Graphical Animations of State Machines

Nguyen Thi Thanh Tam

School of Information Science
Japan Advanced Institute of Science and Technology
September, 2017

Master's Research Project Report

Graphical Animations of State Machines

1510205 Nguyen Thi Thanh Tam

Supervisor : Professor Kazuhiro Ogata
Main Examiner : Professor Kazuhiro Ogata
Examiners : Professor Kunihiko Hiraishi
Professor Toshiaki Aoki

School of Information Science
Japan Advanced Institute of Science and Technology

August, 2017

Contents

Acknowledgements

This master research report would not have been possible without the guidance and the help of my supervisor Professor Kazuhiro Ogata. I would like to express my sincere gratitude to him for the continous support of not only my research but also my life.

My sincere thanks also goes to Joseph Liard for creating the drawSVG tool which is a free online drawing application for designers and developers. The tool has assist me to implement my idea in this research.

Last but not least, I would like to thank my parents and husband for their unending support and encouragement all the time.

To all those people, this thesis is dedicated.

List of Figures

Chapter 1

Introduction

1.1 Overview

The world crucially depends on software. It would be impossible to even imagine our lives without use of any software. The societal reliability is almost the same as that of software. How much human beings rely on software must be increasing in the future. Therefore, we need to have reliable technologies to make software truly reliable. A possibly promising technology is systems verification with interactive theorem proving (ITP). Hence, many proof assistants have been developed, such as PVS [19], ACL2 [1], Isabelle [10], and Coq [3]. One of the most intellectual activities in interactive theorem proving is lemma conjecture. Accordingly, many researches have been conducted, trying to come up with how to conjecture lemmas. None of them, however, is good enough. Thus, we need to make further efforts to come up with a better way to do so.

Various kinds of systems can be formalized as state machines. A state machine $M \triangleq \langle S, I, T \rangle$ consists of a set S of states including the set I of initial states and a binary relation $T \subseteq S \times S$ over states. An element $(s, s') \in T$ is called a state transition of M. The set R_M of the reachable states of M is inductively defined as follows: $I \subseteq R_M$ and if $s \in R_M$ and $(s, s') \in T$, then $s' \in R_M$. A state predicate p is called an invariant of M if and only if $(\forall s \in R_M)\, p(s)$. Many requirements of software can be formalized as invariants. Since verifications of other classes of properties often require invariants as lemmas, invariants are the most fundamental class of properties of state machines. $s_0, s_1, \ldots, s_n$ is called a finite computation of M if and only if $s_0 \in I$ and $(\forall i \in \{0, \ldots, n-1\})\, (s_i, s_{i+1}) \in T$. Note that each state in any finite computations of M is a reachable state of M.

1.2 The problems and solutions

While Ogata was formally verifying with interactive theorem proving that a state machine formalizing a communication protocol enjoys an invariant, he happened to find out that the reachable states of the state machine are classified into six state patterns. From the six state patterns, we conjectured several useful lemmas that are also invariants to complete the formal verification [18]. Although the six state patterns are very useful for conjecturing

lemmas, it took time to obtain those six state patterns. This might be because obtaining such state patterns of a state machine is almost equivalent to conjecturing lemmas or invariants of the state machine. We would like to obtain such state patterns of a given state machine with a reasonable amount of efforts. We utilize human beings' ability to recognize patterns in various kinds of data including graphical animations. We believe that if human beings carefully watch graphical animations of finite computations of a given state machine, they can recognize patterns. Besides that, the reachable states of the state machine can be classified into patterns because finite computations of the state machine consist of reachable states of the state machine. Thus, we would like to develop a state machine graphical animation tool to support human users find out interesting state patterns from which they could figure out useful lemmas used for theorem proving.

On the other hand, model checking is one of the most popular automated verification techniques, and many model checkers have been developed. Among them are SMV [16], Spin [9], SAL [4], PAT [22] and the Maude LTL model checker [21], where LTL stands for linear temporal logic. Specifications in which associative and/or commutative binary operators are used can be model checked by the Maude LTL model checker, which also deals with inductively defined data structures, while SMV, Spin, SAL and PAT cannot. A counterexample generated by Maude LTL model checker if any, consists of a sequence $s_0; \ldots; s_m$ of states and a loop $(s_{m+1}; \ldots; s_n)^\infty$ of states such that s_{m+1} is a successor state of s_m and s_n. Since the loop is repeatedly played, the repetition could help human users realize what happen in the loop. It would be preferable to graphically animate a counterexample so that human users could comprehend it better. Thus, we can also use the state machine graphical animation tool help human users comprehend a counterexample.

1.3 Goal and Contribution

The research aims at designing and implementing a state machine graphical animation tool. The main motivation for the tool is to support users get better understanding state machines, recognize patterns to be used for conjecturing lemmas in interactive theorem proving, and comprehend counterexamples better.

Since the tool support the users find out such state patterns of a given state machine with a reasonable amount of effort to help them in conjecturing useful lemmas, the research could give non-trivial contributions to systems verification based on interactive theorem proving. Besides that, the tool could also help human get better understandings of counterexamples and realize why such them occur to improve and enhance systems. Thus, the research is not only help human users get better understanding state machines and systems but also clear more about counterexample and ensure the quality of software.

1.4 Report Outline

The rest of the research project report is organized as follows:

- Chapter 2: Preliminaries

  This chapter presents some preliminaries, such as Alternating Bit Protocol (ABP) that is used as a running example, Maude that is the rewritting logic-based specification or programming language, how to formalize ABP as state machine in Maude, how to express states of a state machine, the Kripke structrue and the Maude LTL model checker.

- Chapter 3: Motivating Example

  This chapter presents the motivating example for the research.

- Chapter 4: Design and Implementation

  This chapter describes the design and implementation of the tool.

- Chapter 5: Generation of Long Computations

  This chapter describes a way to generate long computations which are useful for the running tool to help users be able to recognize patterns in them.

- Chapter 6: Experiment

  This chapter reports on an experiment done with the tool.

- Chapter 7: Applications

  This chapter presents some applications of the tool such as comprehending counterexamples generated by the Maude LTL model checker, and analyzing MCS list-based queuing lock and some variants with the combination of Maude and the animation tool.

- Chapter 8: Related Work

  This chapter mentions some existing related works.

- Chapter 9: Future Work

  This chapter discusses some future work.

- Chapter 10: Conclusion

  This chapter concludes the research project.

Chapter 2

Preliminaries

2.1 Alternating Bit Protocol

Alternating Bit Protocol (ABP) is a communication protocol such that the window size is 1 in Sliding Window Protocol used in Transmission Control Protocol (TCP), the most important communication protocol on the globe. ABP consists of a sender and a receiver. The sender maintains one bit $bit1$ and a packet pac to be delivered. The receiver maintains one bit $bit2$ and a list $list$ that contains the packets that have been received. Two unreliable channels $chan1$ and $chan2$ are used. Since they are unreliable channels, their elements may be lost (or dropped) and duplicated. Fig. 2.1 shows a snapshot of ABP. There are eight possible actions in ABP:

- send1: The sender puts a pair $\langle bit1, pac \rangle$ into $chan1$.
- rec1: The sender gets the top element Boolean b from $chan2$. If $b \neq bit1$, $bit1$ is complemented and pac is incremented.
- send2: The receiver puts $bit2$ into $chan2$.
- rec2: The receiver gets the top element $\langle b, p \rangle$ from $chan1$ if $chan1$ is not empty. If $b = bit2$, $bit2$ is complemented and p is added to $list$.
- drop1: The top of $chan1$ is deleted if it is not empty.
- dup1: The top of $chan1$ is duplicated if it is not empty.
- drop2: The top of $chan2$ is deleted if it is not empty.
- dup2: The top of $chan2$ is duplicated if it is not empty.

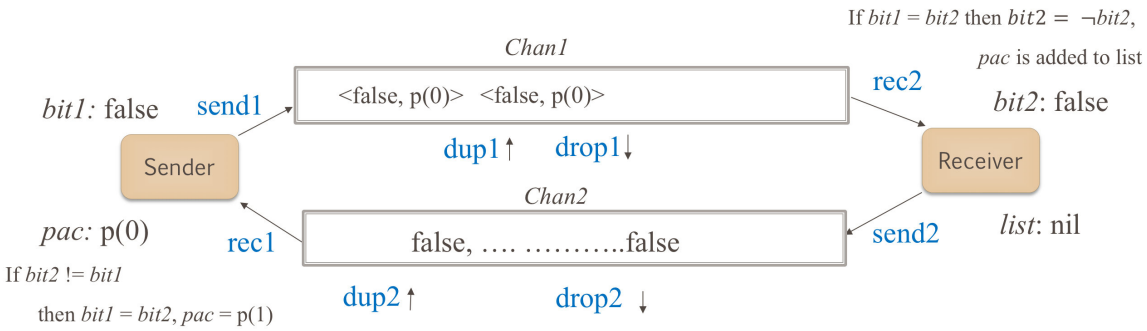

Figure 2.1: A snapshot of ABP

2.2 Maude

Maude [2] is a rewriting logic-based computer language and system. It is one of the direct successors of OBJ3 [5], the most famous algebraic specification language and system mainly designed by Joseph A. Goguen. Specifications can be written in Maude flexibly. Associative and/or commutative binary operators can be freely used in specifications and then complex concurrent and distributed systems can be succinctly specified.

Maude is equipped with many functionalities, among which are model checking (the Maude search command and the Maude LTL model checker), and meta-programming. A metaprogram is a program that takes programs as inputs and performs some useful computations. Besides that, Maude is equipped with the search command that exhaustively traverses the reachable states from a given state to find states that match some pattern and satisfy some condition in a breadth-first manner. It is also equipped with a metalevel function that is the counterpart of the search command.

For a state machine specification in Maude, a state `S`, a pattern `P` and a condition `C`, the Maude search command exhaustively traverses the reachable states from `S` to find states that match `P` and satisfy `C`:

```
search [N] in Mod : S =>* P such that C .
```

where `N` is a natural number. The search command tries to find at most N solutions. Note that a solution is basically a state `A` that matches `P` and satisfies `C`, but since there may be more than one substitution σ such that σ(`P`) = `A`, there may be more solutions than the number of such states and such substitutions are called solutions of the search.

As an example, ABP can be formalized as a state machine M_{ABP} by using Maude. T_{ABP} is described as the eight rewrite rules. The state transitions of ABP are specified as follows:

```
crl [send1] : (chan1: PC) (bit1: B1) (pac: P) => (chan1: (PC < B1,P >))
(bit1: B1) (pac: P) if len(PC) < Len /\ ord(P) < NoP .
rl [rec1] : (chan2: (B BC)) (bit1: B1) (pac: P)
=> (chan2: BC) (bit1: (if B1 == B then B1 else not B1 fi))
```

```
(pac: (if B1 == B then P else next(P) fi)) .
crl [send2] : (chan2: BC) (bit2: B2) => (chan2: (BC B2)) (bit2: B2)
if len(BC) < Len .
rl [rec2] : (chan1: (< B,P >  PC)) (bit2: B2) (list: L)
=> (chan1: PC) (bit2: (if B2 == B then not B2 else B2 fi))
(list: (if B2 == B then (P L) else L fi)) .
rl [drop1] : (chan1: (PC1 BP PC2)) => (chan1: (PC1 PC2)) .
rl [drop2] : (chan2: (BC1 B BC2)) => (chan2: (BC1 BC2)) .
crl [dup1] : (chan1: (PC1 BP PC2)) => (chan1: (PC1 BP BP PC2))
if len(PC1 BP PC2) < Len .
crl [dup2] : (chan2: (BC1 B BC2)) => (chan2: (BC1 B B BC2))
if len(BC1 B BC2) < Len .
```

where `PC`, `PC1` and `PC2` are Maude variables of Boolean-packet pair queues, `BC`, `BC1` and `BC2` are ones of Boolean queues, `B`, `B1` and `B2` are ones of Booleans, `P` is one of packets, and `Len` and `NoP` are natural numbers. The function `len` takes a queue and returns the number of its elements. And the function `ord` takes a packet `pac(n)`, where `n` is a natural number, and returns `n` as an ordinal of the packet. `send1`, `rec1`, etc are the labels of the rewrite rules.

2.3 State Expression

States can be expressed in various ways. In this research, a state is expressed as an associative-commutative collection of name-value pairs such as: $(name1 : value1)$ $(name2 : value2)$ ....
Name-value pairs are called observable components and associative-commutative collections are called soups. Thus, a state is expressed as a soup of observable components.

Each state of M_{ABP} is characterized by the six values as shown in Fig. 2.1. ABP formalized as a state machine whose states are expressed as soups of observable components. Therefore, each state of M_{ABP} is expressed as following:
`(chan1: prq) (chan2: bq) (bit1: b1) (bit2: b2) (pac: p) (list: ps)`
where `prq` is a queue of Boolean-packet pairs, `bq` is a queue of Booleans, `b1` is a Boolean, `b2` is a Boolean, `p` is a packet, and `ps` is a list of packets. For example, `chan1` is a name, `prq` is a value, and `(chan1: prq)` is an observable component.

Since `(chan1: prq) (chan2: bq) (bit1: b1) (bit2: b2) (pac: p) (list: ps)` is a soup of observable components, even if the order in which the observable components appear is changed, such as `(chan2: bq) (bit1: b1) (chan1: prq) (bit2: b2) (pac: p) (list: ps)`, it represents the same state. The initial state of M_{ABP} is expressed as follows: `(chan1: empty) (chan2: empty) (bit1: false) (bit2: false) (pac: pac(0)) (list: nil)`.

2.4 Kripke Structure and Maude LTL model checker

Let S be a set and π be an infinite sequence $e_0; ...; e_i; \ldots$ of S, where each $e_i \in S$, and then $\pi(i) = e_i$ (the ith element in π) and $\pi^i = e_i; \ldots$ (the ith suffix obtained by deleting the first i elements from π) for each natural number i. Let $e_0; ...; e_n$ be a non-empty finite sequence of S, and then $(e_0; ...; e_n)^\infty = e_0; ...; e_n; e_0; ...; e_n; \ldots$ (the infinite sequence in which the finite sequence repeats infinitely often). Let U be a universal set of symbols.

A Kripke structure (KS) K is a 5 tuple $\langle S, I, P, L, T\rangle$, where S is a set of states, $I \subseteq S$ is the set of initial states, $P \subseteq U$ is a set of atomic state propositions, L is a labeling function whose type is $S \to 2^P$, and $T \subseteq S \times S$ is a total binary relation. An element $(s, s') \in T$ may be written as $s \to s'$ and referred as a state transition.

A path of K is an infinite sequence $s_0; \ldots; s_i; s_{i+1}; \ldots$ of S such that $(s_i, s_{i+1}) \in T$ for each natural number i. A computation of K is a path π of K such that $\pi(0) \in I$. Let $\mathcal{P}$ be the set of all paths of K, and $\mathcal{C}$ be the set of all computations of K. A finite prefix $s_0; \ldots; s_n$ of a computation (or path) of K is called a finite computation (or path) of K. The syntax of a formula φ in Linear Temporal Logic (LTL) for K is $\varphi ::= \top \mid p \mid \varphi \wedge \varphi \mid \bigcirc \varphi \mid \varphi \,\mathcal{U}\, \varphi$, where $p \in P$. Let $\mathcal{F}$ be the set of all formulas in LTL for K.

