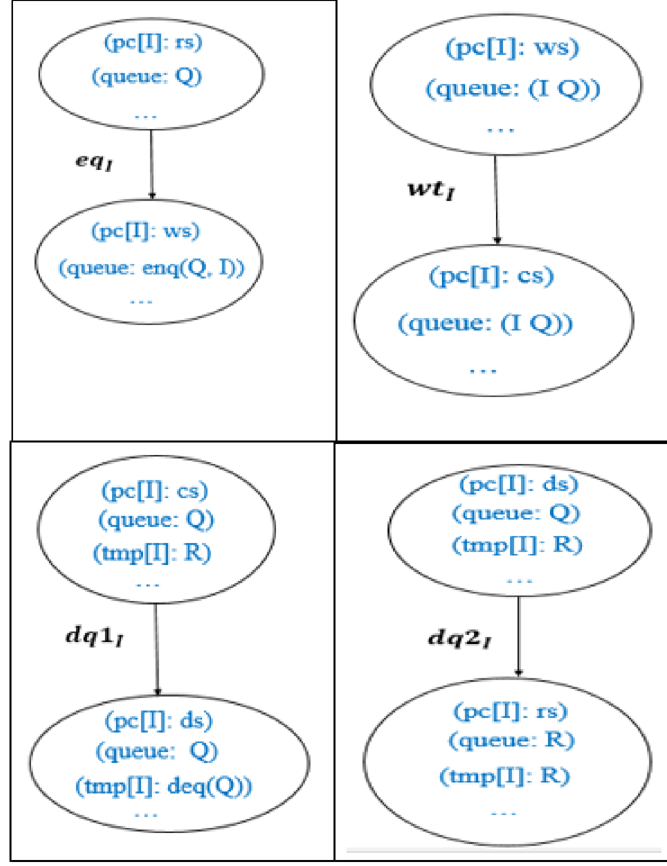

Figure 5.6: State Transition Diagram of FQLOCK1

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(tmp[p] : q_p)$ - It says that the content of tmp is q_p ;
- $(queue : q)$ - It says that the content of $queue$ is q ;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, queue is a name, and $l \in$ Loc and $q \in$ PidQueue are values. We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in FQlock1.

- Set of States, S = {(pc[1] : L_1)...(pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)...(tmp[N] : R_N)
  | $L_1 \ldots L_N \in$ Loc, Q, R$\in$ Queue}.
- Initial State, I = {(pc[1] : rs) ...(pc[N] : rs) (queue : empty)
  (tmp[1] : empty)...(tmp[N] : empty)}.

- T_{eq} = {((pc[1] : L_1) ...(pc[I] : ds)...(pc[N] : L_N)
  (queue : Q) (tmp[1] : R_1)...(tmp[I]: R_I)...(tmp[N] : R_N),
  (pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N) (queue : enq (Q, I))
  (tmp[1] : R_1)...(tmp[I]: R_I)...(tmp[N] : R_N))
  | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Loc, Q, R, R_1 ...R_N ∈ Queue}
- T_{wait} = {((pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N)
  (queue : (I Q)) (tmp[1] : R_1)...(tmp[I]: R_I)...(tmp[N] : R_N),
  (pc[1] : L_1)...(pc[I] : cs) ...(pc[N] : L_N) (queue : (I Q))
  (tmp[1] : R_1)...(tmp[I]: R_I)...(tmp[N] : R_N))
  | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Loc, Q, R, R_1 ...R_N ∈ Queue}
- T_{dq1} = {((pc[1] : L_1)...(pc[I] : cs)...(pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)...(tmp[I]: R_I)...(tmp[N] : R_N),
  (pc[1] : L_1)...(pc[I] : ds)...(pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)...(tmp[I]: deq (Q))...(tmp[N] : R_N))
  | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Loc, Q, R, R_1 ...R_N ∈ Queue}
- T_{dq2} = {((pc[1] : L_1)...(pc[I] : ds)...(pc[N] : L_N) (queue : Q)
  (tmp[1] : R_1)...(tmp[I]: R_I)...(tmp[N] : R_N),
  (pc[1] : L_1)...(pc[I] : rs)...(pc[N] : L_N) (queue : R)
  (tmp[1] : R_1)...(tmp[I]: R)...(tmp[N] : R_N))
  | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Loc, Q, R, R_1 ...R_N ∈ Queue}

5.2.3 Model Checking of FQlock1

The search command does not find any counterexamples [12] that FQlock1 enjoys the mutex property. Therefore, FQlock1 is likely to enjoy the mutex property.

The following LTL search command used for checking the FQlock1.

```
red in FQLOCK1-CHECK : modelCheck(init,lofree) .
```

To use the Maude LTL model checker to check if FQlock1 enjoys the lockout freedom property, we need two kinds of atomic propositions **wait(P)** and **crit(P)**, where **P** is a process ID. Users are also supposed to specify a labeling function. For our purpose, we declare the three equations : **eq(pc[P] : ws) S | = want(P) = true., eq (pc[P] : cs) S | = crip(P) = true., and eq S | = PROP = false** [**owise**] ., where **P** is a Maude variable of process IDs, **S** is a Maude variable of atomic propositions. The three equations say a state **s** satisfies **want(P)** if and only if **(pc[P] : ws)** appears in **s** and **s** satisfies **crit(P)** if and only if **(pc[P] : cs)** appears in **s**. Then, users are

supposed to specify LTL formulas to check. The lockout freedom property is expressed as **want(P)** $\leadsto$ **crit(P)**, where $\leadsto$ is the LTL leadsto operator. In Maude, the formula is specified as **eq lpfree = (wait(p1)** $\mapsto$ **crit(p1))** /\ **(wait(p2)** $\mapsto$ **crit(p2))**, where the operator $\mapsto$ denotes the leadsto $\leadsto$. The model checking is conducted by reducing **modelCheck(init,lofree(p1))**, finding a counterexample. A counterexample generated by Maude LTL model checker consists of a finite computation leading to an infinite loop in which a finite path repeats forever. In the counterexample generated by Maude LTL model checker of the lockout freedom property for FQlock1.

Counterexamples generated by the Maude LTL model checker are not necessarily the shortest ones. Therefore, the second author mainly developed a meta-program in Maude that takes a counterexample generated by the Maude LTL model checker and generates a shorter one[12]. For the counterexample generated by Maude LTL model checker of the lockout freedom property for FQlock1, the finite computation in the shortened one consists of nine states and the finite path in the loop of the shortened one consists of four states. The finite computation in the shortened one is as follows:

```
 counterexample({queue: empty (pc[p1]: rs) (pc[p2]: rs)
(tmp[p1]: empty) tmp[p2]: empty,'eq}
{queue: (p1 empty) (pc[p1]: ws) (pc[p2]: rs) (tmp[p1]: empty)
tmp[p2]: empty,'eq}
{queue: (p1 p2 empty)(pc[p1]: ws) (pc[p2]: ws) (tmp[p1]: empty)
 tmp[p2]: empty,'wt}
{queue: (p1 p2 empty) (pc[p1]: cs) (pc[p2]: ws) (tmp[p1]: empty)
tmp[p2]: empty,'dq1} {queue: (p1 p2 empty) (pc[p1]: ds) (pc[p2]: ws)
(tmp[p1]: p2 empty) tmp[p2]:empty,'dq2} {queue: (p2 empty) (pc[p1]: rs)
(pc[p2]: ws) (tmp[p1]: p2 empty) tmp[p2]: empty,'eq} {queue: (p2 p1 empty)
(pc[p1]: ws) (pc[p2]: ws)(tmp[p1]: p2 empty) tmp[p2]: empty,'wt}
{queue: (p2 p1 empty) (pc[p1]: ws)(pc[p2]: cs) (tmp[p1]: p2 empty)
tmp[p2]: empty,'dq1} {queue: (p2 p1 empty)(pc[p1]: ws) (pc[p2]: ds)
(tmp[p1]: p2 empty) tmp[p2]: p1 empty,'dq2} {queue: (p1 empty) (pc[p1]: ws)
(pc[p2]: rs) (tmp[p1]: p2 empty) tmp[p2]: p1 empty,'eq} {queue: (p1 p2 empty)
(pc[p1]: ws) (pc[p2]: ws) (tmp[p1]: p2 empty) tmp[p2]: p1 empty,'wt}
{queue: (p1 p2 empty) (pc[p1]: cs) (pc[p2]: ws) (tmp[p1]: p2 empty)
tmp[p2]: p1 empty,'dq1} {queue: (p1 p2 empty) (pc[p1]: ds) (pc[p2]: ws)
(tmp[p1]: p2 empty) tmp[p2]: p1 empty,'dq2} {queue: (p2 empty) (pc[p1]: rs)
(pc[p2]: ws) (tmp[p1]: p2 empty) tmp[p2]: p1 empty,'wt} {queue: (p2 empty)
(pc[p1]: rs) (pc[p2]: cs) (tmp[p1]: p2 empty) tmp[p2]: p1 empty,'dq1}
{queue: (p2 empty) (pc[p1]: rs) (pc[p2]: ds) (tmp[p1]:p2 empty)
tmp[p2]: empty,'eq}
{queue: (p2 p1 empty) (pc[p1]: ws) (pc[p2]:ds) (tmp[p1]: p2 empty)
tmp[p2]: empty,'dq2}, {queue: empty (pc[p1]: ws) (pc[p2]: rs)
(tmp[p1]: p2 empty)
tmp[p2]: empty,'eq} {queue: (p2 empty) (pc[p1]: ws) (pc[p2]: ws)
(tmp[p1]: p2 empty) tmp[p2]: empty,'wt}
```

```
{queue: (p2empty) (pc[p1]: ws) (pc[p2]: cs) (tmp[p1]: p2 empty)
tmp[p2]: empty,'dq1} {queue: (p2 empty) (pc[p1]: ws) (pc[p2]: ds)
(tmp[p1]: p2 empty) tmp[p2]:empty,'dq2})
```