An arbitrary path $\pi \in \mathcal{P}$ of K and an arbitrary LTL formula $\varphi \in \mathcal{F}$ of K, $K, \pi \models \varphi$ is inductively defined as $K, \pi \models \top$, $K, \pi \models p$ if and only if $p \in L(\pi(0))$, $K, \pi \models \neg\varphi_1$ if and only if $K, \pi \not\models \varphi_1$, $K, \pi \models \varphi_1 \wedge \varphi_2$ if and only if $K, \pi \models \varphi_1$ and $K, \pi \models \varphi_2$, $K, \pi \models \bigcirc \varphi_1$ if and only if $K, \pi^1 \models \varphi_1$, and $K, \pi \models \varphi_1 \,\mathcal{U}\, \varphi_2$ if and only if there exists a natural number i such that $K, \pi^i \models \varphi_2$ and for all natural numbers $j < i$, $K, \pi^j \models \varphi_1$, where φ_1 and φ_2 are LTL formulas. Then, $K \models \varphi$ if and only if $K, \pi \models \varphi$ for each computation $\pi \in \mathcal{C}$ of K.

The temporal connectives $\bigcirc$ and $\mathcal{U}$ are called the next operator and the until operator, respectively. The other logical and temporal connectives are defined as usual as follows: $\bot \triangleq \neg\top$, $\varphi_1 \vee \varphi_2 \triangleq \neg(\neg\varphi_1 \wedge \neg\varphi_2)$, $\varphi_1 \Rightarrow \varphi_2 \triangleq \neg\varphi_1 \vee \varphi_2$, $\Diamond\, \varphi \triangleq \top \,\mathcal{U}\, \varphi$, $\Box\, \varphi \triangleq \neg(\Diamond\, \neg\varphi)$, and $\varphi_1 \rightsquigarrow \varphi_2 \triangleq \Box\,(\varphi_1 \Rightarrow \Diamond\, \varphi_2)$. The temporal connectives $\Diamond$, $\Box$ and $\rightsquigarrow$ are called the eventually operator, the always operator and the leadsto operator, respectively.

For a state machine (precisely a Kripke structure) M and an LTL formula φ, Maude LTL model checker checks if M satisfies φ. If M does not satisfy φ, it generates a counterexample that consists of a sequence $s_0; \ldots; s_m$ of states of M and a loop $(s_{m+1}; \ldots; s_n)^\infty$ of states of M such that for $i = 0, 1, ...n-1$ (s_i, s_{i+1}) is a state transition of M and so is (s_n, s_{m+1}).

Chapter 3

Motivating Example

When Ogata was formally verifying that ABP satisfies a desired property, he found that $R_{M_{\mathrm{ABP}}}$ is classified into six patterns shown in Fig. 3.1. From the six state patterns, we were able to conjecture several useful lemmas to complete the formal verification. For example, SP3 allows us to conjecture the following lemma:

> if $chan2$ contains two Booleans $b1$ and $b2$ in a raw such that $b1 \neq b2$ and $b1$ is closer to the top, then each Boolean b appearing in $chan2$ later than $b2$ is the same as $b2$ and $b2$ is the same as $bit2$;

and SP6 allows us to conjecture the following lemma:

> if $chan1$ contains two pairs $\langle b1, p1\rangle$ and $\langle b2, p2\rangle$ in a raw such that $\langle b1, p1\rangle \neq \langle b2, p2\rangle$ and $\langle b1, p1\rangle$ is closer to the top, then each pair $\langle b, p\rangle$ appearing in $chan1$ later than $\langle b2, p2\rangle$ is the same as $\langle b2, p2\rangle$ and $\langle b2, p2\rangle$ is the same as $\langle bit1, pac\rangle$.

If it is possible to find out such state patterns of a given state machine with a reasonable amount of effort, this could give non-trivial contributions to systems verification based on interactive theorem proving because such state patterns help human users conjecture useful lemmas.

Human beings are very good at recognizing patterns in various kinds of data, such as sounds, still images, and graphical animations. If human beings carefully watch graphical animations of finite computations of a state machine, they could recognize underlying patterns from which they could conjecture useful lemmas. It would require much fewer efforts and less time to watch graphical animations of finite computations of a state machine M than to try to formally prove that M enjoys invariants so as to conjecture lemmas. This has motivated us to develop the state machine graphical animation tool. We do not try to create anything that imitates human beings' ability to recognize patterns but try to make the best use of this ability so as to conjecture lemmas in this research[1].

[1]Our group has also been attempting [8] to automatically extract state patterns of a given state machine with Inductive Logic Programming, a combination of machine learning and logic programming.

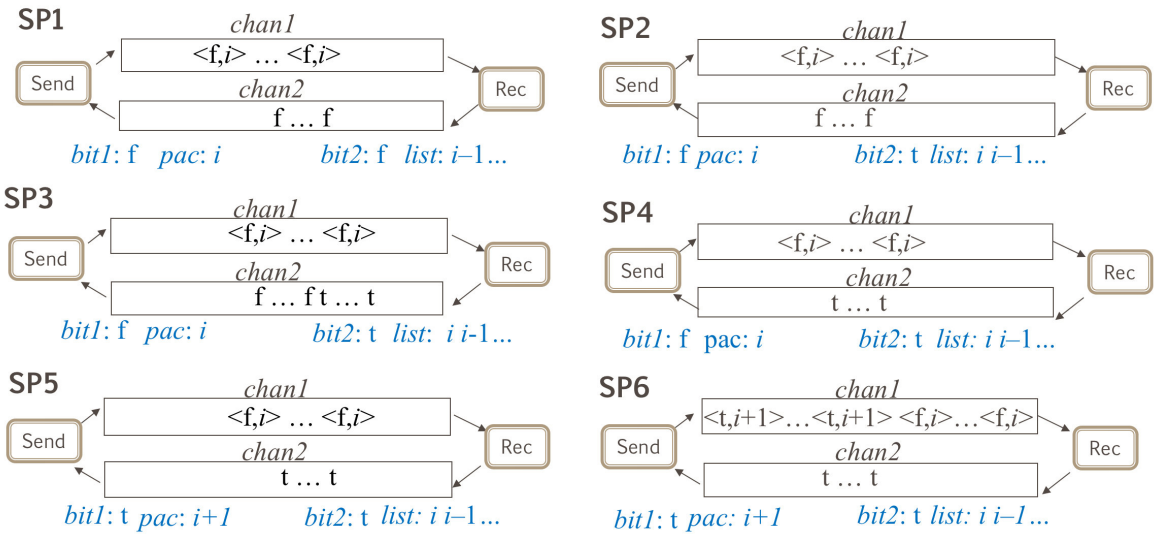

Figure 3.1: Six state patterns of $R_{M_{\mathrm{ABP}}}$

Chapter 4

Design and Implementation

4.1 Design

If the state machine graphical animation tool deals with state machines internally, we need to design an internal representation of state machines or adopt some existing ones. It would be clumsy to ask human users to write state machines in such an internal representation. Consequently, we need to design a specification language for state machines or adopt some existing ones. If so, it would be necessary to translate state machines written in a specification language into those written in an internal representation. We should develop multiple translators for multiple specification languages to make it possible for any state machines to be graphically animated. Since many specification languages have been and would be proposed, however, it would not be smart to develop a translator for each specification language because it is not a trivial task to develop even one translator for one specification language.

We have not designed the state machine graphical animation tool such that it deals with state machines internally but designed it such that it basically takes a finite computation of a state machine. This is because tools, such as model checkers, that can deal with state machines can generate finite computations of state machines. We need to fix how to represent each state of state machines and finite sequences of states. It would be much easier, however, to transform some different state representations to that used for the state machine graphical animation tool than to translate state machines written in a specification language into those written in another one. Besides, it would be straightforward to transform some different representations of finite state sequences to that used for the state machine graphical animation tool once different state representations have been transformed into that used for the tool.

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.

It would be possible to make a clear correspondence between term (or text) state representations and graphical state representations. This correspondence is treated as

part of the input data to the state machine graphical animation tool, together with a finite computation of a state machine. Although human users are supposed to write such a correspondence, we do not think that this is a non-trivial piece of code (or programs).

In our design of the state machine graphical animation tool, a finite computation of a state machine can be regarded as a film. Accordingly, the speed of the animation can be adjusted by changing (redrawing) the current state to the successor state in a specified amount of time, such as 10ms and 50ms.

If we try to generate all finite computations whose length is some specific bound and the bound is large enough, we quickly encounter the notorious state explosion problem. If the number of packets to be delivered is 10 and the capacity of each channel is 10, then the Maude search command could exhaustively traverse $R_{M_{\mathrm{ABP}}}$ up to depth 37 but encountered the state explosion problem when the depth was 38. It would be unnecessary to generate all finite computations up to some shallow depth but necessary to generate some long finite computations. It would be inadequate to generate computations in which some specific state transitions are only taken. We will describe how to generate adequate long computations later.

4.2 Implementation

4.2.1 The structure of an input file

The graphical animation tool does not deal with state machines themselves internally. Instead, what is fed into the tool is basically a finite computation of a state machine. The input file format is described.

An example input file of M_{ABP} is as follows:

```
###keys
chan1 chan2 bit1 bit2 pac list
###textDisplay
chan1::::REV::::<_,_>++++empty
###states
(chan1: empty chan2: empty bit1: false bit2: false pac: pac(0) list: nil) ||
(chan1: (< false,pac(0) > empty) chan2: empty bit1: false bit2: false pac: pac(0)
list: nil) || (chan1: empty chan2: empty bit1: false bit2: true pac: pac(0)
list: (pac(0) nil)) || (chan1: empty chan2: (true empty) bit1: false bit2: true
pac: pac(0) list: (pac(0) nil)) || chan1: empty chan2: empty bit1: true
bit2: true pac: pac(1) list: (pac(0) nil)
```

There are three segments in an input file as follows:

- 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. For example, The value of $(chan1 : prq)$ should be displayed such that its top appears right most. The format used in this part is as follows:

```
key::::option:::regex(0)++++....++++regex(i)
```

The format consists of three parts: key, option and regexs. A key appearing in the key segment is written in the key part. REV, VER or VER-REV is written in the option part. REV specifies a collection, such as queues and lists, is displayed such that its top appears right most, VER specifies a collection, such as stacks, is displayed vertically such that its top appears top most, and VER-REV specifies a collection is displayed vertically such that its top appears bottom most. A list of regular expressions is written in the regexs part. For example, the textDisplay segment of M_{ABP} is as follows:

```
chan1::::REV::::<_,_>++++empty
```

Two regular expressions `<_,_>` and `empty` are written in the regexs part. They match texts, such as `<false,p(0)>` and `empty`, appearing in the observable component $(chan1 : prq)$. If the value of $(chan1 : prq)$ is `<false,p(0)> <true,p(1)> empty`, then what is displayed as the value of $(chan1 : prq)$ is `empty <true,p(1)> <false,p(0)>` because of REV.

- 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.

4.2.2 Drawing state machine pictures

It would be possible to implement the tool from scratch, but take a lot of effort as well as much time to do so. We would like to make the tool available in as many platforms and/or environments as possible. We would like to make it extensible as well as maintainable as much as possible. Therefore, it would not be preferable to implement it from scratch if there exist some technologies available to achieve our goal. 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 graphic images. It is useful to use SVG for drawing pictures of state machines. Since SVG is supported by almost all major web browsers, it makes it possible to make the tool available in as many platforms and/or environments as possible. Several tools with which SVG animations can be made have been developed. One of them is DRAW-SVG [14], which we have used in this research. DRAW-SVG is designed and developed by Joseph LIARD. It is a free online drawing application for designers and developers, making it possible to create fully standard compliant SVG. By using API based on

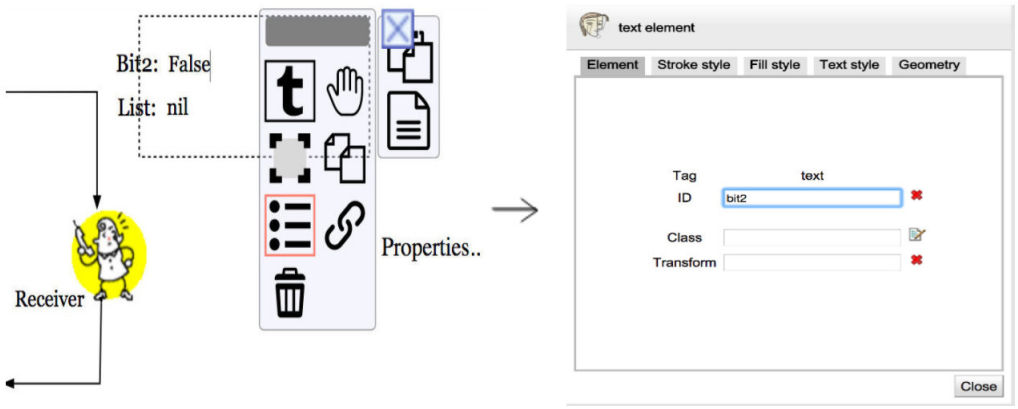

Figure 4.1: Setting id for the svgText of *bit*2

Mozilla jsSchannel, we use DRAW-SVG as an integrated drawing tool within our tool to support users draw SVG pictures for any state machines. Our tool is available on the website https://tamntt.bitbucket.io/Research/GraphicalAnimation/. The display of DRAW-SVG is supported by all currently available browsers except for Internet Explorer. However, it is optimized for Chrome and FireFox.

Human users can use DRAW-SVG to draw, save, edit, and open any SVG pictures of any state machines easily and visually. After drawing the picture of a state machine, the user needs to edit properties for texts on the picture so that the observable components of the state machine can appear on the picture when the state machine is animated. As clicking a text on the picture and choosing the icon of properties, a pop-up will be displayed for editing properties. In this pop-up, the *name* as an ID for the text of an observable component (*name* : *value*) is set for the text so that the *value* can be displayed at the place where the text is located. The ID will be used for mapping it to the values whose name is *name* appearing in an input data when we run the graphical animation tool. For example, Fig. 4.1 shows *bit*2 is set as the ID of the observable component (*bit*2 : *b*2) so that the Boolean *b*2 is displayed at the designated place on a state machine picture. Fig. 4.2 shows a picture of M_{ABP} drawn with the tool.

We have three options for setting properties to display graphically states of a input files. It depends on the purpose and expectation of the user for animation. Three options are as following:

- Option 1: We just set the property ID of a SVG text is *name* of *name* : *value* pair. By this way, values of this SVG text will be displayed and updated on it. For example, we can use this way to set property ID for SVG texts of ABP. Fig. 4.3 shows a state transition from state 32 to state 33 in a finite computation of M_{ABP}. Values of names in every state will be displayed on SVG texts of which we have set properties ID is same with *name*.

- Option 2: If we want to display name-value pairs at different locations. We have an example input file of a state machine M_{A} as follows:

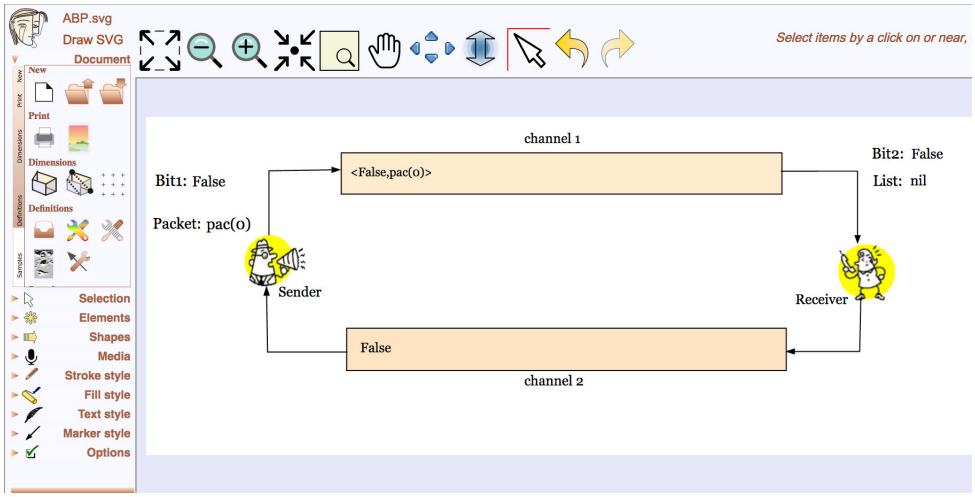

Figure 4.2: A picture of M_{ABP}

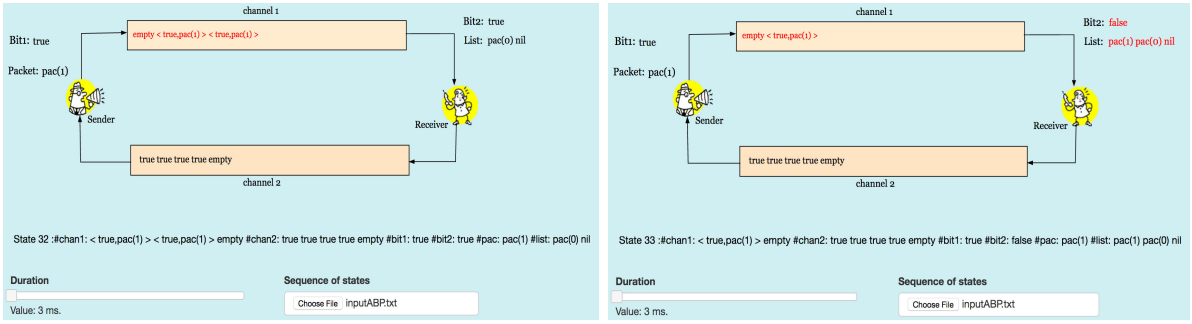

Figure 4.3: A step running of an animation

```
###keys
p1 p2
###textDisplay
###states
((p1: rs) p2: rs) ||
((p1: cs) p2: rs) ||  (p1: cs) p2: cs
```

We can draw two SVG elements as rectangles to display for two locations (rs and cs), and draw two circles with texts for display two processes $p1$ and $p2$ for every location. A SVG picture to animate M_{A} is showed on the Fig. 4.4.

The process$p1$, and $p2$ is displayed by two SVG elements contain a circle and a text. Thus, we will set properties for the circle and the text of every process at every location. The property *class* of them will be also set is *groups*. And the property ID of the circle, and the text of every process will be set as structure KEY_VALUE, where KEY is the name, and $VALUE$ is the value of a name-value pair. For process $p1$ at location rs, we will set the property ID is $p1 : rs$ for

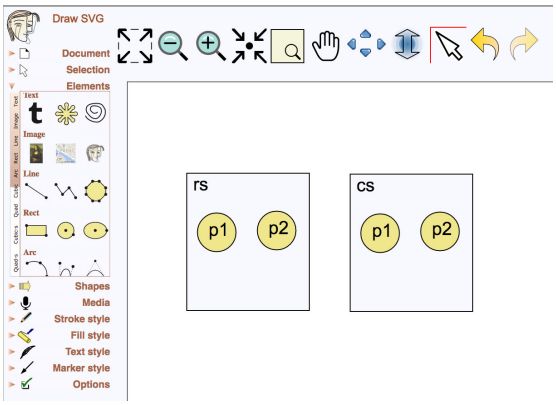

Figure 4.4: A picture of M_{A}

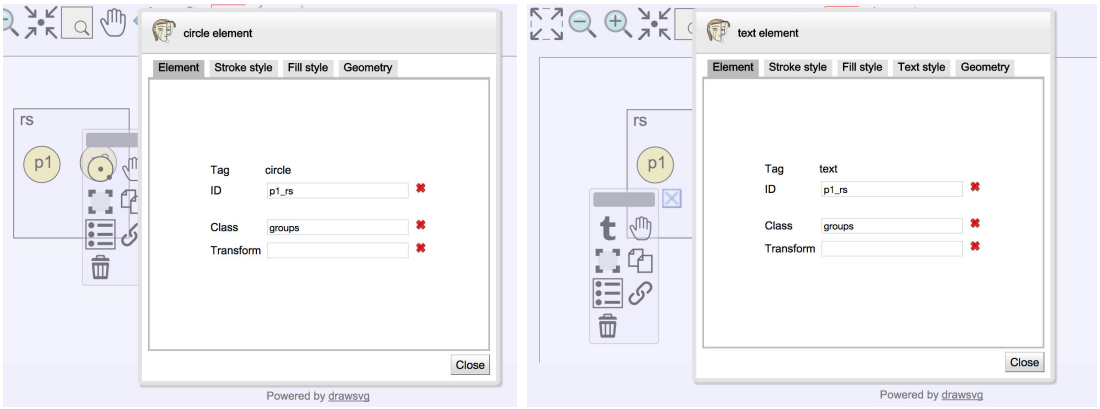

Figure 4.5: Setting properties for process $p1$ at location rs of M_{A}

.

both the circle, and text element which visualize process $p1$. Fig. 4.5 shows how to set properties for process $p1$ at location rs of M_{A}.

By this way, we can see that locations of processes are changed and displayed graphically when the tool animate states. Fig. 4.6 show the tool displays three states of M_{A}.

- Option 3: We just set property *class* as the structure $groupsVALUE1VALUE2 \ldots VALUEn$ which is the $VALUE1, VALUE2, \ldots, VALUEn$ are values of name-value pairs which we want to display. For example, we will draw a SVG picture for a state machine M_{B} which has an input file as following:

```
###keys
r1 r2 r3
###textDisplay
###states
((r1 : < 1,nil >) (r2 : < 2,nil >) r3 : < 5,nil >) ||
```

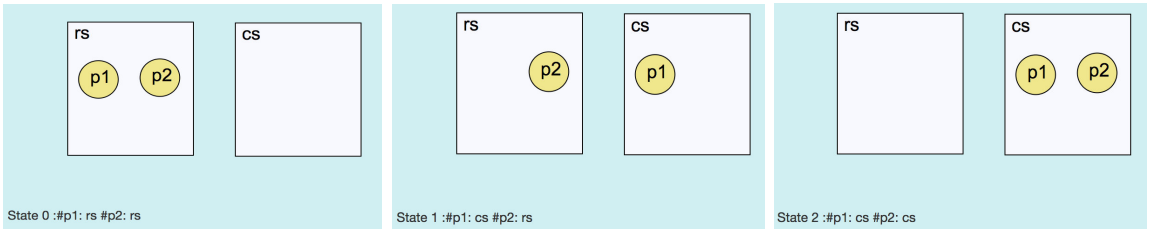

Figure 4.6: The tool run the input file of M_{A}

.

```
((r1 : < 1,L >) (r2 : < 2,nil >) r3 : < 5,nil >) ||
(r1 : < 0,nil >) (r2 : < 2,nil >) r3 : < 5,nil >
```

The drawn SVG picture of M_{B} is shown on Fig. **??**.

And Fig. 4.8 shows the sequence of states which has three states of M_{B}. Every *name* : *value* pair in every state has a value as `< L, OPT>`, where L is a location such as 0, 1, . . . , 9, and OPT is a option of a location such as nil, L, or R. If the option is nil, the location will be display as a circle. If the option is L, the location will be displayed as a circle and a left arrow from the circle. And a circle and a right arrow from the circle will be display for the case that option is R. To do that, we need to draw all circles which represent for all locations such as 0, 1, . . . , 9. A circle of a location will be displayed if it has the option is nil, L, or R. Thus, we will set property *class* of a circle of the location 0 as `groups < 0,nil > < 0,L > < 0,R >`. We also set property *class* for path elements which are arrows is *groups* and corresponding `< L, OPT>`. Fig. 4.9 shows some screenshots of setting the property *class* for the location 0 and the left arrow for option L of location 0.

4.2.3 Running tool

After getting a drawn picture of a state machine and importing a prepared input file, the tool can run to play a graphical animation of the state machine. The tool allows human users to adjust the duration of the speed of animation. The unit of duration is millisecond. The smaller the duration is, the faster the animation is played. Animations can be played step by step in addition to that they can be played automatically from the beginning to the end. When an animation is played step by step, we can observe each state transition graphically. For example, Fig. 4.3 shows a state transition (done by rec2) from state 32 to state 33 in a finite computation of M_{ABP}.

4.2.4 The algorithm of graphical animation

The algorithm used in the tool is as follows:

```
Function: animation(svg, states, keys, textDisplay, duration)
```

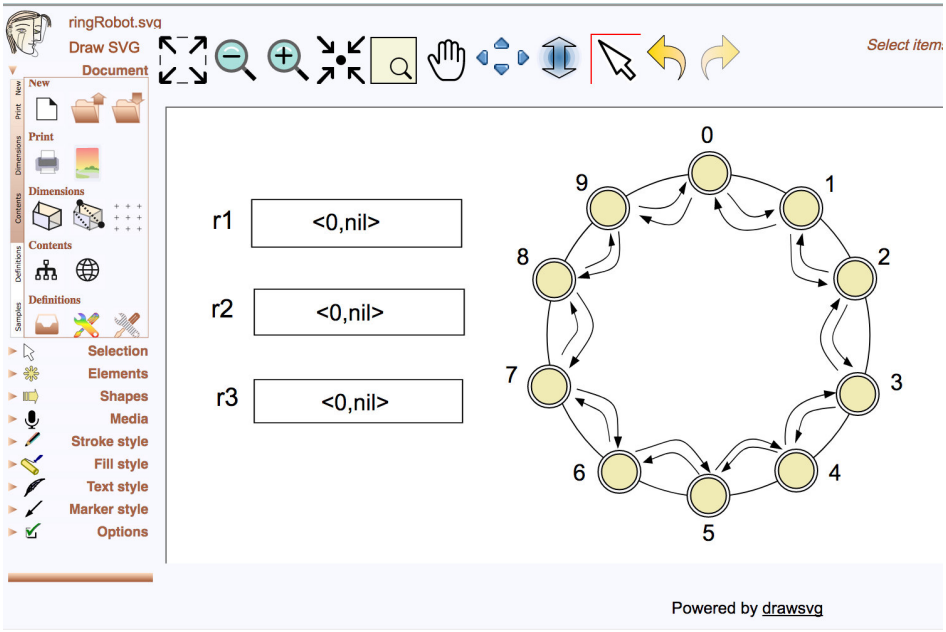

Figure 4.7: The SVG picture of M_{B}

.

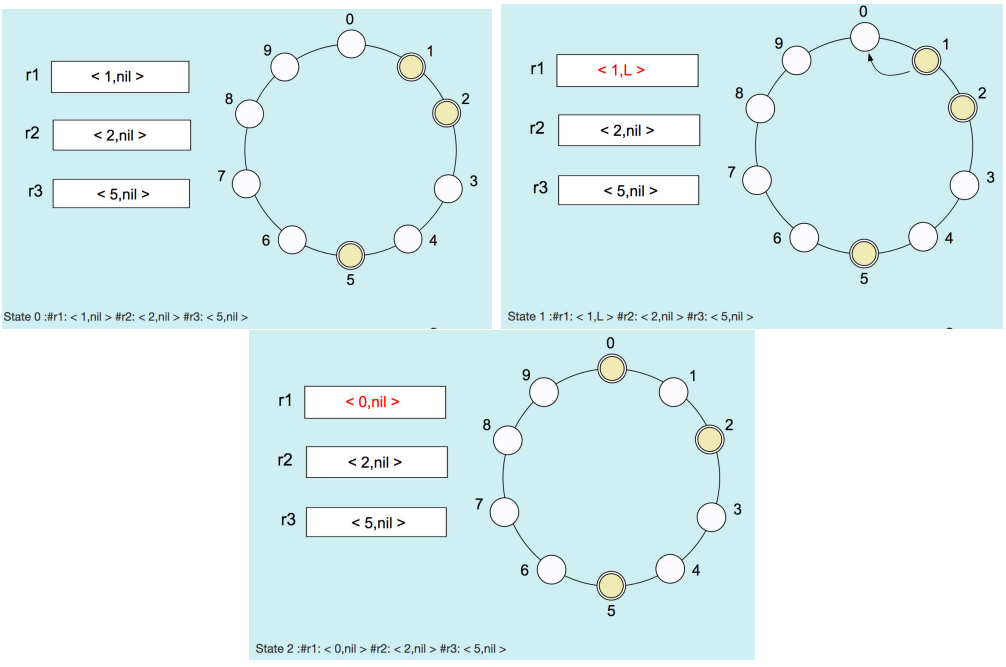

Figure 4.8: The tool run the input file of M_{A}

.

```
setHiddenElementsByClass('groups');
```

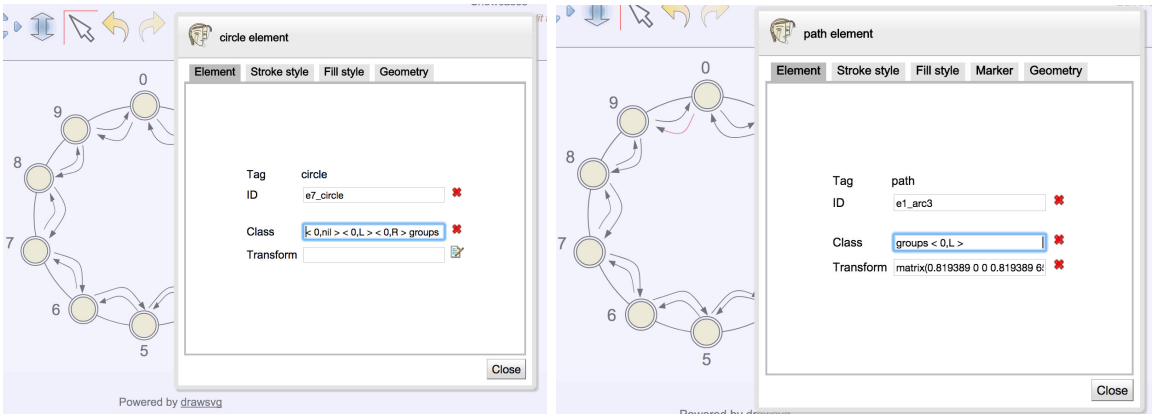

Figure 4.9: Setting property *class* for a circle element and a path element for the location 0 of M_{B}

.

```
for(i = 0, i < size(seqStates), i+1)
  state = states[i];
  preState = if i > 0 then seqState[i-1] else state;
  for(j = 0, j < size(keys), j+1)
    key = keys[j];  value2 = state[key]; value1 = preState[key]
    svgText = svg.selectById(key);  attr = '';
    if(value1 != value2) attr = 'RED_COLOR';
    else  attr = 'BLACK_COLOR';
    setTransition(svgText, attr, key, value2, duration, textDisplay[key]);
    showElementsById(key + '_' + value2);
    showElementsByClass(value2);
```

The algorithm has been implemented in JavaScript. The parameters `keys`, `textDisplay` and `states` are set the three segments in an input file, respectively. The parameter `duration` is a value of animation duration that has been set by a human user. The parameter `svg` is an object of the SVG picture that has been drawn and got.