5.2.4 Graphical Animations of FQlock1

The Fig. 5.7 to 5.13 show that the each state which found by Maude software for FQlock1 version. We used SMGA for drawing the thirteen pictures. These pictures make it possible to reorganize at which location of located the each process is, what the value content in queue and tmp.

```
###keys
queue pc[p1] pc[p2] tmp[p1] tmp[p2]

###textDisplay
queue::::REV::::_ _
tmp[p1]::::REV::::_ _
tmp[p2]::::REV::::_ _

###states
(queue: empty (pc[p1]: rs) (pc[p2]: rs) (tmp[p1]: empty)
tmp[p2]: empty) ||
(queue: (p1 empty) (pc[p1]: ws) (pc[p2]: rs) (tmp[p1]: empty)
 tmp[p2]: empty) ||
(queue: (p1 p2 empty) (pc[p1]: ws) (pc[p2]: ws) (tmp[p1]: empty)
tmp[p2]: empty) ||
(queue: (p1 p2 empty) (pc[p1]: cs) (pc[p2]: ws) (tmp[p1]: empty)
tmp[p2]: empty) ||
(queue: (p1 p2 empty) (pc[p1]: ds) (pc[p2]: ws) (tmp[p1]: p2 empty)
 tmp[p2]: empty) ||
(queue: (p2 empty) (pc[p1]: rs) (pc[p2]: ws) (tmp[p1]: p2 empty)
tmp[p2]: empty) ||
(queue: (p2 empty) (pc[p1]: rs) (pc[p2]: cs) (tmp[p1]: p2 empty)
tmp[p2]: empty) ||
(queue: (p2 empty) (pc[p1]: rs) (pc[p2]: ds) (tmp[p1]: p2 empty)
tmp[p2]: empty) ||
(queue: (p2 p1 empty) (pc[p1]: ws) (pc[p2]: ds) (tmp[p1]: p2 empty)
 tmp[p2]: empty)||
(queue: empty (pc[p1]: ws) (pc[p2]: rs) (tmp[p1]: p2 empty)
tmp[p2]: empty) ||
(queue: (p2 empty) (pc[p1]: ws) (pc[p2]: ws) (tmp[p1]: p2 empty)
tmp[p2]: empty) ||
(queue: (p2 empty) (pc[p1]: ws) (pc[p2]: cs) (tmp[p1]: p2 empty)
```

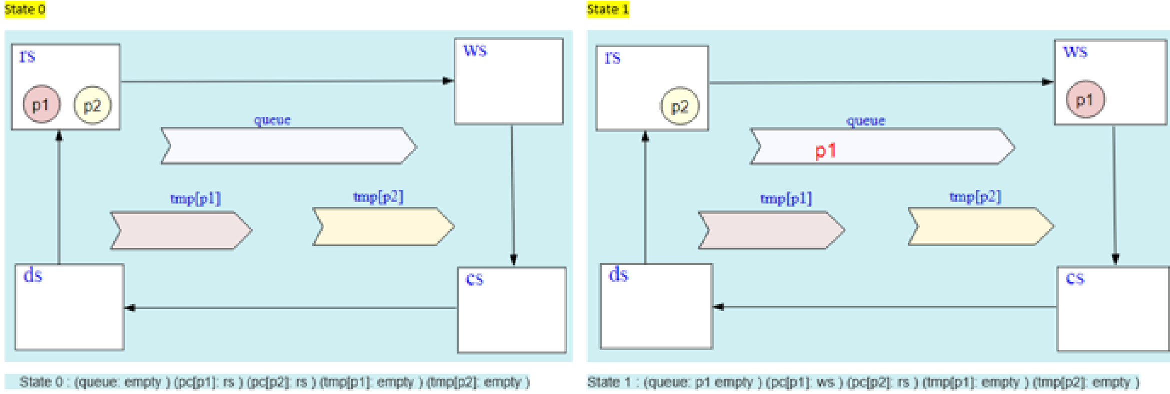

Figure 5.7: States 0 and 1 of counterexample of the lockout freedom property for FQLOCK1

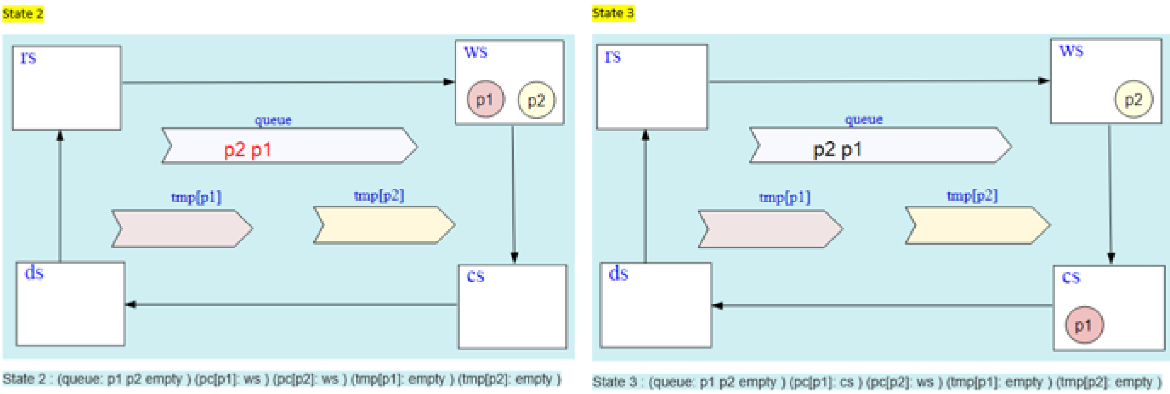

Figure 5.8: States 2 and 3 of counterexample of the lockout freedom property for FQLOCK1

```
tmp[p2]: empty) ||
(queue: (p2 empty) (pc[p1]: ws) (pc[p2]: ds) (tmp[p1]: p2 empty)
 tmp[p2]: empty)
```

5.3 Qlock Mutual Exclusion Protocol

The Qlock protocol [13] is a mutual exclusion protocol that uses an atomic queue of process identifiers. Where *queue* is a queue of process IDs shared by all processes. The standard function of queues are enq, top and deq. The processes are located at three labels rs (Remainder Section), ws (Waiting Section) and cs (Critical Section). We suspect that each of enqueuing an element into *queue* and dequeuing *queue* is atomic, and so is one is one iteration of the loop at ws. Initially each process is at rs (Remainder Section) and *queue* is empty.