Firstly, the function `setHiddenElementsByClass` is used to hide elements have properties *class* is *groups*.

When switching the picture of the previous state s with the picture of the successor state s', the values `value1` and `value2` of each observable component in s and s' are compared. The SVG element `svgText` that will be displayed as the value of the observable component in s' can be obtained by `svg.selectById(key)` where `key` is the name of the observable component. If `value1` and `value2` are different, red is used as the color attribute for `svgText`. Otherwise, black is used. Then, function `setTransition` is used to display `svgText` as the value of the observable component in s'. This function is used for display SVG elements which is set properties as the Option 1. To show elements have properties ID as KEY_VALUE which is the Option 2 of setting properties, we use

the function `showElementsById`. And lastly, `showElementsByClass` is used for showing elements which is set properties as the Option 3.

4.2.5 Filtering states

Observing graphical animations of a state machine may allow human users to recognize some relations among values of some observable components, such as the equivalence of *bit*1 and *bit*2 of the ABP. It would be useful to select the states among the ones in a given input file such that some condition is fulfilled and display their graphical representations. The tool allows human users to define such a condition. The format of a condition is as follows:

```
(state['key1'] op1 state['key2']) op2 (state['key3'] op4 'value') ...
```

where `key1`, `key2`, and `key3` are names of observable components in states and keys appearing in the key segment of an input file, `op1`, `op2`, and `op3` are JavaScript comparison and logical operators, and `value` is a value. An example (called Cond1) of the conditions is as follows:

```
(state['bit1'] == state['bit2'] && state['chan1'] != 'empty'
                                        && state['chan2'] != 'empty')
```

This condition can select the states such that *bit*1 equals *bit*2, *chan*1 is not empty, and *chan*2 is not empty. Let Cond2 be the condition obtained by replacing `==` with `!=` in Cond1.

In addition to the condition that has been just described, it is possible to write constraints on the value of each observable component if the value is a collection, such as a list and a queue. The format of a constraint is as follows:

```
key::::regex1++++regex2++++...++++regexn::::cond::::opt
```

where `key` is the name of an observable component, `regex(1)`, `regex(2)`, ..., `regex(n)` are regular expressions used to detect elements in the value, `cond` is a condition to be satisfied by the elements, and `opt` is either `NONE` or `REPEAT`. Let the value of the observable component be `true true true false false false empty`. If `opt` is `NONE`, the value as it is, namely `true true true false false false empty` is displayed. If `opt` is `REPEAT`, its abbreviation `true` ... `true false` ... `false` is displayed. Even though two values are different but their abbreviations are the same, the two values are treated as equals if `opt` is `REPEAT`. Eight examples of the constraints are as follows:

```
chan1::::<_,_>::::topElement(_) == bottomElement(_)::::NONE
chan1::::<_,_>::::topElement(_) == bottomElement(_)::::REPEAT
chan1::::<_,_>::::topElement(_) != bottomElement(_)::::NONE
chan1::::<_,_>::::topElement(_) != bottomElement(_)::::REPEAT
chan2::::_ _::::topElement(_) == bottomElement(_)::::NONE
```

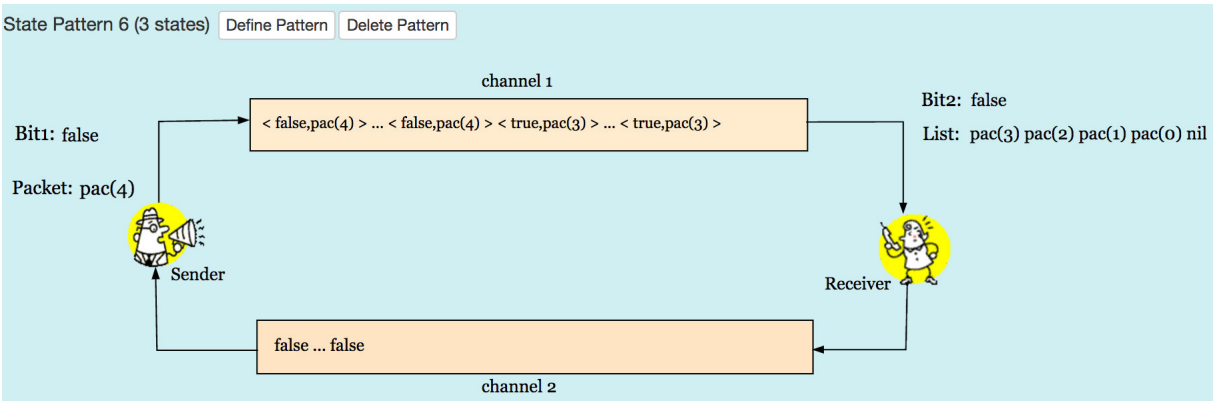

Figure 4.10: A state that satisfies Cond1, Const4 and Const6

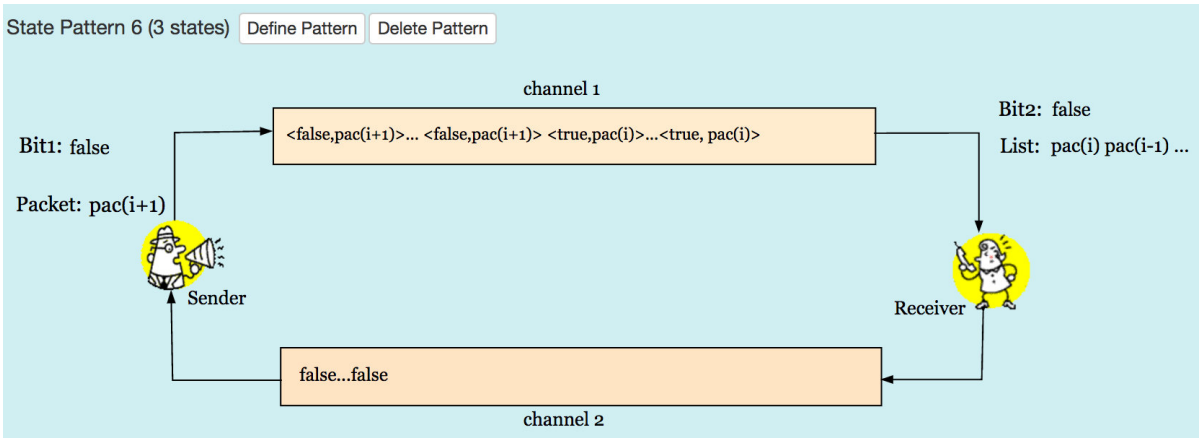

Figure 4.11: A state pattern

```
chan2::::_ _::::topElement(_) == bottomElement(_)::::REPEAT
chan2::::_ _::::topElement(_) != bottomElement(_)::::NONE
chan2::::_ _::::topElement(_) != bottomElement(_)::::REPEAT
```

where `topElement` and `bottomElement` refer to the top and bottom of the value (the queue), respectively.

Given an input file in which the keys and textDisplay segments are the same as the input file shown earlier and the states segment is a finite computation (called FC150) that consists of 150 states, when Cond1, Const4, and Const6 are used and we ask the tool to find state patterns, the tool finds 18 occurrences of states that satisfy Cond1, Const4, and Const6. Since some states occur more than once in the finite computation, the tool also finds seven different states in it. One of them is shown in Fig. 4.10.

4.2.6 Describing and displaying state patterns

For each of the states selected among the ones in a given input file such that some conditions and/or constraints are fulfilled, human users may recognize a state pattern. The tool allows human users to describe a state pattern and display it graphically. For example, from a state shown in Fig. 4.10, one may recognize the state pattern written as follows:

```
(chan1: < true,pac(i) > ... < true,pac(i) > < false,pac(i+1) > ...
< false,pac(i+1) > chan2: false ... false  bit1: false bit2: false
pac: pac(i+1) list: pac(i) pac(i-1) ...)
```

The content of *chan*1 should be displayed in the reverse order. The tool allows us to specify it as follows:

```
chan1::::REV::::<_,_>++++\.\.\.
```

Then the tool displays the state pattern shown in Fig. 4.11 that is essentially equivalent to SP6 shown in Fig. 3.1.

Chapter 5

Generation of Long Computations

It would be necessary to play a very long animation so that some non-trivial characteristics of the reachable states could be observed. Since Maude provides metaprogamming functionalities, we can write a metaprogram to generate a long computation of a state machine in Maude for displaying a long animation. A metaprogram is a program that takes programs as inputs and performs some useful computations. It is necessary to deal with a Maude specification (or program) of a state machine M to generate a long computation of M. Therefore, we have written a metaprogram that takes a Maude specification of M as one input to generate a long computation of M. The algorithm to generate a long computation of M is as follows:

```
genSeq(Mod,S,B,R)
  seq := S;  len := 1;
  while len < B
    succs := findAllSuccs(Mod,S);
    if succs = empty then break;
    s' := selectNextTerm(succs,R rem length(succs));
    seq.add(s');  len = len + 1;  R = random(R quo 100000);
 return seq;
```

in which `Mod` is the Maude specification of M, `S` is the first state of the computation, `B` is a bound that is the length of the computation being generated, and `R` is a seed of random numbers. As `R` indicates, the successor state of a state will be randomly chosen so that various different computations can be generated.

The function `findAllSuccs` takes `Mod`, `S` representing a state and returns a collection of successor states of `S` obtained by applying each of the rewrite rules to `S` if possible. `S` may be a deadlock state, namely that it may not have any successor states. If that is the case, the empty collection is returned. The function `selectNextTerm` will get a collection of successor states and a number as an index to return the next state in this collection at the index position. The function `random` generates a pseudo-random number based on the given seed. Based on the pseudo-random number generated, the function `selectNextTerm` will return the next state. Since modules, terms, etc. are expressed

as Maude terms, Maude makes it possible to write metaprograms in Maude as ordinary programs (or specifications) in Maude.

What is returned by the function `genSeq` is a finite computation but the computation is represented as a meta-term. Hence, such a meta-represented term should be converted to another representation that can be used for the tool. Then, we have defined the function `downTermList` as follows:

```
op nil : -> ListSys [ctor] .
op _||_ : Sys ListSys -> ListSys [ctor] .
op downTermList : TermList -> ListSys .
eq downTermList(empty) = nil .
eq downTermList(TE) = downTerm(TE, nil)  .
eq downTermList((TE,TList)) = downTerm(TE, nil) || downTermList(TList) .
```

where `TE` and `TList` are Maude variables of sorts `Term` and `TermList`. `ListSyst` is the sort of finite computations that can be used for the tool. The function `downTerm` takes a meta-represented term and convert it into an object-level representation of the term. For example, we can generate the finite computation FC150 of M_{ABP} whose length is 150 by reducing the following term:

```
downTermList(genSeq(upModule('ABP,false),upTerm(init),5,150)) .
```

where the function `upModule` takes a module name as a quoted term, such as `'ABP` where `ABP` is the name of a module in which ABP is specified, and converts it into a meta-represented term of the module and the function `upTerm` takes a term and converts it into a meta-represented term of the term. The way to generate finite computations can generate finite computations up to about 100000 for M_{ABP}.

Chapter 6

Experiment

We have used two finite computations FC150, and FC500 of M_{ABP} to conduct the experiment for state patterns recognition. Observing animations from them have made us find out some of the six state patterns shown in Fig. 3.1. Even if we may not find out any interesting state patters, we can ask the tool to look for the states in the animation that satisfy conditions and/or constraints.

We have used Cond$i$ for $i = 1; 2$ and Constj for $i = 1, 2, \ldots 5$. as defined conditions and constraints, respectively. The condition Const1, Const2, ..., Const5 are placed in the Regex part of the tool. And the condition Cond1, and Cond2 are placed in the Condition part of the tool. Some definitions of these conditions, and constraints are shown in the Fig. 6.1.

Firstly,we used FC150 for running tool and selecting states satisfied these conditions. The tool found 55 occurrences of the states that satisfied Cond1 among which there were 40 different ones. When we used C12 as well, the tool found 37 occurrences of the states that satisfied Cond1, and Const2 among which there were 11 different states. Taking a close look at those 11 different state patterns made us recognize SP1 and SP5 shown in Fig. 3.1. After using the finite computation FC150 of M_{ABP}, we continued conducting the experiment with FC500 by using these conditions for selecting. The Fig. 6.2 shows the experimental results of FC150, and FC500 are in which OS, DSP, SP and SPj(,k) stand for the number of occurrences of states, the number of different states or state patterns, state patterns, and SPj (and SPk), respectively.

By presenting the result of FC150, and FC500 on charts shown in Fig. 3.1, and Fig. 3.1 respectively, we can see that selecting states satisfied conditions is able to support users recognize useful state patterns and reduce amounts of states to detect them. The tool reveals that there is no state that satisfies some condition and constraints. Although the tool does not prove it, this information is crucial.

Name	Definition
Cond1	state['chan1'] != 'empty' && state['chan2'] != 'empty' && state['bit1'] == state['bit2']
Cond2	state['chan1'] != 'empty' && state['chan2'] != 'empty' && state['bit1'] != state['bit2']
Const1	chan1::::<_,_>::::topElement(_) == bottomElement(_)::::NONE chan2::::_ _::::topElement(_) == bottomElement(_)::::NONE
Const2	chan1::::<_,_>::::topElement(_) == bottomElement(_)::::REPEAT chan2::::_ _::::topElement(_) == bottomElement(_)::::REPEAT
Const3	chan1::::<_,_>::::topElement(_) != bottomElement(_)::::REPEAT chan2::::_ _::::topElement(_) == bottomElement(_)::::REPEAT
Const4	chan1::::<_,_>::::topElement(_) == bottomElement(_)::::REPEAT chan2::::_ _::::topElement(_) != bottomElement(_)::::REPEAT
Const5	chan1::::<_,_>::::topElement(_) != bottomElement(_)::::REPEAT chan2::::_ _::::topElement(_) != bottomElement(_)::::REPEAT

Figure 6.1: The definition of conditions

Conditions, Constraints		FC150		FC500		
Name	Included Conditions, Constrains	OS	DSP	OS	DSP	SP
C1	Cond1	55	40	72	57	1,5,6
C11	Cond1 and Const1	37	24	49	36	1,5
C12	Cond1 and Const2	37	11	49	15	1,5
C13	Cond1 and Const3	18	7	23	10	6
C14	Cond1 and Const4	0	0	0	0	
C15	Cond1 and Const5	0	0	0	0	
C2	Cond2	39	32	50	38	2,3,4
C21	Cond2 and Const1	18	14	22	17	2,4
C22	Cond2 and Const2	18	8	22	11	2,4
C23	Cond2 and Const3	0	0	0	0	
C24	Cond2 and Const4	21	10	28	10	3
C25	Cond2 and Const5	0	0	0	0	

Figure 6.2: The result of experiment.