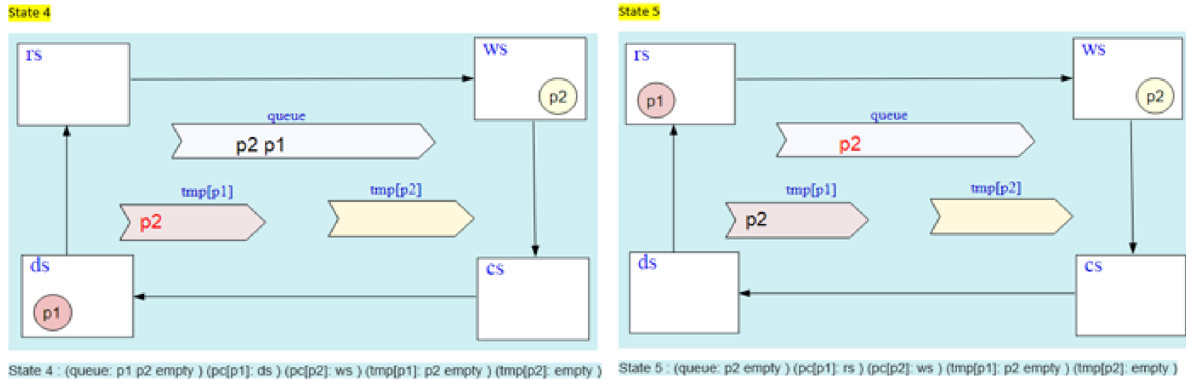

Figure 5.9: States 4 and 5 of counterexample of the lockout freedom property for FQLOCK1

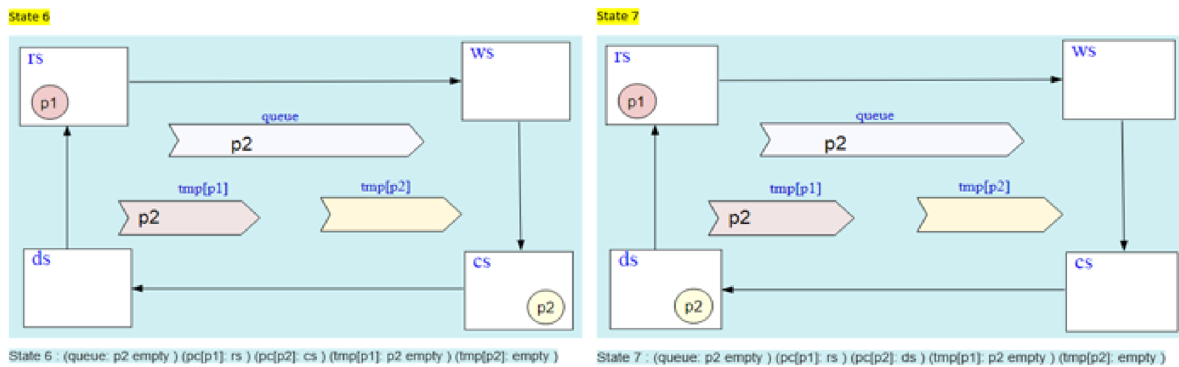

Figure 5.10: States 6 and 7 of counterexample of the lockout freedom property for FQLOCK1

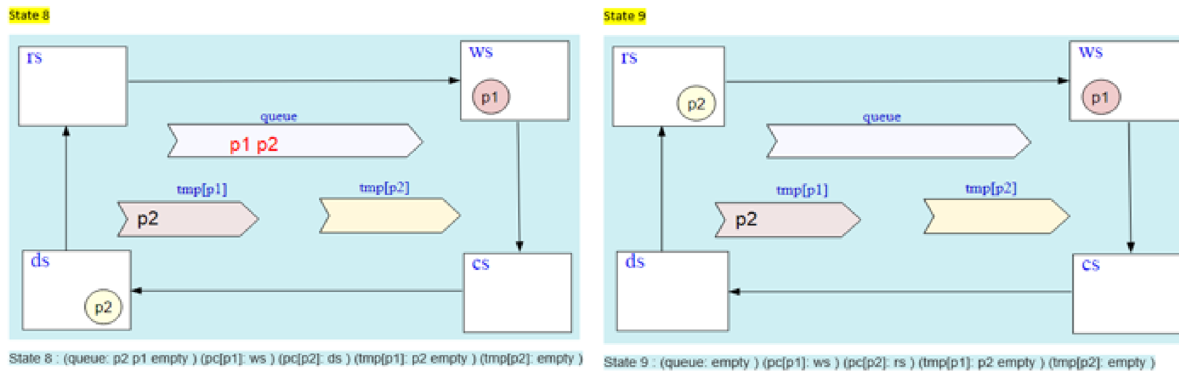

Figure 5.11: States 8 and 9 of counterexample of the lockout freedom property for FQLOCK1

The pseudo-code Qlock for a process i can be described as follows:

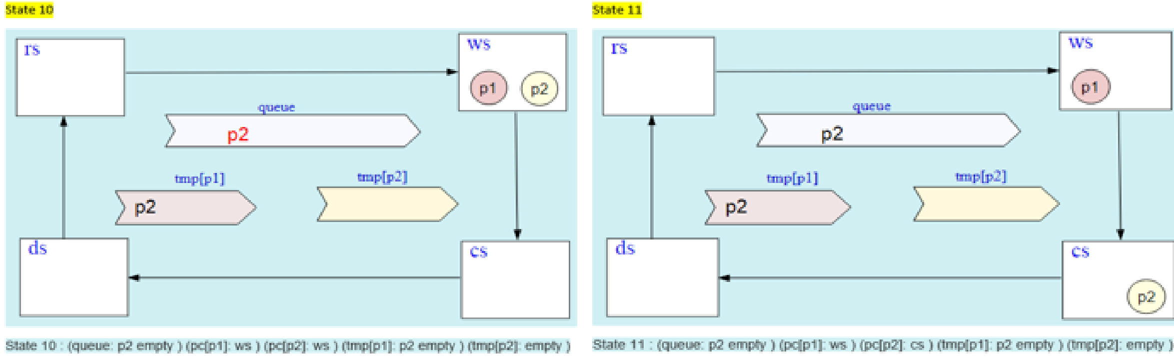

Figure 5.12: States 10 and 11 of counterexample of the lockout freedom property for FQLOCK1

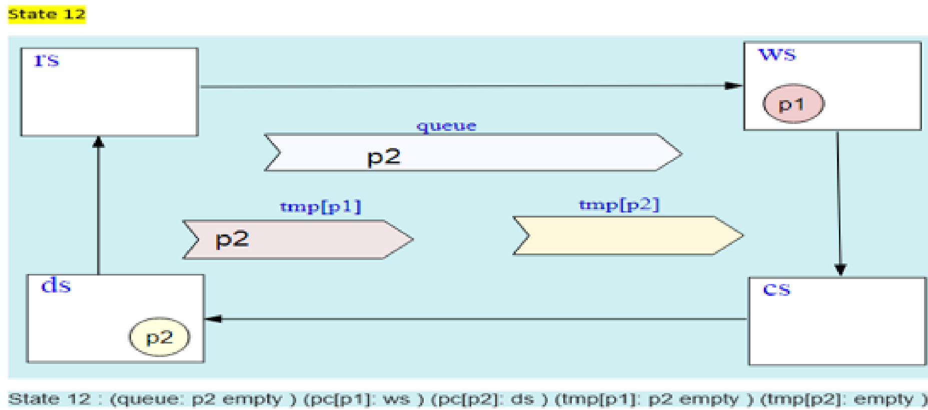

Figure 5.13: State 12 of counterexample of the lockout freedom property for FQLOCK1

Loop: "Remainder Section"
rs: enq($queue$,i);
ws: **repeat until** top($q$ueue) = i;
    "Critical Section"
cs: deq($q$ueue);

5.3.1 Specification of Qlock in Maude

We assume that there are five processes whose are denoted by p1, p2, p3, p4 and p5. I and Q are Maude variables of process IDs, successively. There are three rewrite rules : eq (Enqueuing), wt (Waiting) and dq (dequeuing). There are two Maude variables I and Q and process ID queues. I Q represents the queue such that I is the top element and Q is the queue obtained by deleting the top.

The state transitions of Qlock are specified as the following three rewrite rules :

```
rl [eq] : (pc[I]: rs) (queue: Q) => (pc[I]: ws) (queue: enq(Q,I)) .
```