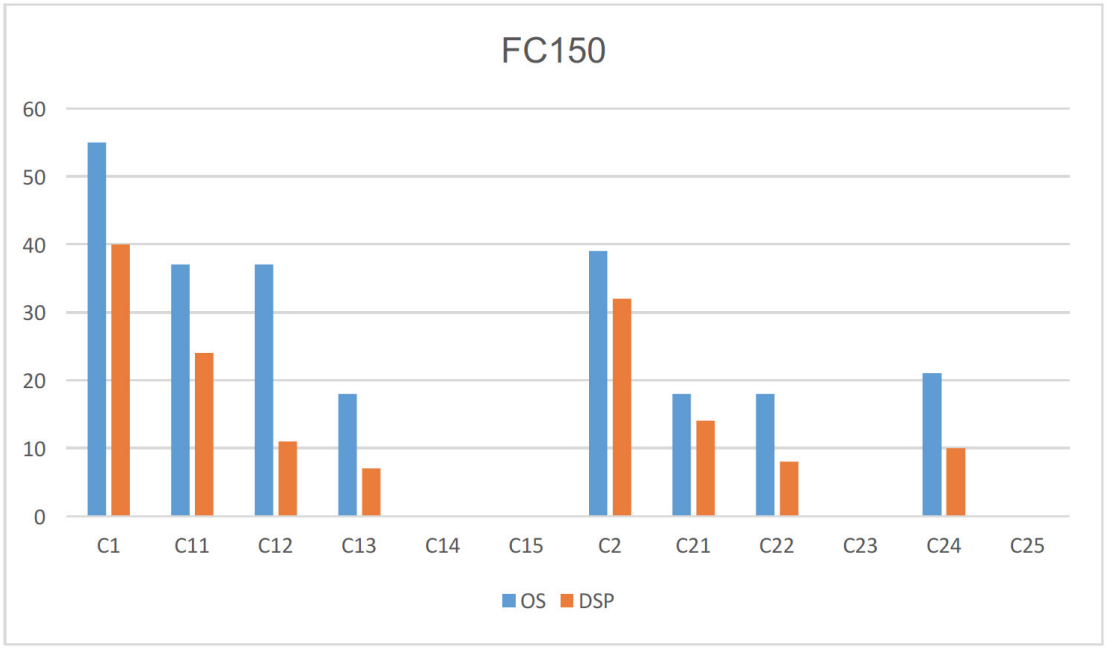

Figure 6.3: The result chart of FC150.

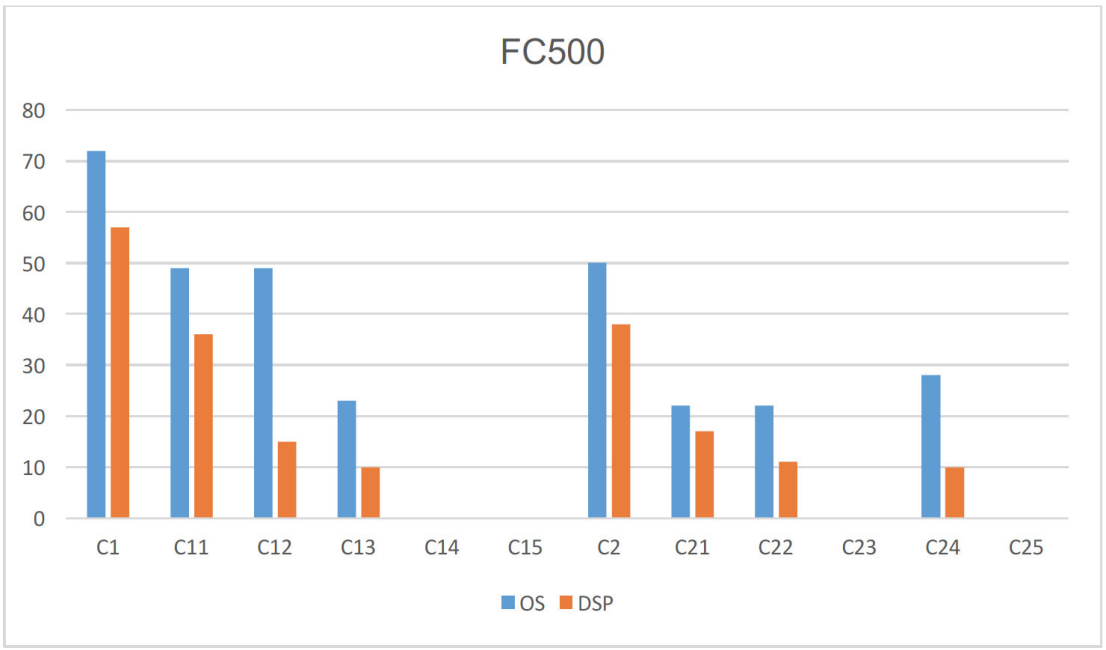

Figure 6.4: The result chart of FC500.

Chapter 7

Applications

7.1 Comprehending Counterexamples Generated by the Maude LTL Model Checker

7.1.1 Introduction

Although Maude LTL model checker has many good points, counterexamples it generates may not be necessarily the shortest ones. The shorter a counterexample, the easier it is to comprehend the counterexample. Maude is equipped with the search command. The command exhaustively traverses the reachable states from a given state s_0 to find some states s that match some pattern and/or satisfy some condition in a breadth-first manner. A path from s_0 to s is also generated. Since the search is performed in a breadth-first manner, the path is the shortest one from s_0 to s. Maude also allows us to write meta-programs as ordinary programs because Maude provides many useful metalevel functions including descent and ascent functions, one of which is the counterpart of the search command. A descent function converts a metalevel representation of a term, a module, etc. into an ordinary representation of the term, the module, etc. and an ascent function does the reverse conversion. We have implemented a meta-program that takes a state machine specification module and a counterexample generated by Maude LTL model checker and generates a shorter counterexample, in which the metalevel search function is used.

The state machine graphical animation tool basically takes a finite computation $s_0; s_1; \ldots; s_k$ of M. As some model checkers, such as PAT, make it possible to represent a counterexample graphically and run it step by step, we have realized our tool could help human users comprehend a counterexample. Therefore, we have extended the tool so that the tool can graphically animate a counterexample that consists of $s_0; \ldots; s_m$ and $(s_{m+1}; \ldots; s_n)*$. Since the loop is repeatedly played, the repetition could help human users realize what happen in the loop. Unlike other model checkers, the tool allows human users design pictures or flames of animations, adjust the speed of automatic plays of animations, and select states from a counterexample such that some conditions and/or constraints are satisfied, which could be likely to help human users comprehend a counterexample better.

7.1.2 A flawed mutual exclusion protocol (FQlock)

Let us consider a flawed mutual exclusion protocol as an example. The protocol is called FQlock whose pseudo code executed by a process p_i is as follows:

Loop
"Remainder Section (RS)"
rs : $queue := \mathrm{enq}(queue, p_i)$;
ws : **repeat until** $\mathrm{top}(queue) = p_i$;
"Critical Section (CS)"
cs : $tmp_i := \mathrm{deq}(queue)$;
cs : $queue := tmp_i$;

where *queue* is a queue of process IDs that is shared by all processes. Queuing a process ID p_i into *queue* is done atomically, while dequeuing *queue* is not.

Inductively defined data structures and associative and/or commutative binary operators can also be used in a specification processed by Maude LTL model checker. State transitions are specified as rewrite rules that could be equipped with conditions. For example, the state transitions of FQlock are specified as follows:

```
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) .
```

where `eq`, `wt`, etc. are the labels of the rewrite rules, `I` is a Maude variable of the sort `Pid`, and `Q` and `R` are Maude variables of the sort `Queue`. `(pc[I]: rs)` and `(queue: Q)` are observable components, where `pc[I]:` and `queue:` are names and `rs` and `Q` are values. The name `pc[I]:` has the parameter `I`. Queues of process IDs are inductively constructed with the two constructors:

```
op empty : -> Queue [ctor] .
op __ : Pid Queue -> Queue [ctor] .
```

where `empty` denotes the empty queue and the juxtaposition operator `__` is also used to construct non-empty queues. Given process IDs `p1`, `p2` and `p3`, the term `p1 p2 p3 empty` denotes the queue of process IDs that consists of the three process IDs in this order.

Let us suppose there are two processes whose IDs are `p1` and `p2` and then the initial state of FQlock is denoted as the term `(pc[p1]: rs) (pc[p2]: rs) (tmp[p1]: empty) (tmp[p2]: empty) (queue: empty)`. The term will be referred as `init`. By substituting `I` and `Q` with `p2` and `empty`, the left-hand side of the rewrite rule `eq` is the same as the term. Therefore, the rewrite rule `eq` can be applied to the term. If it is with the substitution, the term rewrites to `(pc[p1]: rs) (pc[p2]: ws) (tmp[p1]: empty) (tmp[p2]: empty) (queue: p2 empty)`. This is how state transitions are done by rewriting.

7.1.3 Maude LTL Model Checker

It is an explicit-state on-the-fly LTL model checker. The model checking algorithm used is the same as the one used in Spin. FQlock is used to describe how to use it. Users are supposed to specify atomic propositions. Let us suppose we model check FQlock enjoys the lockout freedom property when there are two processes. The lockout freedom property says whenever each process wants to enter the critical section, it will eventually be there. Therefore, we specify two kinds of atomic propositions `wait(P)` and `crit(P)`, where `P` is a process ID. If there are two processes whose IDs are `p1` and `p2`, there are totally four atomic propositions `wait(p1)`, `wait(p2)`, `crit(p1)` and `crit(p2)`. Users are also supposed to specify a labeling function. For our purpose, we declare the three equations:

```
eq (pc[P] : ws) S |= wait(P) = true .
eq (pc[P] : cs) S |= crit(P) = true .
eq S |= PROP = false [owise] .
```

where `P` is a Maude variable of the sort `Pid`, `S` is a Maude variable of the sort `Sys` that is for states, and `PROP` is a Maude variable of the sort `Prop` that is for atomic propositions. The three equations say a state s satisfies `wait(P)` if and only if `(pc[P]: ws)` $\subseteq s$ and s satisfies `crit(P)` if and only if `(pc[P]: cs)` $\subseteq s$.

Then, users are supposed to specify LTL formulas to check. The lockout freedom property is expressed as `wait(P)` $\leadsto$ `crit(P)` for all `P` = {p1,p2}. In Maude, the formula is specified as `eq lofree = (wait(p1) |-> crit(p1)) /\ (wait(p2) |-> crit(p2)) .`, where the operator `_|->_` denotes the leadsto operator $\leadsto$.

Model checking that the Kripke structure formalizing FQlock satisfies the lockout freedom property `lofree` is conducted by reducing the term `modelCheck(init,lofree)`. Since FQlock does not enjoy the lockout freedom property, Maude LTL model checker generates a counterexample that 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]: p1empty,'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: (p2 empty) (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})
```

The first element that is a finite computation consists of 17 states and the second element that is a loop of states consists of four states. This is a counterexample but not the shortest one.

7.1.4 Shorter Counterexamples

For a Kripke structure K, an LTL formula φ and a counterexample $\pi \triangleq s_0; \ldots; s_k; (s_{k+1}; \ldots; s_m)^\infty$ such that $K, \pi \not\models \varphi$, π is the shortest counterexample for K and φ if and only if there is no counterexample $\pi' \triangleq s_0; \ldots; s_l; (s_{l+1}; \ldots; s_n)^\infty$ such that $K, \pi' \not\models \varphi$, and $(l \leq k \wedge m - k \leq n - l) \wedge \neg(l = k \wedge m - k = n - l)$. π' is shorter than π (denoted as $\pi' < \pi$) if and only if $(l \leq k \wedge m - k \leq n - l) \wedge \neg(l = k \wedge m - k = n - l)$. π' is shorter than or equal to π (denoted as $\pi' \leq \pi$) if and only if $(l \leq k \wedge m - k \leq n - l)$. Let $\mathrm{fc}(\pi)$ be $s_0; \ldots; s_k$ and $\mathrm{loop}(\pi)$ be $s_{k+1}; \ldots; s_m$.

For a Kripke structure K and a finite path $s_0; \ldots; s_k$ of K, the shortest path from s_0 to s_k can be found by exhaustively traversing the reachable states of K from s_0 in a breadth-first manner. The proof is conducted by contradiction. If there were a shorter path from s_0 to s_k than the one found by the search, the search would find s_k at a shallower position because of a breadth-first manner.

Maude is equipped with the search command that exhaustively traverses the reachable states from a given state to find states that match some pattern and satisfy some condition in a breadth-first manner. Since Maude is also equipped with a metalevel function that is the counterpart of the search command, we can write a meta-program that takes a module that corresponds to a Kripke structure and a counterexample π and returns a counterexample π' such that $\pi' \leq \pi$.

Given a module `M` in which a system or Kripke structure is specified, an initial state `ST` of the system and an LTL formula `LF`, if Maude model checker generates π such that $\pi(0)$ is `ST` and $\mathtt{M}, \pi \not\models \mathtt{LF}$, π' is generated with the following functions `shortert`, and `loopSeqStates` such that $\pi' \leq \pi$, $\mathrm{fc}(\pi')$ is `shorter(M,ST,LF)`, and $\mathrm{loop}(\pi')$ is `loopSeqStates(M,ST,LF)`:

```
eq shortest(M,ST,LF) = subShorter(M,modelCheck(ST,LF)) .
eq loopSeqStates(M,ST,LF) = subloopSeqStates(M,modelCheck(ST,LF)) .
```

The function `subShorter` is defined as follows:

```
eq subShorter(M,true) = nil .
eq subShorter(M,counterexample(TL1,TL2)) = searchSequenceStates(M,
    getTermFromTransList(TL1,0),
    getTermFromTransList(TL1,lengthTransList(TL1) - 1),nil,'*,unbounded,0) .
```

If no counterexample is found, `nil` (the empty list) is returned. Otherwise, let π be a counterexample found. `TL1` is $\mathrm{fc}(\pi)$ and `TL2` is $\mathrm{loop}(\pi)$. The first and last states of `TL1` are used as the second and third parameters of the function `searchSequenceStates`. The function is defined as follows:

```
eq searchSequenceStates(M,I,P,Cond,T,B,N)
       = trace2TermList(M,metaSearchPath(upModule(M,false),I,P,Cond,T,B,N)) .
```

The function `metaSearchPath` is a metalevel function that is the counterpart of the Maude search command. More precisely, it is the counterpart of the combination of the search command and the show path command. It takes a module meta-represented, a state meta-represented from which the search is carried out, a pattern meta-represented such that the search tries to find states that match the pattern, a condition such that the search tries to find states that satisfy the condition, a search type, a bound (or a depth) and a natural number. The search type specifies how many transitions it takes to reach states to be found, for example, zero or more transitions, one or more transitions and exactly one transition. `'*` specifies zero or more transitions. The natural number n specifies the nth path (or solution) to a state to be found is returned. Note that 0 means the first solution. `nil` as the condition means true. `unbounded` as the bound asks the search to try to exhaustively traverse all reachable states from a given state until a designated state (or solution) is found or all possible states have been visited. The function `upModule` is an ascent function that takes a name of a module and returns a term representing the module (the module meta-represented).
The function `trace2TermList` converts a path meta-represented into a user-define list meta-represented. In the user-defined lists, `nil` denotes the empty list, and `_||_` is used as the constructor for non-empty lists.
Let `FQLOCK` be a module in which FQlock is specified. By reducing the term `downTermSearch(shorter('FQLOCK,init,lofree))`, the following term is returned:

```
(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
```

The list consists of nine states, while fc(π) consists of 17 states, where π is the counterexample generated by Maude LTL model checker.
The function `loopSeqStates` is defined likewise. The term is returned by reducing the term `downTermSearch(loopSeqStates('FQLOCK,init,lofree))` as following:

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

The list consists of four states as loop(π) does. The meta-program that consists of the two metalevel functions `shorter` and `loopSeqStates` could shorten a given counterexample π and generate π' such that $\pi' \leq \pi$ but does not guarantee π' is the shortest.

7.1.5 Graphical Animations of Counterexamples

It would be easier to comprehend a shorter counterexample, but a text representation of a counterexample could not be necessarily intuitively understandable. A graphical representation of a counterexample would help human users comprehend it more intuitively. Accordingly, some model checkers, such as PAT and Alloy, are equipped with functionalities that represent counterexamples graphically.

A counterexample is not necessarily a finite computation but consists of a finite computation and a loop. The tool could be used to graphically animate a counterexample by regarding the counterexample as a finite computation. We believe, however, it would be meaningful to repeatedly animate the loop part of a counterexample. Therefore, we have extended the tool so that the tool can animate a counterexample such that the loop part is repeatedly animated.

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

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

###textDisplay

###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)

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

There are four regions: `###keys`, `###textDisplay`, `###states` and `###loop`. The fourth region `###loop` has been newly added. For a counterexample π to be animated, fc(π) is written. The fourth region `###loop` is specific to counterexamples. loop(π) is written there.

Fig. 7.1 shows the picture we have designed for animations of FQlock when there are two processes whose IDs are `p1` and `p2`. This is just one possible picture for animations of FQlock and each user is allowed to design his/her own picture or flame of animations of FQlock.

Feeding the input file whose contents have been shown into the tool, the animation of the counterexample is played. The finite computation part is played and then the loop part is repeatedly played, which is shown in Fig. 7.2. Looking at the loop part being

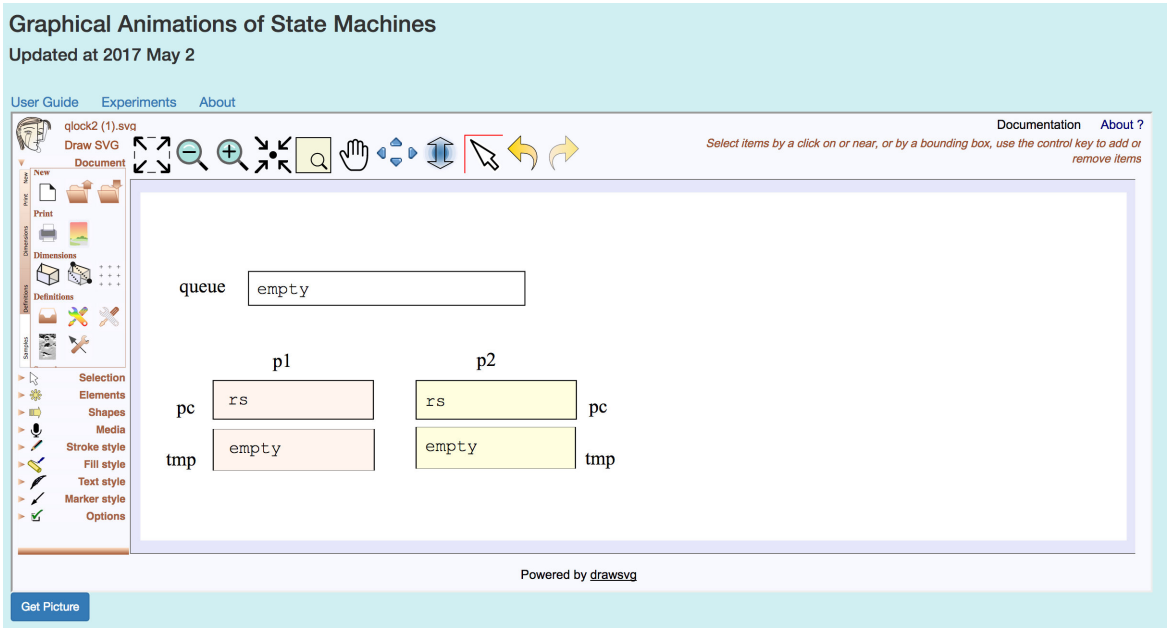

Figure 7.1: Picture of FQLock

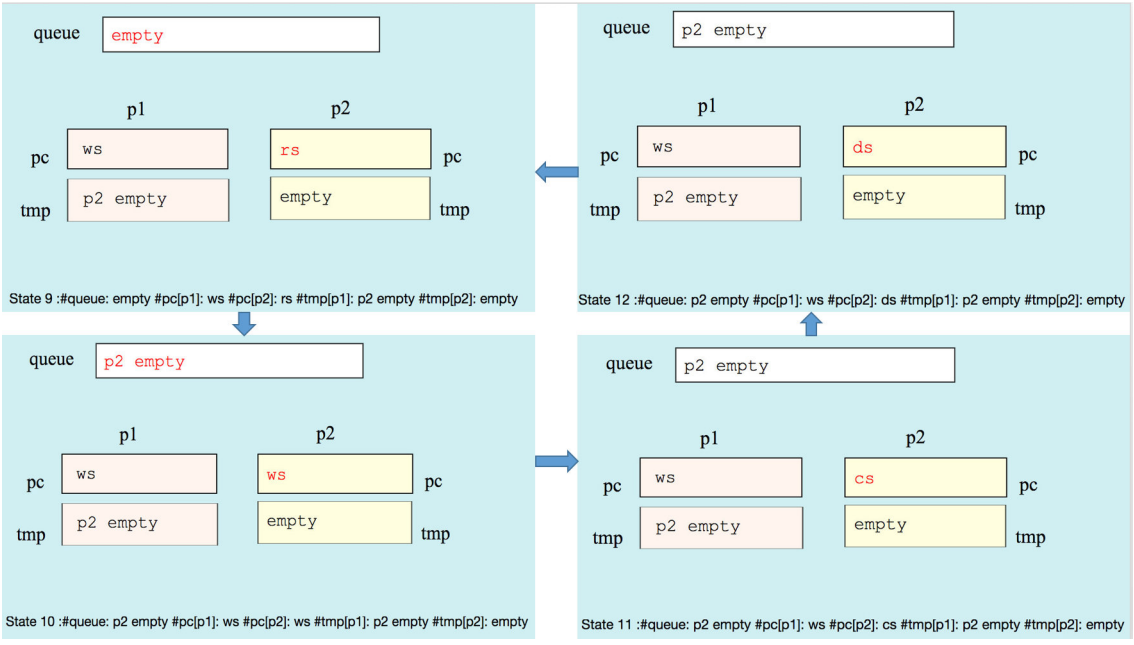

Figure 7.2: Loop part of the counterexample of `lofree`

animated, we realize the process `p2` visits `rs`, `ws`, `cs` and `ds` iteratively, while the process `p1` keeps in `ws`. Looking at the loop being animated helps us realize this counterexample is a really one of `lofree`, but does not reveal why FQlock goes into the loop.

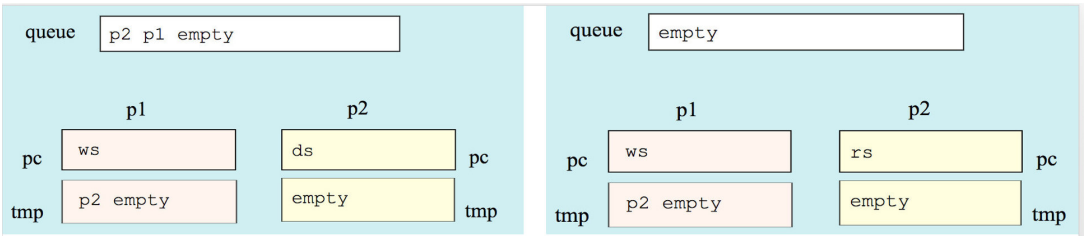

Figure 7.3: Two states in which `pc[p1]` is `ws`

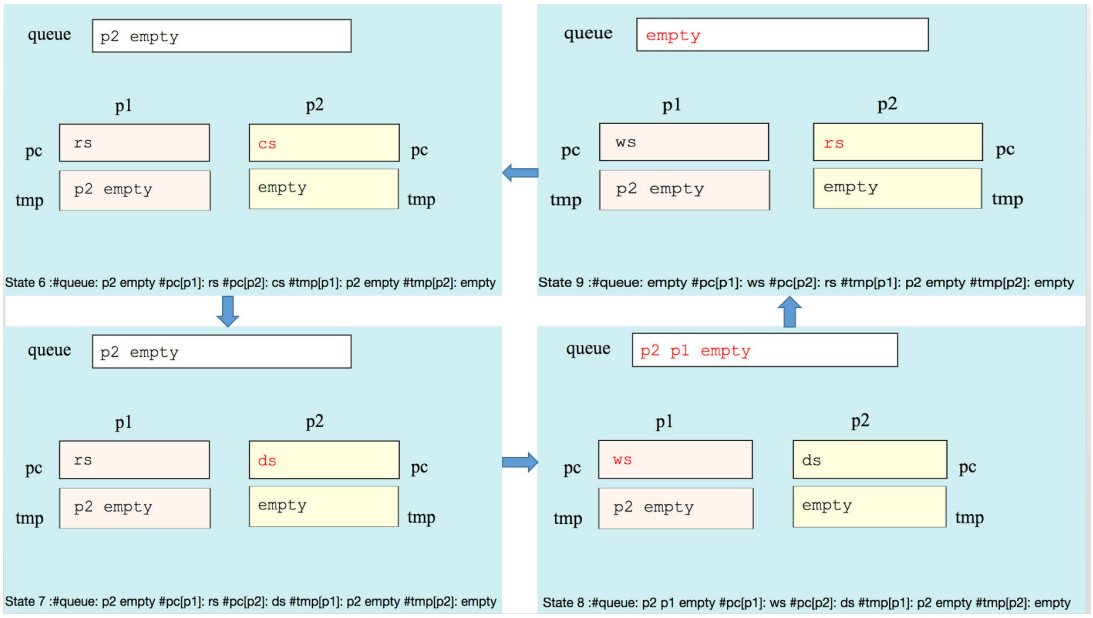

Figure 7.4: Three state transitions leading to the counterexample

The tool allows users to select some states that satisfy some conditions and/or constraints. Thus, we have asked the tool to select the states that satisfy the condition that the value of the observable component whose name is `pc[p1]` is `ws`. The tool found seven such states from the counterexample, among which we found two interesting states shown in Fig. 7.3. The left state occurs earlier than the right state. In the left state, `pc[p1]` is `ws` and `queue` is `p2 p1 empty`, in which `p1` is in the queue, while in the right state `queue` is `empty` but `pc[p1]` is still `ws`, implying that `p1` will never be in `cs`. In the left state `pc[p2]` is `ds` and `tmp[p2]` is `empty` and in the right state `pc[p2]` is `rs`. Therefore, the rewrite rule `dq2` must have been applied to the left state with the substitution that `I` is `p2`, `Q` is `p2 p1 empty` and `R` is `empty`. We took a close look at the animation at around the left state shown in Fig. 7.3 and then found three state transitions leading to the counterexample, which are shown in Fig. 7.4. We can do this quickly because the tool lets us know the states selected appear at which positions in an input, directly displays the state whose position is given and plays the animation from the state step by step backwardly

as well as forwardly. The first state transition is done by the rewrite rule `dq1`, the second one is done by `eq`, and the third one is done by `dq2`. Since dequeuing `queue` is not atomic, the second one has interrupted the dequeuing of `queue` done by the two rewrite rules `dq1` and `dq2`. This is the reason why the counterexample occurs. One possible remedy is to make dequeuing `queue` atomic.

7.2 Analysis of MCS List-based Queuing Lock

7.2.1 Introduction

The MCS list-based queuing lock (MCS protocol) is a mutual exclusion protocol whose variants have been used in Java virtual machines. It has been invented by John M. Mellor-Crummey and Michael L. Scott [17] who were awarded the 2006 Edsger W. Prize in Distributed Computing[1]. MCS protocol uses a global queue to control processes such that there exists at most one process in the critical section and any process that wants to enter the critical section will be eventually there, but the global queue is not an atomic queue. The queue is a linked structure, and neither enqueuing an element into the queue at the end nor dequeuing the queue are done atomically. MCS Protocol uses two atomic operators fetch&store and comp&swap to make the two basic operations to the queue conducted safely. Accordingly, MCS protocol is not simple and deserve to be formally and carefully analyzed. We have conducted a case study in which MCS protocol and some variants have been analyzed with Maude and the animation tool, demonstrating the usefulness of the combination of Maude and the animation tool.

7.2.2 MCS List-based Queuing Lock

A pseudo-code of MCS protocol for each process p is as follows:

rs: "Remainder Section"
l1: $next_p$:= nop;
l2: $pred_p$:= fetch&store($glock, p$);
l3: **if** $pred_p \neq$ nop {
l4: $lock_p$:= true;
l5: $next_{pred_p}$:= p;
l6: **repeat while** $lock_p$; }
cs: "Critical Section"
l7: **if** $next_p$ = nop {
l8: **if** comp&swap($glock, p$, nop)
l9: **goto** rs;
l10: **repeat while** $next_p$ = nop; }
l11: $locked_{next_p}$:= false;
l12: **goto** rs;

[1] https://www.podc.org/dijkstra/2006-dijkstra-prize/

There is one global variable *glock* shared by all processes participating in MCS protocol. Its type is process IDs (or Pid). Initially, *glock* is nop, a dummy process ID. Each process p maintains three local variables $next_p$, $lock_p$ and $pred_p$ whose types are Pid, Bool and Pid, respectively. Initially, $next_p$, $lock_p$ and $pred_p$ are nop, false and nop, respectively. $next_p$ is used to construct a global queue of processes (or process IDs). Basically, $next_p$ refers to the next element of the queue if p is in the queue. Since enqueuing an element into the queue and dequeuing the queue are not atomically done, however, $next_p$ may be nop even though p is not the bottom element of the queue. $pred_p$ refers to the previous element of the queue while p is being put into the queue. $lock_p$ is the local lock on which process p is spinning while $lock_p$ is true to wait for entering the critical section. *glock* basically refers to the bottom element if the queue is not empty. Since the two basic operations to the queue are not atomic, however, *glock* may not refer to the real bottom element while some process IDs are being put into the queue.

To safely conduct the two basic operations to the queue non-atomically, two atomic operations are used: fetch&store and comp&swap. fetch&store(x, v) does the following atomically: $tmp := x$, $x := v$, and tmp is returned, where tmp is a temporary variable. comp&swap(x, v_1, v_2) does the following atomically: if $x = v_1$, then $x := v_2$ and true is returned; otherwise, false is returned.

MCS protocol is formalized as a state machine whose states are expressed as soups of observable components. When there are three processes, a state is expressed as:

```
(glock: G) (pc[p1]: L1) (pc[p2]: L2) (pc[p3]: L3)
(next[p1]: P1) (next[p2]: P2) (next[p3]: P3)
(lock[p1]: B1) (lock[p2]: B2) (lock[p3]: B3)
(pred[p1]: Q1) (pred[p2]: Q2) (pred[p3]: Q3)
```

where `G`, `P`i and `Q`i for $i = 1, 2, 3$ are process IDs, `L`i for $i = 1, 2, 3$ are locations, such as rs, l1 and cs, and `B`i for $i = 1, 2, 3$ are Booleans. Initially, `G`, each `P`i and each `Q`i are nop, each `L`i is rs, each `B`i is false. The initial state will be referred as `init`.

The state transitions are described in terms of rewrite rules as follows:

```
rl [want] : (pc[P]: rs) => (pc[P]: l1) .
rl [stnxt] : (pc[P]: l1) (next[P]: Q) => (pc[P]: l2) (next[P]: nop) .
rl [stprd] : (glock: Q) (pc[P]: l2) (pred[P]: Q1)
=> (glock: P) (pc[P]: l3) (pred[P]: Q) .
rl [chprd] : (pc[P]: l3) (pred[P]: Q)
=> (pc[P]: (if Q == nop then cs else l4 fi)) (pred[P]: Q) .
rl [stlck] : (pc[P]: l4) (lock[P]: B) => (pc[P]: l5) (lock[P]: true) .
rl [stnpr] : (pc[P]: l5) (pred[P]: Q) (next[Q]: Q1)
=> (pc[P]: l6) (pred[P]: Q) (next[Q]: P) .
rl [chlck] : (pc[P]: l6) (lock[P]: false)
=> (pc[P]: cs) (lock[P]: false) .
rl [exit] : (pc[P]: cs) => (pc[P]: l7) .
rl [rpnxt] : (pc[P]: l7) (next[P]: Q)
=> (pc[P]: (if Q == nop then l8 else l11 fi)) (next[P]: Q) .
rl [chglk] : (glock: Q) (pc[P]: l8)  =>
(glock: (if Q == P then nop else Q fi))
(pc[P]: (if Q == P then l9 else l10 fi)) .
```

```
rl [go2rs] : (pc[P]: l9) => (pc[P]: rs) .
crl [rpnxt2] : (pc[P]: l10) (next[P]: Q)
=> (pc[P]: l11) (next[P]: Q) if Q =/= nop .
rl [stlnx] : (pc[P]: l11) (next[P]: Q) (lock[Q]: B)
=> (pc[P]: l12) (next[P]: Q) (lock[Q]: false) .
rl [gotrs] : (pc[P]: l12) => (pc[P]: rs) .
```

where `want`, `stnxt`, etc. are the labels of the rewrite rules.

7.2.3 Analyzing the mutual exclusion property

The mutual exclusion property that should be enjoyed by mutual exclusion protocols, such as MCS protocol, says that there exists at most one process in the critical section at any given moment. Therefore, the search command can be used to check if MCS protocol enjoys the property as follows:

```
search [1] in MCS-INIT : init =>* (pc[I]: cs) (pc[J]: cs) S .
```

where `MCS-INIT` is the module in which MCS protocol is specified in Maude, `I` and `J` are Maude variables of process IDs, and `S` is a Maude variable of states (or soups of observable components). If Maude finds a solution, MCS protocol does not enjoy the property. Maude did not find any solutions, implying that MCS protocol enjoys the property when there are three processes.

7.2.4 Graphical Animations of MCS Protocol

Fig. 7.5 shows the picture we have designed or drawn for MCS protocol when there are three processes. To display appearances of processes such as process $p1$ at location cs, process $p2$ at location $L1$, ..., we have to set property ID, and $class$ of SVG elements which will display graphical animation of processes to map the input file correctly.

We need to draw all SVG elements of all process at all locations, then set property $class$ is $group$ for them. By this way, the tool will detect and hide them until a current state which is displaying contains information of them. To showing SVG elements which have same name and value in a current state, we need to set property ID of them as KEY_VALUE, where KEY, $VALUE$ is a name, and a value of a name-value pair respectively. For example, we will set property ID of SVG element which animate process $p1$ at rs location is $pc[p1]\_rs$, and set property $class$ is $groups$.

The tool basically takes a sequence of states and plays it graphically. It can select and display the states that satisfy a condition from the input finite computation. We asked the tool to select and display the states such that the location of `p1` is `l10`. The tool found 16 such states in the input finite computation. Fig. 7.6 shows one of the 16 states.

7.2.5 Analyzing the lockout freedom property

We suppose that there are two processes `p1` and `p2` and let `init` denote the initial state in which the two processes participate in MCS protocol. Let us suppose we model check

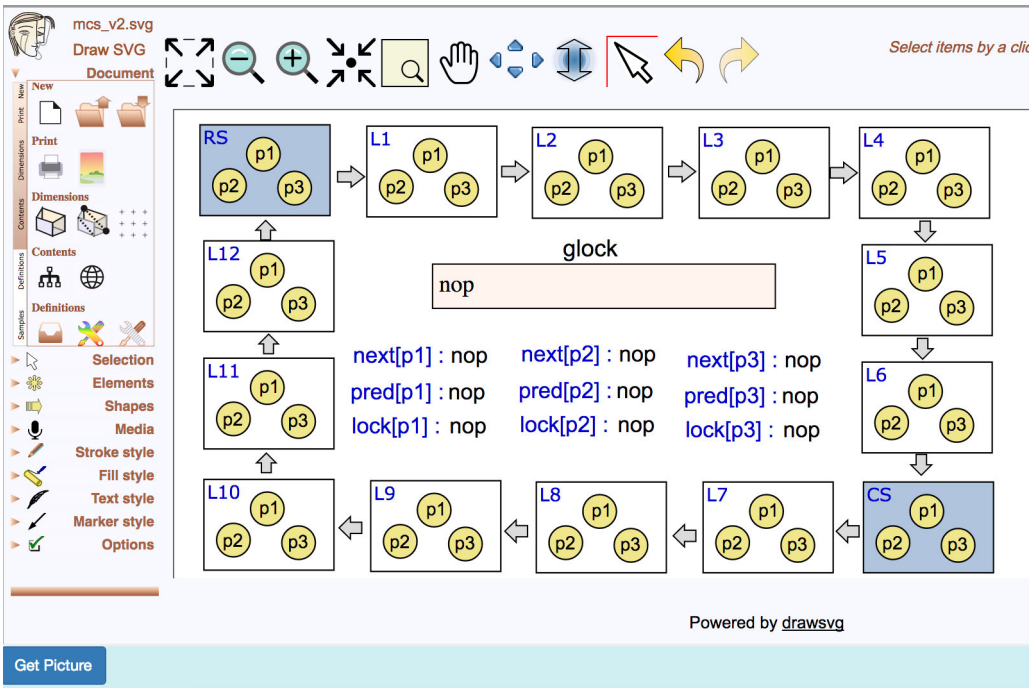

Figure 7.5: Picture of MCS Protocol

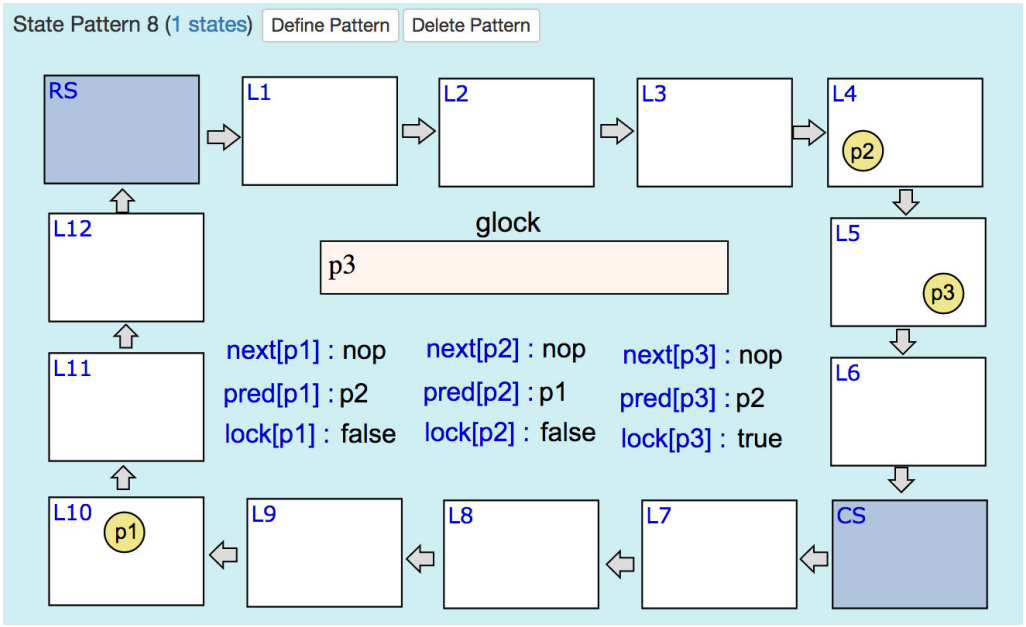

Figure 7.6: A state such that `p1` is at `l10`

MCS protocol enjoys the lockout freedom property when there are two processes. The lockout freedom property says whenever each process wants to enter the critical section,

it will eventually be there.

Model checking that the Kripke structure formalizing MCS protocol satisfies the lockout freedom property `lofree(p1)` for `p1` is conducted by reducing the term `modelCheck(init, lofree(p1))`. Maude model checker generates a counterexample.

MCS protocol uses two atomic operators `fetch&store` and `com&swap`. We model check the two properties for a variant of MCS protocol in which `comp&swap` is not used. The two lines at l8 and l9

l8: **if** comp&swap($glock, p$, nop)
l9: **goto** rs;

change to the following three lines:

l8: **if** $glock = p$ {
l8': $glock$:= nop;
l9: **goto** rs; }

Accordingly, the rewrite rule `chglk` is replaced with the following two rewrite rules:

```
rl [chglk'] : (glock: Q) (pc[P]: l8)
=> (glock: Q)
(pc[P]: (if Q == P then l8' else l10 fi)) .
rl [stglk] : (glock: Q) (pc[P]: l8')
=> (glock: nop) (pc[P]: l9) .
```

Model checking the two properties for the variant, the search command does not find any counterexamples for the mutual exclusion property but the LTL model checker finds a counterexample for the lockout freedom property even if a fair scheduler is adopted. Note that we can use exactly the same assumption used to model check that MCS protocol enjoys the lockout freedom property.

The counterexample generated by Maude LTL model checker for the lockout freedom under the use of a fair scheduler consists of a finite computation that consists of 17 states leading to an infinite loop such that a finite state sequence that consists of 9 states is repeated forever. Feeding the counterexample generated by Maude LTL model checker, the extended tool graphically animates it, repeating the loop part, which lets us realize only `p2` enters and leaves the critical section repeatedly while `p1` is waiting at l6 until $lock_{\texttt{p1}}$ becomes false. Fig. 7.8 shows the 26 pictures of the states composing the counterexample. The first 17 states is the finite computation, while the last 9 states is the finite state sequence that repeats forever, making the loop. Note that state 0 is the top state of the finite computation.

In state 8, `p1` is at l2 and is enqueuing it into the global queue, and `p2` is at l8 and is dequeuing the global queue. `p2` checks the condition of the **if** statement at l8. Since $glock$ is not `p2`, `p2` moves to l8'. In state 9, `p1` executes $pred_{\texttt{p1}}$:= fetch&store($glock$, `p1`); at l2, making $glock$ `p1` and $pred_{\texttt{p1}}$ `p2`. In state 9, since $pred_{\texttt{p1}}$ is `p2`, the predecessor of `p1` is `p2` in the global queue, meaning that `p1` has not been extracted from the global queue. In

what follows, since $pred_{\texttt{p1}}$ is not nop, `p1` sets $next_{\texttt{p2}}$ to `p1` and $lock_{\texttt{p1}}$ true, and waits at l6 until $lock_{\texttt{p1}}$ becomes false. In state 13, `p2` executes $glock$:= nop; at l8'. Therefore, in state 14, $glock$ is nop, meaning that the global queue is empty, although `p1` is waiting at l6 until $lock_{\texttt{p1}}$ becomes false. This is way `p1` is waiting at l6 forever and only `p2` enters and leaves the critical section repeatedly.

MCS protocol, and a variant in which comp&swap is naively disused have been analyzed with Maude and the state machine graphical animation tool. The tool can graphically animate any finite state sequence and any counterexample that consists of a finite state sequence leading a loop in which a finite state sequence repeats forever if they can be converted into what can be fed into the tool.

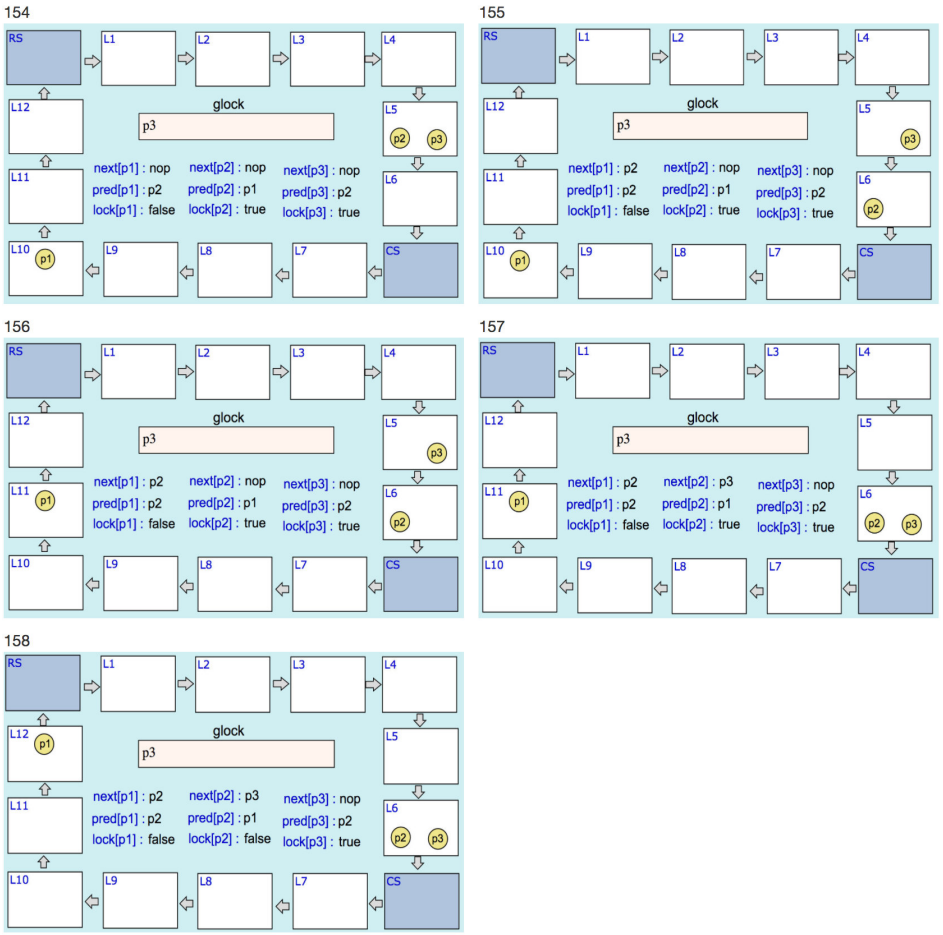

Figure 7.7: States 154, 155, 156, 157 and 158

The tool lets us know the state appear in the input finite computation at position 153. In the state, since `p1` is at l10, `p1` is dequeuing the global queue, while since `p2` and `p3` are `l4` and `l5`, `p2` and `p3` are enqueuing `p2` and `p3` into the global queue, respectively, but none of them has completed. Given a state number n, the tool displays the state at position n. We asked the tool to display the state at position 153 and play the animation from the state step by step. Fig. 7.7 shows the five states at positions 154, 155, 156, 157 and 158 from the top.

In state 153, `p2` executes the assignment at l4, setting $lock_{\texttt{p2}}$ true, and moves to l5 but has not yet completed enqueuing `p2` into the global queue. In state 154, `p2` executes the assignment at l5, setting $next_{\texttt{p1}}$ to `p2`, and moves to l6, when `p2` has eventually completed enqueuing `p2` into the global queue. In state 155, `glock` is `p3`, meaning that `p3` is the bottom element of the global queue but `p3` has not completed enqueuing `p3` into the global queue. In state 155, `p1` leaves the loop at l10 and moves to l11 but has not yet completed dequeuing the global queue. In state 156, `p3` executes the assignment at l5, setting $next_{\texttt{p2}}$ to `p3`, and moves to l6, when `p3` has eventually completed enqueuing `p3` into the global queue. In state 157, `p1` executes the assignment at l11, setting $lock_{\texttt{p2}}$ false, letting know `p2` is ready to enter the critical section, and moves to l12. In state 158, the global queue consists of `p2` and `p3` in this order because $lock_{\texttt{p2}}$ is `false`, $next_{\texttt{p2}}$ is `p3`, $lock_{\texttt{p2}}$ is `true`, $next_{\texttt{p2}}$ is `nop`, and `glock` is `p3`.

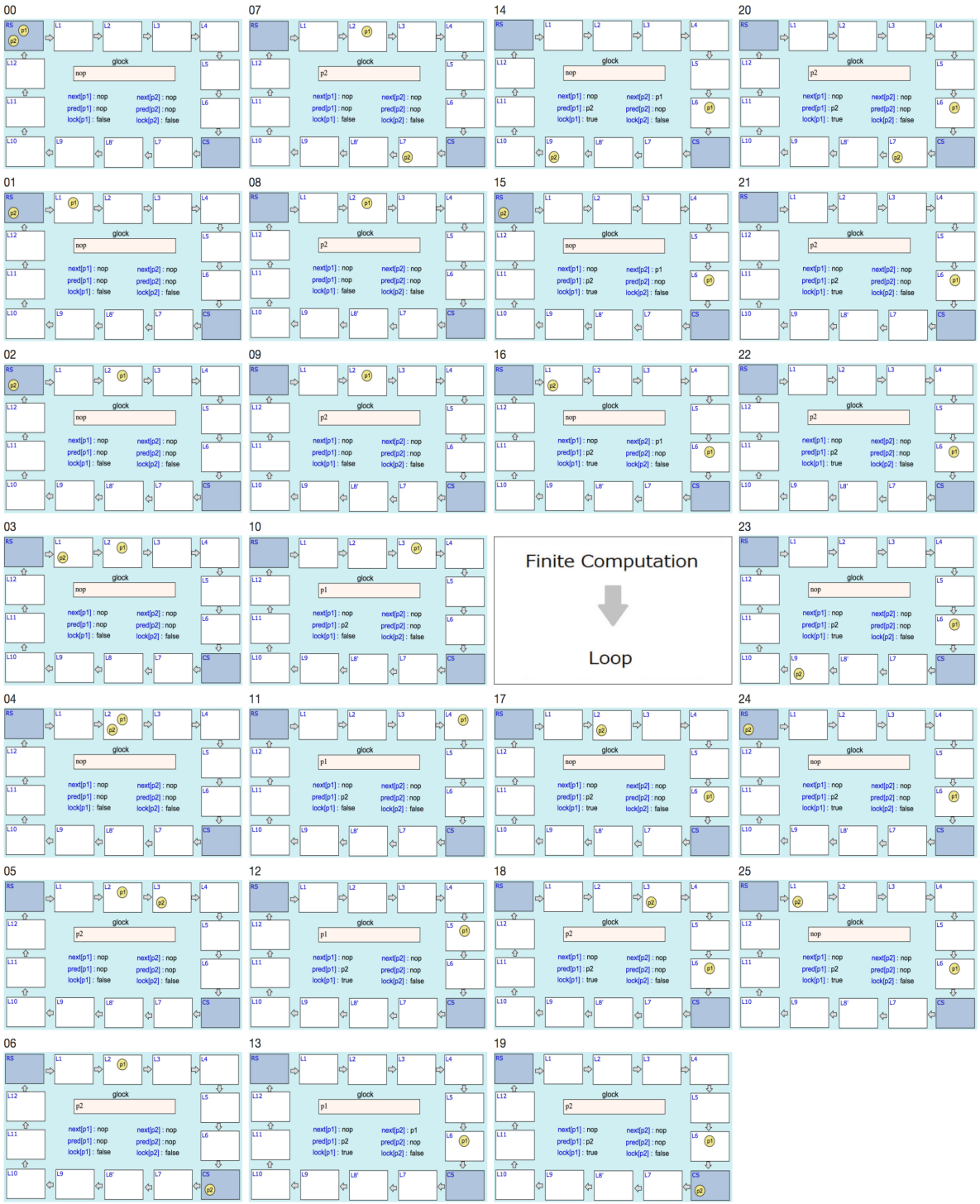

Figure 7.8: A counterexample for the lockout freedom property for MCS protocol in which comp&swap is not naively used.

Chapter 8

Related Work

Most formal specification languages, such as Z, B method and Event-B, are not executable, although some, such as VDM and VDM++, are semi-executable. Therefore, some researches have been carried out, making formal specifications written in such languages run, for example, by translating sub-sets of such languages into programming languages. Running formal specifications is called specification animation. Specification animation makes it possible to help human users get better understandings of formal specifications. Therefore, specification animation has been used to improve some other activities, such as refinement [6, 7], inspection and formal specification construction [15, 12], and software monitoring [13]. Although specification animation does not necessarily mean visual and graphical animations, some tools make it possible to play graphical animations [12]. The formal specification language we have used is Maude. Since Maude is executable, we do not need to develop any translators.

Although there is no generic graphical user interface (GUI) for Maude, a GUI for Maude-NPA [20] is a security protocol analysis tool was implemented in Maude. In verification process, the GUI for Maude-NPA animates completely the Maude-NPA search tree generation process. It allows users visualize the complete search tree, display representations nodes of the search tree graphically. By displaying the search tree, the tool support users to analyze a specification of a security protocol. Each state in the tree is displayed as a textual information and a graphical representation which help users easier to understand the analyzing process of the protocol.

Our tool is generic enough such that the tool is independent from Maude and can graphically animate any finite state sequence and any counterexample that consists of a finite state sequence leading a loop in which a finite state sequence repeats forever if they can be converted into what can be fed into the tool.

Some model checkers, such as Alloy [11] and PAT [22], are equipped with graphical animations of scenarios, such as counterexamples. Such graphical animations of counterexamples help human users get better understandings of the reason why the counterexamples occur. Such model checkers, however, do not allow human users to draw pictures used for graphical animations. Alloy and PAT do not allow users to adjust the speed of animations and select some states that satisfy some conditions and/or constraints from a counterexample.

Alloy is a relational logic-based specification language and analyzer. The Alloy analyzer contains a SAT-based bounded model checker. When Alloy finds a counterexample, it builds a graphical representation of the counterexample, which can be animated and run step by step. The picture or flame of animations is automatically created. PAT [22] is not only a collection of model checkers but also allows users to build their own model checkers. When PAT finds a counterexample, it builds a graphical representation of the counterexample as a directed graph such that vertices are states and edges are state transitions. The construction of the directed graph is animated step by step. It is possible to take a close look at each vertex or state.

On the other hand, many researchers have been convinced that (graphical) specification animation can help human users get better understandings of formal specifications, but to the best of our knowledge none of them have tried to utilize graphical specification animation for conjecturing lemmas in interactive theorem proving.

Chapter 9

Future Work

For the future work, we will improve and enhance the graphical animation tool to support users figure out useful state patterns quickly and display state machines intuitively and visually.

We will apply the tool to a concrete case study, tackle with the tool a non-trivial protocol such that we have not formally verified that it enjoys some invariants, finding out interesting state patterns and conjecturing lemmas from those state patterns to complete the formal verification. Thus, one piece of our future work is to recognize useful patterns from several graphically animated computations, conjecture useful lemmas from the animated computations and formally verify MCS protocol enjoys the mutual exclusion property and the lockout freedom property. We will formally verify with theorem proving that MCS enjoys properties, which requires lemmas. By looking at the graphical simulations, we will try to detect state patterns which is useful for conjecturing lemmas. The purpose of this work is to demonstrate the tool helps human users recognize patterns and conjecture lemmas from them. For detail, we will make a specification of MCS in CafeOBJ which is an algebraic specification language used for writting formal specifications and verifying properties of systems. Then, we will write proof score showing that MCS enjoys propertiesproperties, while observing the animation and finding some interesting characteristics and conjecturing some lemmas. Then, we will apply the tool to Paxos which is a protocol used for solving consensus in asynchronous systems.

On the other hand, there are some existing studies in which graphical specification animations would be used to help human users inspect formal specifications and make them better [15, 12]. Our main goal is to help human users comprehend counterexamples better, but better understandings of counterexamples must be able to be used to make formal specifications better. One piece of our future work is to investigate the relation between our way of using graphical animations and their way.

Chapter 10

Conclusion

We have developed the graphical animation of state machines tool which is available at the website https://tamntt.bitbucket.io/ Research/ *GraphicalAnimation*/). By displaying graphical animations of sequences of states, the tool help users to recognize some useful state patterns which can be used for conjecturing lemmas in interactive theorem proving. The tool can animate counterexamples generated by the Maude LTL model checker to help human users comprehend them. The tool also allows users to select some states that satisfy some conditions and/or constraints to support them detect patterns. Formally verifying that a system enjoys an invariant with interactive theorem proving, a human user first repeatedly conducts case splitting tasks based on come conditions and/or constraints and then may reach a case in which he/she needs to use some lemmas. The human users can use those conditions and/or constraints to make the tool filter out states in a finite computation. By this way, users can figure out useful lemmas used for theorem proving. The tool allows human users to design pictures (or flames) of animations. Therefore, intuitively understandable pictures could be used, helping human get better understandings of counterexamples and realize why they occur. Moreover, the layout of the places where each observable value is displayed can be decided by human users. We have written a meta-program in Maude to shorten counterexamples of properties, and a meta-program to generate long a sequence of states for long animations in which users can recognize patterns.

The experiment demonstrates the tool could help human users find out interesting state patterns. We conducted another case study in which animations of a state machine formalizing FQlock to demonstrated the usefulness of graphical animations of counterexamples. We also analyzed the MCS list-based queuing lock (MCS protocol), a mutual exclusion protocol, and some variants with Maude and the state machine graphical animation tool.

For future work, we will enhance the tool and conduct some more case studies in which we will tackle with the tool some protocols or systems such that we have not formally verified that they enjoy some invariants, finding out interesting state patterns and conjecturing lemmas from those state patterns to complete the formal verification.

Bibliography

[1] ACL2. The website of ACL2. `http://www.cs.utexas.edu/users/moore/acl2/`, 2017.

[2] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic*. LNCS 4350. Springer, 2007.

[3] Coq. The website of Coq. `https://coq.inria.fr/`, 2016.

[4] Leonardo de Moura, Sam Owre, and N. Shankar. The sal language manual. 2003.

[5] Joseph A. Goguen, Timothy Winkler, Jos Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing obj, 1993.

[6] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Refinement-animation for Event-B - towards a method of validation. In *ABZ 2010*, LNCS 5977, pages 287–301. Springer, 2010.

[7] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Validation of formal models by refinement animation. *Sci. Comput. Program.*, 78(3):272–292, 2013.

[8] Dung Tuan Ho, Min Zhang, and Kazuhiro Ogata. Case studies on extracting the characteristics of the reachable states of state machines formalizing communication protocols with inductive logic programing. In *ILP (Late Breaking Papers)*, pages 33–47, 2015.

[9] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Softw. Eng.*, 23(5):279–295, 1997.

[10] Isabelle. The website of Isabelle. `https://isabelle.in.tum.de/`, 2016.

[11] Daniel Jackson. *Software Abstraction (Revised edition)*. The MIT Press, 2012.

[12] Mo Li and Shaoying Liu. Integrating animation-based inspection into formal design specification construction for reliable software systems. *IEEE Trans. Reliability*, 65(1):88–106, 2016.

[13] Hui Liang, Jin Song Dong, Jing Sun, and W. Eric Wong. Software monitoring through formal specification animation. *ISSE*, 5(4):231–241, 2009.

[14] Joseph Liard. Draw SVG website. `http://www.drawsvg.org/`, 2015.

[15] Shaoying Liu. Validating formal specifications using testing-based specification animation. In *FormaliSE@ICSE 2016*, pages 29–35, 2016.

[16] Kenneth L. McMillan. *The SMV System*. Springer US, Boston, MA, 1993.

[17] John M. Mellor-Crummey and Michael L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1):21–65, 1991.

[18] Kazuhiro Ogata. Lecture 8 Analysis of Alternating Bit Protocol 2. Sinaia Shcool on Formal Verification of Software Systems (`http://www.jaist.ac.jp/~kokichi/class /SinaiaSchoolFVSS0803/`), 2008.

[19] PVS. The website of PVS. `http://pvs.csl.sri.com/`, 2014.

[20] Sonia Santiago, Carolyn L. Talcott, Santiago Escobar, Catherine A. Meadows, and José Meseguer. A graphical user interface for Maude-NPA. In *9th Spanish Conference on Programming and Languages (9th PROLE)*, Electr. Notes Theor. Comput. Sci., pages 3–20. Elsevier, 2009.

[21] Ambarish Sridharanarayanan Steven Eker, Jose Meseguer. The Maude LTL model checker. In *WRLA 2002*, ENTCS 71, pages 162–187. Elsevier, 2013.

[22] Jun Sun, Yang Liu, Jin Song Dong, and Jun Pang. PAT: Towards flexible verification under fairness. In *21st CAV*, LNCS 5643, pages 709–714. Springer, 2009.