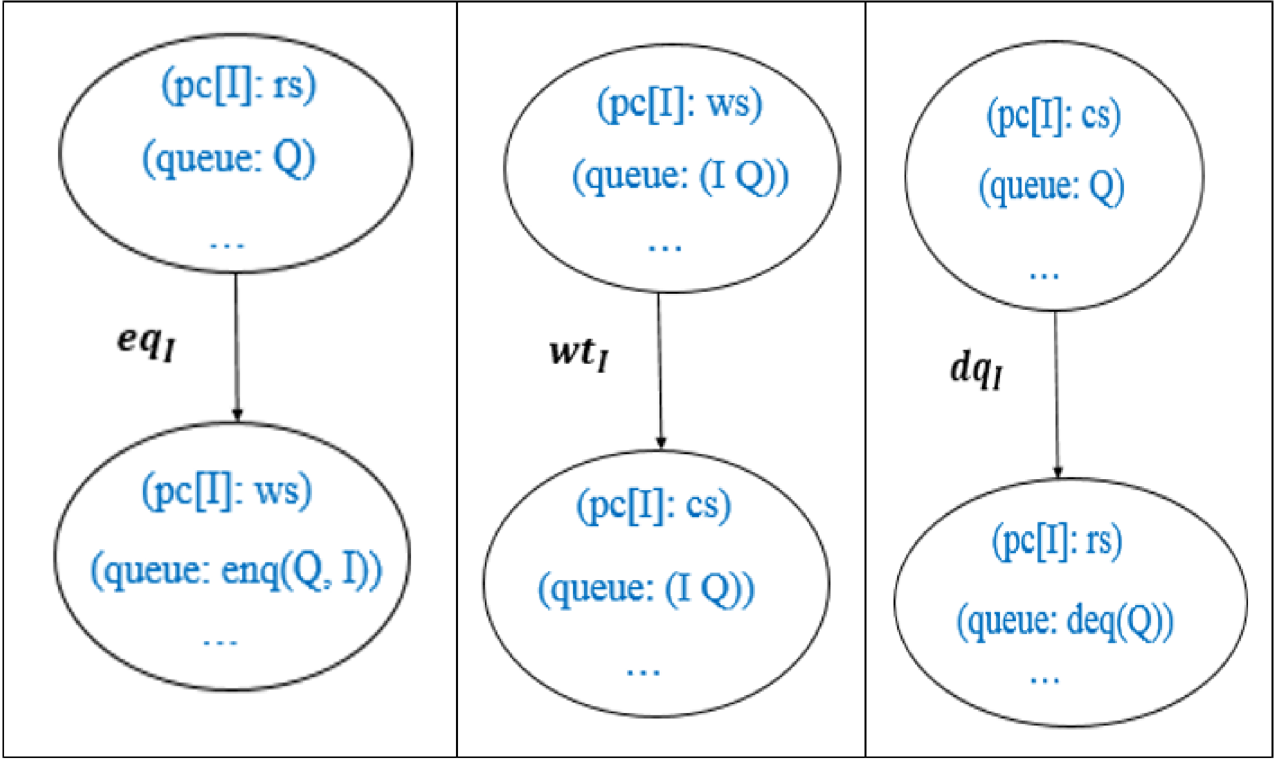

Figure 5.14: State Transition Diagram of QLOCK

```
rl [wt] : (pc[I]: ws) (queue: (I Q)) => (pc[I]: cs) (queue: (I Q)) .
rl [dq] : (pc[I]: cs) (queue: Q) => (pc[I]: rs) (queue: deq(Q)) .
```

eq, wt and dq are the names of the three rewrite rules, respectively. The details description of three rewrite rules follows:

rule 1(eq) : a process I is located at rs, the content of queue is Q. After that a process I is located at ws, the content of queue is enq(Q, I).

rule 2(wt) : a process I is located at ws, the content of queue is (I Q). After that a process I is located at cs, the content of queue is (I Q).

rule 3(dq) : a process I is located at cs, the content of queue is Q. After that a process I is located at rs, the content of queue is deq(Q).

Fig. 5.14 shows the three state transition eq_I, wt_I and dq_I respectively. After the transition from one state to another state we can indicate the process IDs I.

5.3.2 Specification of Qlock as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, cs, ws}, of locations, and PidQueue be the set of queues of process identifiers. empty ∈ PidQueue is the empty queue. if $p \in$ Pid and $q \in$ PidQueue, then p|q ∈ PidQueue. The two function enq and

deq for PidQueue are defined as follows: for each $q \in$ PidQueue and each p, $p' \in$ Pid, enq(empty,p) = p | empty, enq(p' | q, p) = p' | enq(q, p), deq(empty) = empty, and deq(p | q) = q.

Two kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(queue : q)$ - It says that the content of *queue* is q;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, queue is a name, and $l \in$ Loc and $Q \in$ PidQueue are values. We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in Qlock.

- Set of States, S = $\{$(pc[1] : L_1)...(pc[N] : L_N) (queue : Q) | $L_1 \ldots L_N \in$ Loc, Q $\in$ PidQueue$\}$.
- Initial State, I = $\{$(pc[1] : rs) ...(pc[N] : rs) (queue : empty)$\}$.
- T_{eq} = $\{$((pc[1] : L_1) ...(pc[I] : rs)...(pc[N] : L_N) (queue : Q), (pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N) (queue : enq(Q, I))) | I $\in \{$1, ..., N$\}$, $L_1, \ldots, L_N \in$ Loc, Q $\in$ PidQueue$\}$
- T_{wt} = $\{$((pc[1] : L_1)...(pc[I] : ws)...(pc[N] : L_N) (queue : (I, Q)), (pc[1] : L_1)...(pc[I] : cs) ...(pc[N] : L_N) (queue : (I, Q))) | I $\in \{$1, ..., N$\}$, $L_1, \ldots, L_N \in$ Loc, Q $\in$ PidQueue$\}$
- T_{dq} = $\{$((pc[1] : L_1)...(pc[I] : cs)...(pc[N] : L_N) (queue : Q), (pc[1] : L_1)...(pc[I] : rs)...(pc[N] : L_N) (queue : deq(Q))) | I $\in \{$1, ..., N$\}$, $L_1, \ldots, L_N \in$ Loc, Q $\in$ PidQueue$\}$

5.3.3 Model Checking of Qlock

- **Confirming Characteristics with Maude**

Pro. 1 can be confirmed as follows:

```
search [1] in QLOCK : init =>*
(pc[I]: L1) (pc[J]: L2) S such that
not (L1== cs implies not (L2 == cs)) .
```

Where QLOCK is the module in which Qlock is specified, I and J are Maude variables of process IDs, L1 and L2 are Maude variables of labels, and S is a Maude variable observable component soups. Prop. 1 can be rephrased as that whenever a process I is at cs, no other process J is at cs. The search tries to find a state in which Prop. 1 is broken.

No such state was found by the search command. Thus, we have confirmed Prop. 1 with Maude.

We have guessed the whatever there is a process at cs, queue is not empty. The guessed property is called Pro. 2, which will be confirmed by the following search command:

```
search [1] in QLOCK : init =>*
(pc[I]: L1) (queue: Q) S such that
not (L1== cs implies not (Q == empty)) .
```

Where I is a Maude variable of process IDs, L1 is a Maude variables of labels, Q is a Maude variables of process ID queues, and S is a Maude variables of observable component soups. The search command did not find any states, and then we have confirmed Prop. 2.

Hence, Prop. 2 can be refined as follows: whenever there is a process at cs, the process is the top of queue. The guessed property is called Prop. 3, which will be confirmed by the following search command:

```
search [1] in QLOCK : init =>*
(pc[I]: L1) (queue: (J Q)) S  such that
not (L1 == cs implies I == J) .
```

Where I and J are Maude variables of process IDs, L1 is a Maude variable of labels, Q is a Maude variables of ID queues, and S is a Maude variable of observable component soups. The search command did not find any states, and then we have confirmed Prop. 3.

The five properties will be confirmed by the following five search commands:

```
search [1] in QLOCK : init =>*
(pc[I]: L1) (queue: Q) S such that
not (Q == empty implies L1 == rs) .

search [1] in QLOCK : init =>*
(pc[I]: L1) (queue: Q) S such that
not ((L1 == ws or L1 == cs) implies I $\in$ Q) .

search [1] in QLOCK : init =>*
(pc[I]: L1) (queue: Q) S such that
not (I $\in$ Q implies (L1 == ws or L1 == cs)) .

search [1] in QLOCK : init =>*
(pc[I]: L1) (queue: Q) S such that
not ((not I $\in$ Q) implies L1 == rs) .

search [1] in QLOCK : init =>*
(pc[I]: L1) (queue: Q) S such that
not (L1== rs implies (not I $ \in$ Q)) .
```

Where I is a Maude variable of process IDs, L1 is a Maude variable of labels, Q is a Maude variable of process ID queues, and S is a Maude variable of observable component soups. Each of the five search commands did not find any states, and therefore we have confirmed five guessed properties.

The maude search command find the follows:

```
state 0, Sys: queue: (empty) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]:
rs) (pc[p4]: rs) pc[p5]: rs
===[ rl queue: Q pc[I]: cs => queue: deq(Q) pc[I]: rs [label dq] . ]===>
state 1, Sys: queue: (p3 p2 p1 p4 p5 empty) (pc[p1]: ws)
(pc[p2]: ws) (pc[p3]: ws) (pc[p4]: cs) pc[p5]: ws
===[ rl queue: (I Q) pc[I]: ws => queue: (I Q) pc[I]: cs [label wt] . ]===>
state 3, Sys: queue: (p3 p2 p1 p5 empty) (pc[p1]: ws) (pc[p2]: ws)
 (pc[p3]: ws) (pc[p4]: rs) pc[p5]: ws
===[ rl queue: Q pc[I]: cs => queue: deq(Q) pc[I]: rs [label dq] . ]===>
state 5, Sys: queue: (p2 p3 p1 p5 empty) (pc[p1]: ws) (pc[p2]: ws)
 (pc[p3]: cs) (pc[p4]: rs) pc[p5]: ws
===[ rl queue: (I Q) pc[I]: ws => queue: (I Q) pc[I]: cs [label wt] . ]===>
state 8, Sys: queue: (p2 p1 p5 empty) (pc[p1]: ws) (pc[p2]: ws)
 (pc[p3]: rs) (pc[p4]: rs) pc[p5]: ws
===[ rl queue: Q pc[I]: cs => queue: deq(Q) pc[I]: rs [label dq] . ]===>
state 13, Sys: queue: (p1 p2 p5 empty) (pc[p1]: ws) (pc[p2]: cs)
(pc[p3]: rs) (pc[p4]: rs) pc[p5]: ws
===[ rl queue: (I Q) pc[I]: ws => queue: (I Q) pc[I]: cs [label wt] . ]===>
state 19, Sys: queue: (p1 p5 empty) (pc[p1]: ws) (pc[p2]: rs)
(pc[p3]: rs) (pc[p4]: rs) pc[p5]: ws
===[ rl queue: Q pc[I]: cs => queue: deq(Q) pc[I]: rs [label dq] . ]===>
```

- **LTL Model Checking**

The following LTL search command used for checking the Qlock.

```
red in QLOCK-CHECK : modelCheck(init,lofree) .
```

rewrites: 11279 in 42ms cpu (46ms real) (266863 rewrites/second) result Bool: true

Maude LTL model checker for Qlock did not find any counterexamples. The Qlock protocol likely to enjoy the lockout freedom property.

We suppose that there are five processes p1, p2, p3, p4 and p5 and let init denote the initial state in which the five processes participate in Qlock protocol.

To use Maude LTL model checker, users are supposed to specify atomic propositions. Let us suppose we model check Qlock protocol enjoys the lockout freedom property when there are five processes. The lockout freedom property says whenever each process wants to enter the critical section, it will eventually be there. To express the property in LTL, we need two kinds of atomic propositions **wait(P)** and **crit(P)**, where **P** is a process ID.

Users are also supposed to specify a labeling function. For our purpose, we declare the three equations : **eq(pc[P] : ws) S | = want(P) = true., eq (pc[P] : cs) S | = crip(P) = true., and eq S | = PROP = false** [**owise**] ., where **P** is a Maude variable of process IDs, **S** is a Maude variable of state fragment. The three equations say a state **s** satisfies **want(P)** if and only if **(pc[P] : ws)** appears in **s** and **s** satisfies **crit(P)** if and only if **(pc[P] : cs)** appears in **s**. Then, users are supposed to specify LTL formulas to check. The lockout freedom property is expressed as **want(P)** $\rightsquigarrow$ **crit(P)**, where $\rightsquigarrow$ is the LTL leadsto operator. In Maude, the formula is specified as **eq lpfree = (wait(p1)** $\mapsto$ **crit(p1))** /\ **(wait(p2)** $\mapsto$ **crit(p2))** /\ **(wait(p3)** $\mapsto$ **crit(p3))** /\ **(wait(p4)** $\mapsto$ **crit(p4))** /\ **(wait(p5)** $\mapsto$ **crit(p5))** . , where the operator $\mapsto$ denotes the leadsto operator $\rightsquigarrow$. The model checking is conducted by reducing **modelCheck(init,lofree(p1))**, finding did not any counterexample.

5.3.4 Graphical Animations of Qlock

The Fig. 5.15 to 5.18 show that each state which found by Maude software for Qlock version. We used SMGA for drawing the seven pictures of each state. These pictures make it possible to reorganize at which location of located each process is, what the value stored in the queue.

The contents of an input file that can be fed into the tool are as follows :

```
###keys
queue pc[p1] pc[p2] pc[p3] pc[p4] pc[p5]

###textDisplay
queue::::REV::::_ _

###states
(queue: (empty) (pc[p1]: rs) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]: rs)
(pc[p5]: rs)) ||
(queue: (p5 p4 p3 p2 p1 empty) (pc[p1]: ws) (pc[p2]: ws) (pc[p3]: ws)
(pc[p4]: cs) (pc[p5]: ws)) ||
(queue: (p5 p3 p2 p1 empty) (pc[p1]: ws) (pc[p2]: ws) (pc[p3]: ws)
(pc[p4]: rs) (pc[p5]: ws)) ||
(queue: (p5 p3 p2 p1 empty) (pc[p1]: ws) (pc[p2]: ws) (pc[p3]: cs)
(pc[p4]: rs) (pc[p5]: ws)) ||
(queue: (p5 p1 p2 empty) (pc[p1]: ws) (pc[p2]: ws) (pc[p3]: rs) (pc[p4]: rs)
(pc[p5]: ws)) ||
(queue: (p5 p2 p1 empty) (pc[p1]: ws) (pc[p2]: cs) (pc[p3]: rs) (pc[p4]: rs)
 (pc[p5]: ws)) ||
(queue: (p5 p1 empty) (pc[p1]: ws) (pc[p2]: rs) (pc[p3]: rs) (pc[p4]: rs)
(pc[p5]: ws)
```

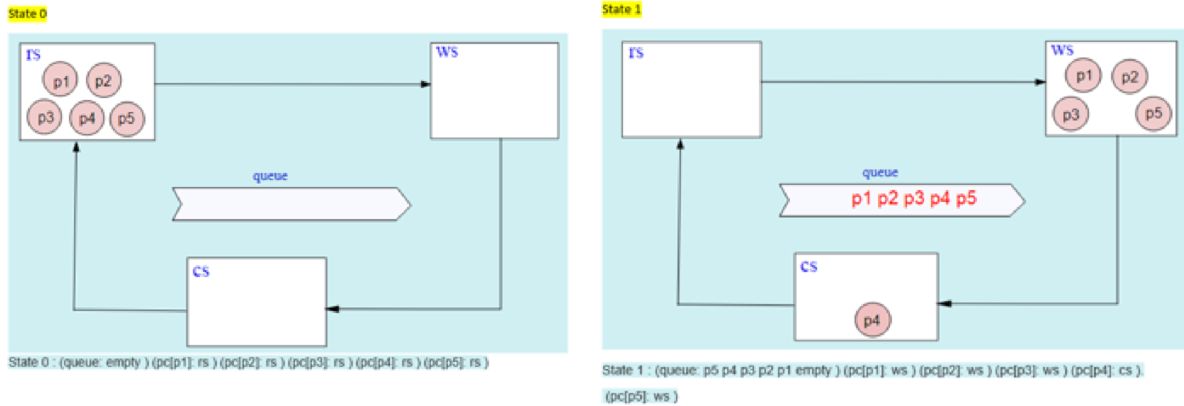

Figure 5.15: States 0 and 1 of QLOCK

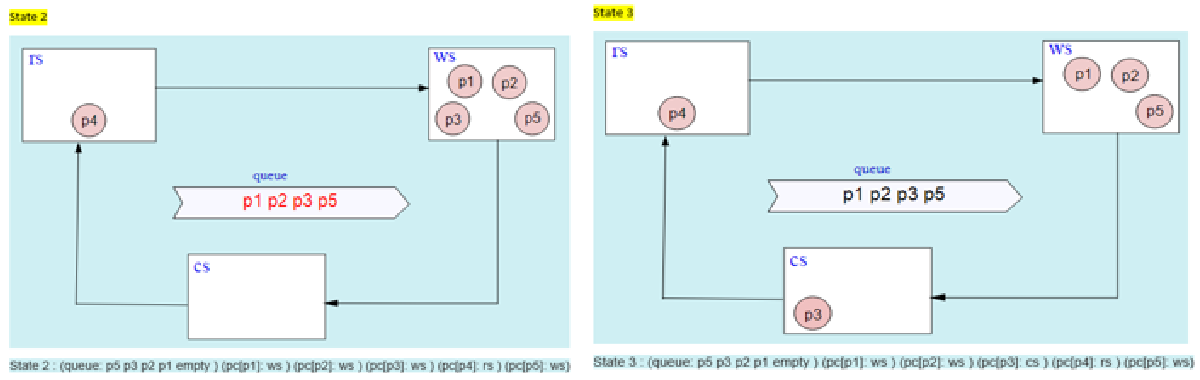

Figure 5.16: States 2 and 3 of QLOCK

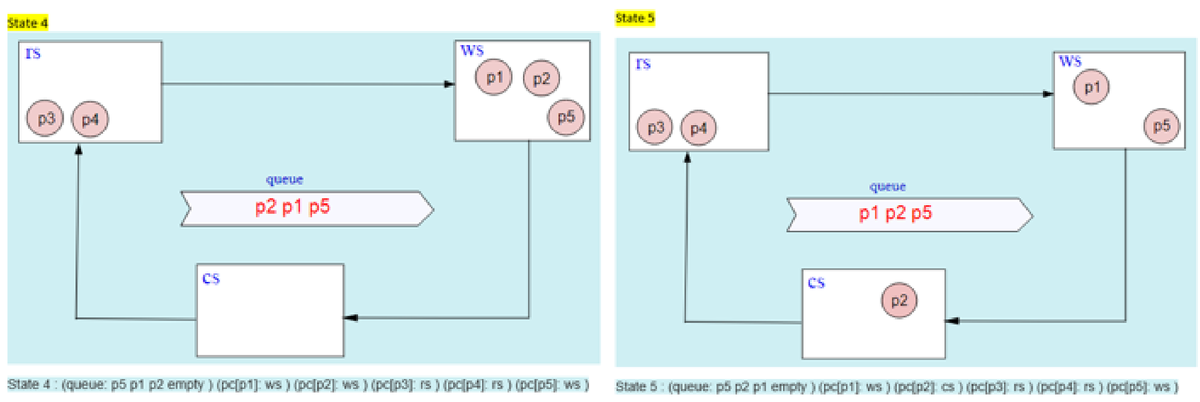

Figure 5.17: States 4 and 5 of QLOCK

5.4 Non-deterministic version of Qlock

The ND-Qlock protocol is a mutual exclusion protocol that uses an atomic queue of process identifiers and can be regarded as an abstract version of the Dijkstra's binary

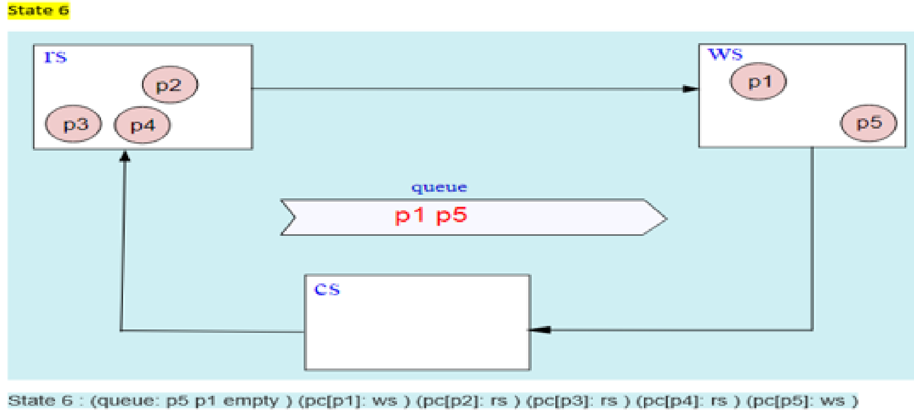

Figure 5.18: State 6 of QLOCK

semaphore.

Where *queue* is an atomic queue of process identifiers shared by all processes and $stm_1 \mid stm_2$ is a non-deterministic choice statement such that either stm_1 or stm_2 is non-deterministically chosen and executed by the process i. This version is called non-deterministic Qlock.

The pseudo-code ND-Qlock for a process i can be described as follows:

Loop: "Remainder Section"
rs: enq(*queue*,i) | **goto** rs;
ws: **repeat until** top (*queue*) = i;
    "Critical Section"
cs: deq (*queue*);

We suppose that each process is at one of the three locations rs (remainder section), ws (waiting section) and cs (critical section). Initially, *queue* is empty and each process is located at rs (remainder section). When the process i is at rs, it can choose one two actions (1) to update *queue* by adding i into it at the end and go to ws, (2) to do something else and stay at rs. When the process is at ws, it waits there until the top of *queue* is i. If the top of *queue* is i, then the process enters the cs (critical section). When the process leaves CS, it deletes the top from *queue* and goes back to rs. We suppose that *queue* and goes back rs. We suppose that *queue* is used in neither Remainder Section nor Critical Section.

5.4.1 ND-Qlock

There are two processes whose are denoted by p1 and p2. Where I and Q are Maude variables of process IDs and queues of process IDs, The four rewrites names are eq (Enqueuing), ds (desequence), wt (waiting), dq (dequeuing) respectively.

The state transitions of ND-Qlock are specified as the following four rewrite rules :

```
rl [eq] : (pc[I]: rs) (queue: Q) => (pc[I]: ws) (queue: enq(Q,I)) .
rl [ds] : (pc[I]: rs)  => (pc[I]: rs) .
rl [wt] : (pc[I]: ws) (queue: (I Q)) => (pc[I]: cs) (queue: (I Q)) .
rl [dq] : (pc[I]: cs) (queue: Q) => (pc[I]: rs) (queue: deq(Q)) .
```

I and Q are Maude variables of process IDs and queues of process IDs, From some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times. eq, ds, wt, and dq are the names of the four rewrite rules, respectively. We suspect that each of enqueuing an element into $q$ueue and the dequeuing $q$ueue is atomic, and so is one iteration of the loop at ws. Initially, the process is located at rs (remainder section) and $q$ueue is empty.

The details description of four rewrite rules follows:

rule 1(eq) : a process I is located at rs, the content of queue is Q. After that a process I is located at ws, the content of queue is enq(Q, I).

rule 2(ds) : From some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times.

rule 3(wt) : a process I is located at ws, the content of queue is (I Q). After that a process I is located at cs, the content of queue is (I Q).

rule 4(dq) : a process I is located at cs, the content of queue is Q. After that a process I is located at rs, the content of queue is deq(Q).

Fig. 5.19 shows the four state transition eq_I, ds_I, wt_I and dq_I respectively. After the transition from one state to another state, we can indicate the process IDs I.

5.4.2 Specification of ND-Qlock as State Machines

Let Pid is the set (or type) of process identifiers, Loc be the set {rs, cs, ws}, of locations, and PidQueue be the set of queues of process identifiers. empty ∈ PidQueue is the empty queue. if $p \in$ Pid and $q \in$ PidQueue, then p|q ∈ PidQueue. The two function enq and deq for PidQueue are defined as follows: for each $q \in$ PidQueue and each p, $p' \in$ Pid, enq(empty,p) = p | empty, enq(p' | q, p) = p' | enq(q, p), deq(empty) = empty, and deq(P | q) = q.

Two kinds of observable components are used:

- $(pc[p_i] : l_p)$ - It says that a process p_i is located at l_p;
- $(queue : q)$ - It says that the content of $queue$ is q;

Where $(pc[p_i])$ is the parametrized name in which $p_i \in$ Pid is a parameter, queue is a name, and $l \in$ Loc and $q \in$ PidQueue are values. We suppose that there are N processes whose identifiers are $p_1, \ldots, p_n \in$ Pid participating in ND-Qlock.

- Set of States, S = {(pc[1] : L_1)...(pc[N] : L_N) (queue : Q)
  | $L_1 \ldots L_N \in$ Loc, Q ∈ PidQueue}.

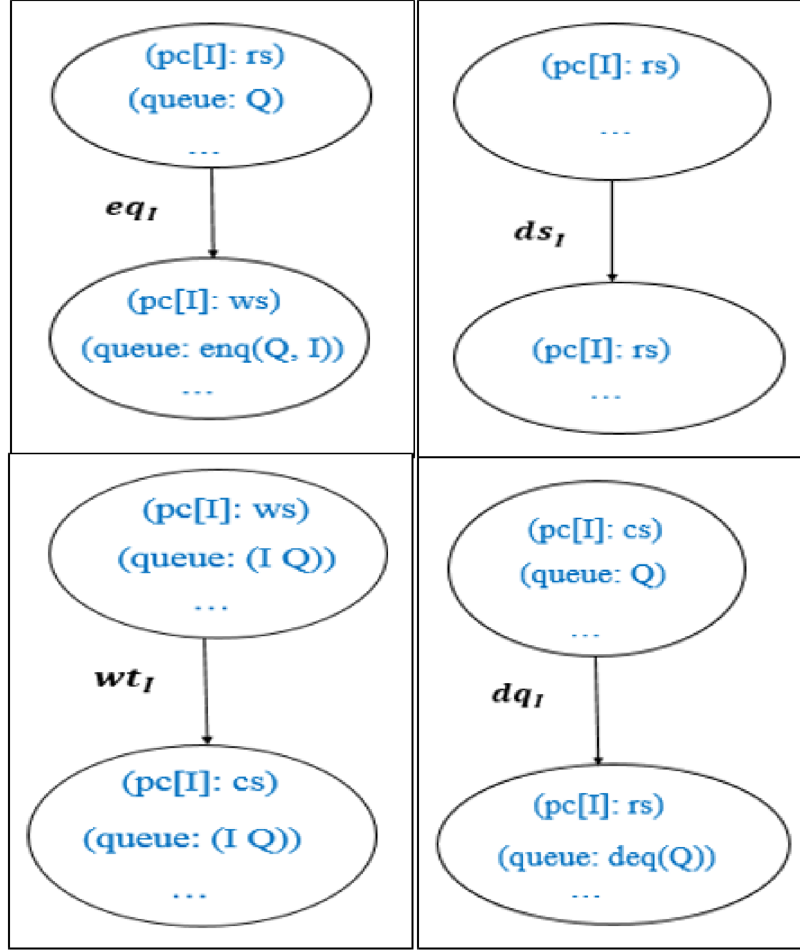

Figure 5.19: State Transition Diagram of ND-QLOCK

- Initial State, I = {(pc[1] : rs) ... (pc[N] : rs) (queue : empty)}.
- T_{eq} = {((pc[1] : L_1) ... (pc[I] : rs)... (pc[N] : L_N) (queue : Q), (pc[1] : L_1)... (pc[I] : ws)... (pc[N] : L_N) (queue : enq(Q, I))) | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Loc, Q ∈ PidQueue}
- T_{ds} = {((pc[1] : L_1) ... (pc[I] : rs)... (pc[N] : L_N) (queue : Q), (pc[1] : L_1)... (pc[I] : rs)... (pc[N] : L_N) (queue : Q)) | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Loc, Q ∈ PidQueue}
- T_{wt} = {((pc[1] : L_1)... (pc[I] : ws)... (pc[N] : L_N) (queue : (I Q)), (pc[1] : L_1)... (pc[I] : cs) ... (pc[N] : L_N) (queue : (I Q))) | I ∈ {1, ..., N}, L_1, ..., L_N ∈ Loc, Q ∈ PidQueue}

- $T_{dq} = \{((\text{pc}[1] : L_1) \ldots (\text{pc}[I] : \text{cs}) \ldots (\text{pc}[N] : L_N)$ (queue : Q),
  $(\text{pc}[1] : L_1) \ldots (\text{pc}[I] : \text{rs}) \ldots (\text{pc}[N] : L_N)$ (queue : deq(Q)))
  $\mid$ I $\in \{1, \ldots, \text{N}\}$, $L_1, \ldots, L_N \in$ Loc, Q $\in$ PidQueue$\}$

5.4.3 Model Checking of ND-Qlock

The following search command used for checking the Ticket.

```
red in NDC-QLOCK-CHECK : modelCheck(init,mutex) .
```

This command verifies that ND-Qlock satisfies the mutex property with the model checker when two processes are involve. The command successfully verifies it.

```
red in NDC-QLOCK-CHECK : modelCheck(init,lofree) .
```

This command find a conterexample showing that ND-Qlock does not satisfy the lockout freedom property with the model checker.

The LTL found the following counter example:

```
counterexample({queue: empty (pc[p1]: rs) pc[p2]: rs,
'eq}, {queue: (p1 empty) (pc[p1]: ws) pc[p2]: rs,'ds})
```

Although p1 is ready to entering the critical section, p2 is always chosen and the rewrite rule ds for p2 is taken, which is not fair for p1.

Chapter 6

Discussion

6.1 Summarized diagram of the report

The picture 6.1 shows that followed diagram of the report what we have done in research project.

- **The need of system specification** : Requirement on a system specification detail, for example, an arrangement of conditions and an arrangement of rewriting rules (transition) with the goal that the system specification determination can be viably executed. For prerequisite on conditions: terminating and confluence. For the necessity of changing principles: admissible and coherence.

- **State transition diagram** : A state diagram comprising of circles to represent to states and guided line portions to represent to transitions between the states. At least one activities (outputs) might be related with each transition. We can see how to change starting with one state then onto the next. The graph represents a finite state machine.

- **State machine mathematical representation** : We represent what is the set of states, initial state, and state transitions. We understood which location is located on each label, where each process located, the value of global and local variable etc.

- **Model checking** : We used Maude search command and LTL model checker to find out the protocol enjoy mutex property and lockout freedom property or not.

- **Graphical Animation by SMGA** : At last, we used SMGA which is a state machine graphical animation tool. We draw each state picture and it also helps to understand counterexamples which are generated by Maude.

6.2 Model checking protocol code testing

Picture 6.2 shows that model checking protocol code testing. First, we make the Maude code based on the specific mutual exclusion protocol version, then we can test the Maude

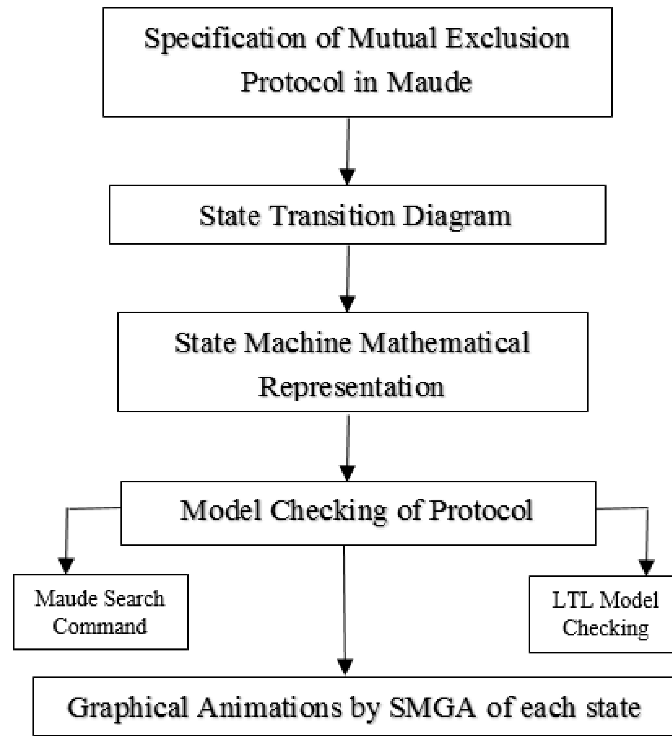

Figure 6.1: Summarize diagram of the project report

code here is any bug or not. If we found any bug then we can check and build it again. We can modeling based on some properties such as Finite-state model extractions, Simplifications, Restrictions. Finally, we can do the model check here is an error or correct. In the event that the model is not right of course, we will re-plan the Maude code. Last (expensive) arrange in the program improvement, Consistency the issue amongst code and model, mostly limited to simplified systems.

6.3 Model checking protocol design testing

Picture 6.3 demonstrates that model checking convention configuration testing. Executable Design Specifications Abstraction from low-level to high-level operations. Displaying Specification of Finite-state demonstrates extraction. Confirmation where utilized State space lessening procedure. There are a few focal points: Applied prior in the outline cycle (Earlier bug discovery) The immediate interpretation of casual program into formal punctuation (no disentanglements), Separation of concerns: the reflection of control from data, Space particular property determination. We can model testing by Maude then understand if any counterexamples happened or true. If the protocol has any counterexamples then we can correct it.

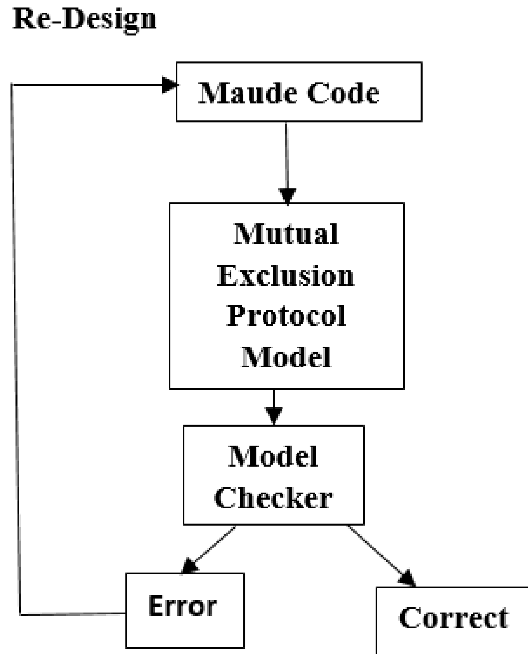

Figure 6.2: Model checking protocol code testing

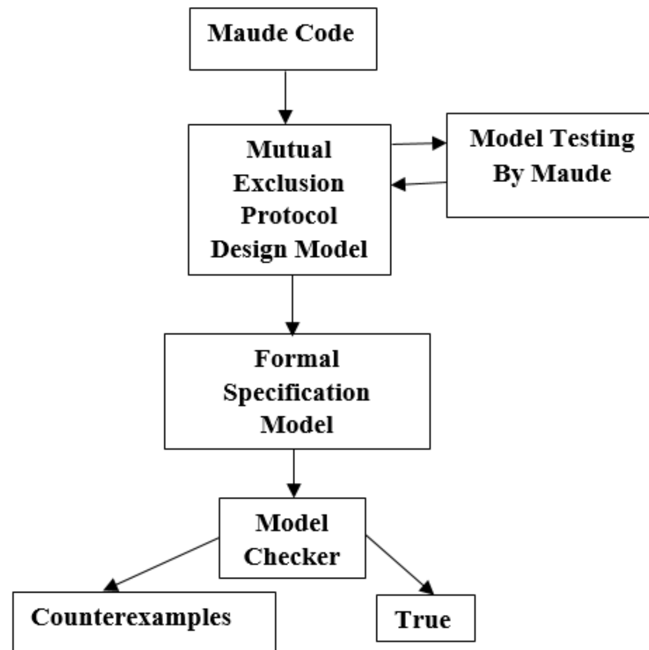

Figure 6.3: Model checking protocol design testing

Chapter 7

Conclusion

We have analysis, mathematical formalize as a state machine, describe state machines in a formal specification by Maude, model check that state enjoys properties based on such formal specification. We have developed state pictures, which graphical animations generated by SMGA make it possible for us to quickly recognize each state picture.

We used some mutual exclusion protocol, such as FTicket, Ticket, ND- Ticket, FAnderson, Anderson, ND- Anderson, FQlock0, FQlock1, Qlock, ND-Qlock as concrete examples to conduct the project report.

Where FTicket does not enjoy the mutex property, while Ticket likely enjoys the mutex property. Where FAnderson does not enjoy the mutex property, while Anderson likely enjoys the mutex property. We have also analyzed two flawed versions FQlock0 and FQlock1 of Qlock, a mutual exclusion (mutex) protocol with Maude and SMGA. Where FQlock0 does not enjoy the mutex property, while FQlock1 does but does not enjoy the lockout freedom property. We have reported on a case study in which we guessed properties of Qlock, a mutex protocol, by observing a graphical animation of Qlock displayed by SMGA. We used Maude search command and the Maude LTL model checker.

We can observe from the Fig. 7.1 the summarize results of all protocols which used in the project report. We understood why the FTicket does not enjoy mutext property but why the Ticket enjoys mutex property. The reason, we used fetch&incmode atomic operation to implement the Ticket protocol but did not use for FTicket protocol. This can atomically read a memory location, increment the value modulo N processes, then writes the result into the memory location and return the old value. This is the main point why Ticket enjoy mutex property and lockout freedom property. But in the case for FTicket we got counterexamples, then produced pictures of each state by SMGA, which make it possible feasible for us to rapidly reorganize the reason why FTicket does not enjoy the mutex property.

On the other hand, ND-Ticket enjoys the mutex property but does not enjoy lockout freedom property. If we compare with Ticket protocol then remark that there used another rewrite rule which said that, from some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times. For the argument, we have taken counterexamples by Maude LTL model checker.

In the case of FAnderson and Anderson protocol, there used one conditional rule for execution of the protocols. There also used Boolean which describes one of two values: true or false. We realized that why the FAnderson does not enjoy mutext property but why the Anderson enjoys mutex property. The cause, we used fetch&incmode atomic operation to execute the Anderson protocol but it did not use for FAnderson protocol. This can atomically read a memory location, increment the value modulo N processes, then writes the result into the memory location and return the old value. When a process i tries to enter the critical section, it indivisibly copies into it local variable place and increments next remainder N using fetch&incmode.

Moreover, ND-Anderson enjoys the mutex property but does not enjoy lockout freedom property. If we comparison with Anderson protocol then the observation that there used another new rewrite rule which said that, from some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times. For the reason, we have received counterexamples by Maude LTL model checker.

As opposed to, we realized why the FQlock0 does not enjoy mutex property but why the FQlock1 enjoys mutex property. The reason, we used in FQlock1 is that it is atomic to enqueue a process ID i into the queue in FQlock1, while it is not in FQlock0. But in FQlock0, it is not atomic to enqueue a process ID i into the queue and it is not atomic to dequeue queue. This is the main condition why FQlock1 enjoy mutex property but FQlock0 does not enjoy mutex property. But in the case for FQlock1 we got counterexamples for Maude LTL model checking, then produced pictures of each state by SMGA, which make it possible feasible for us to rapidly reorganize the reason why FQlock1 does not enjoy the lockout property.

In addition, Qlock enjoys the mutex property and enjoy lockout freedom property. We have indicated on some case study in which we guessed properties of Qlock, a mutual protocol, by observing a graphical animation of Qlock displayed by SMGA, we can confirm the guessed properties by model checking the properties with the Maude search command and also checked by Maude LTL model checker. If we compare with ND-Qlock protocol then the observation that there used another new rewrite rule which said that, from some time on, a process may never try to enter the critical section but keep on staying at rs, or equivalently it may try to enter the critical section a finitely many times. For the reason, we have received counterexamples for ND-Qlock by Maude LTL model checker.

There are a few suppositions from the mutual exclusion protocol, numerous procedures are calculated for the common equipment, however, at any snapshot of time just a single procedure can utilize the equipment, That is, the procedures are mutually excluded from utilizing the equipment. The mutex property is that at most one process is the critical section in any reachable states. common search navigates the reachable states from an offered state to discover states with the end goal that some condition hold. The charge can be utilized to discover the convention fulfills the property or not. In the event that we discovered counterexamples then the property does not enjoy the property.

We used the number of processes N = 2 for model checking of some protocol such as

Protocol Version	State Transition Number	Number of States	Maude Search command	LTL Model Checker	Remarks
FTicket	4	7	Solution		Maude found **solution** meaning FTicket does not enjoy the mutex property.
Ticket	3	13	No Solution	True	**No solution** meaning Ticket enjoy the mutex property. Also enjoy the lockout freedom property.
ND-Ticket	4	2	No Solution	Counter examples	**No solution** meaning ND-Ticket enjoy the mutex property. But does not enjoy the lockout freedom property.
FAnderson	4	7	Solution		Maude found **solution** meaning FAnderson does not enjoy the mutex property.
Anderson	3	13	No Solution	True	**No solution** meaning Anderson enjoy the mutex property. Also enjoy the lockout freedom property.
ND-Anderson	4	2	No Solution	Counter examples	**No solution** meaning ND-Anderson enjoy the mutex property. Does not enjoy the lockout freedom property.
FQlock0	5	7	Solution		Maude found **solution** meaning FQlock0 does not enjoy the mutex property.
FQlock1	4	13	No Solution	Counter examples	**No solution** meaning FQlock1 enjoy the mutex property. Does not enjoy the lockout freedom property.
Qlock	3	6	No Solution	True	**No solution** meaning Qlock enjoy the mutex property. Also enjoy the lockout freedom property.
ND-Qlock	4	2	No Solution	Counter examples	**No solution** meaning ND-Qlock enjoy the mutex property. Does not enjoy the lockout freedom property.

Figure 7.1: Summarize results of all protocols

Ticket, Anderson mutual exclusion protocol. If we increased the processes N value then observed that the number of states also increased. The values changes in each state are shown in red color in the state picture by SMGA.

Bibliography

[1] : https://www.techopedia.com/definition/16447/state-machine, (techopedia)

[2] Tam Thi Thanh Nguyen, Kazuhiro Ogata: Graphical animations of state machines. (In: 15th IEEE International Conference on Dependable, Autonomic and Secure Computing (15th DASC), IEEE) To appear.

[3] M. Clavel, F. Duran, S.Ekar, P.Lincoln, N. Mart-Oliet, J. Meseguer, and C. Talcott.: All about maude. In: LNCS 4350., Springer, (2007) DOI=https://doi.org/10.1007/978–3–540–71999–1.

[4] : (https://en.wikipedia.org/wiki/mutual_exclusion)

[5] Kazuhiro Ogata: i613 algebraic formal methods. In: Term 2-2 course at JAIST, Japan. (2017.)

[6] : (https://en.wikipedia.org/wiki/finite-state_machine)

[7] Joseph A. Goguen, Timothy Winkler, Jos Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud: (Introducing obj,)

[8] J. Liard: Draw svg website. In: http://www.drawsvg.org/. (2015.)

[9] T. T. T. Nguyen and K. Ogata: Graphical animations of state machines. In: in 15th IEEE DASC. IEEE, 2017. (2017)

[10] Kazuhiro Ogata, Kokichi Futatsugi: (Specification and verification of some classical mutual exclusion algorithms with cafeobj)

[11] May Thu Aung, Tam Thi Than Nguyen, K.O.: Analysis of two flawed versions of a mutual exclusion protocol with Maude and SMGA. In: 7th International Conference on Software and Computer Applications (ICSA 2018), ACM (2018) to appear.

[12] Tam Thi Than Nguyen, Kazuhiro Ogata,: A way to comprehend counterexamples generated by the maude ltl model checker. In: 2017 International Conference on Software Analysis, Testing and Evolution, IEEE (2017) DOI 10.1109/SATE.2017.15.

[13] May Thu Aung, Tam Thi Than Nguyen, K.O.: Guessing properties of the Qlock mutual exclustion protocol based on its graphical animations and confirming the properties by model checking. In: 7th International Conference on Software and Computer Applications (ICSA 2018), ACM (2018) to appear.