Graphical Animations of State Machines

Tam Thi Thanh Nguyen and Kazuhiro Ogata

School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
{tamnguyen,ogata}@jaist.ac.jp

Abstract. Systems verification with interactive theorem proving (ITP) is one promising technology that could make software reliable, although it is necessary to utilize many other technologies, such as testing, so as to make software really reliable. Lemma conjecture is one of the most intellectual activities in ITP. While we were performing systems verification with ITP, we happened to find out some state patterns in which the reachable states of a state machine are classified and conjectured several useful lemmas from the state patterns to complete the formal verification. It would be very useful to make it possible to obtain such state patterns of a given state machine with a reasonable amount of efforts. This research utilizes human beings' ability to recognize patterns in various kinds of data, such as graphical animations. The research aims at designing and implementing a state machine graphical animation tool and confirming that human beings can recognize state patterns in state machine graphical animations.

Keywords: DRAW-SVG, graphical animations, lemmas, Maude, state machines, state patterns, SVG

1 Introduction

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. Of course, we need to use multiple technologies for this challenge. One possible promising technology is systems verification with interactive theorem proving. Hence, many proof assistants have been developed, such as PVS, ACL2, Isabelle and Coq. 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 not good enough. Thus, we need to make some more 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 $(\forall s \in R_M)\, p(s)$. $s_0, s_1, \ldots, s_n$ 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. Many requirements of software can be formalized as invariants. Since verifications of the other classes of properties often require invariants as lemmas, invariants are the most fundamental class of properties of state machines.

While we were formally verifying with interactive theorem proving that a state machine formalizing a communication protocol enjoys an invariant, we 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 [1]. 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 in those graphical animations, which are state patterns into which the reachable states of the state machine can be classified because finite computations of the state machine consist of reachable states of the state machine. The research aims at designing and implementing a state machine graphical animation tool and confirming that human beings can recognize state patterns in state machine graphical animations.

The rest of the paper is organized as follows. Sect. 2 mentions Alternating Bit Protocol (ABP) that is used as a running example in this paper and how to formalize ABP as a state machine and describe the state machine in Maude [2], a rewriting logic-based specification/programming language. Sect. 3 describes the motivating example for the research. Sect. 4 describes the requirements to be satisfied by the state machine graphical animation tool. Sect. 5 describes the design of the tool. Sect. 6 describes the implementation of the tool. Graphical animations of a state machine should be long enough so that human beings can recognize patterns in them. Sect. 7 describes a way to generate long computations. Sect. 8 reports on an experiment done with the tool. Sect. 9 mentions some existing related work and Sect. 10 concludes the paper.

2 Preliminaries

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 *bit*1 and a packet *pac* to be delivered. The receiver maintains one bit *bit*2 and a list *list* that contains

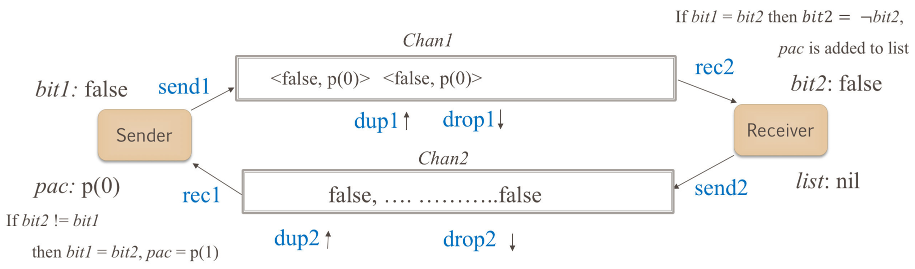


Fig. 1. A snapshot of ABP

the packets that have been received. Two unreliable channels *chan*1 and *chan*2 are used. They are unreliable in that their elements may be lost (or dropped) and duplicated. Fig. 1 shows a snapshot of ABP. There are eight actions done in ABP:

- send1: The sender puts a pair $\langle bit1, pac \rangle$ into *chan*1.
- rec1: The sender gets the top element Boolean b from *chan*2. If $b \neq bit1$, *bit*1 is complemented and *pac* is incremented.
- send2: The receiver puts *bit*2 into *chan*2.
- rec2: The receiver gets the top element $\langle b, p \rangle$ from *chan*1 if *chan*1 is not empty. If $b = bit2$, *bit*2 is complemented and p is added to *list*.
- drop1: The top element of *chan*1 is deleted if it is not empty.
- dup1: The top element of *chan*1 is duplicated if it is not empty.
- drop2: The top element of *chan*2 is deleted if it is not empty.
- dup2: The top element of *chan*2 is duplicated if it is not empty.

ABP can be formalized as a state machine M_{ABP}. There are many specification languages in which state machines can be described. We use Maude [2] to describe state machines. States can be expressed in various ways. In this paper, a state is expressed as an associative-commutative collection of name-value pairs. 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. 1. Therefore, each state of M_{ABP} is `(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 is expressed as `(chan1: empty) (chan2: empty) (bit1: false) (bit2: false) (pac: pac(0)) (list: nil)`. T_{ABP} is described as the eight rewrite rules:

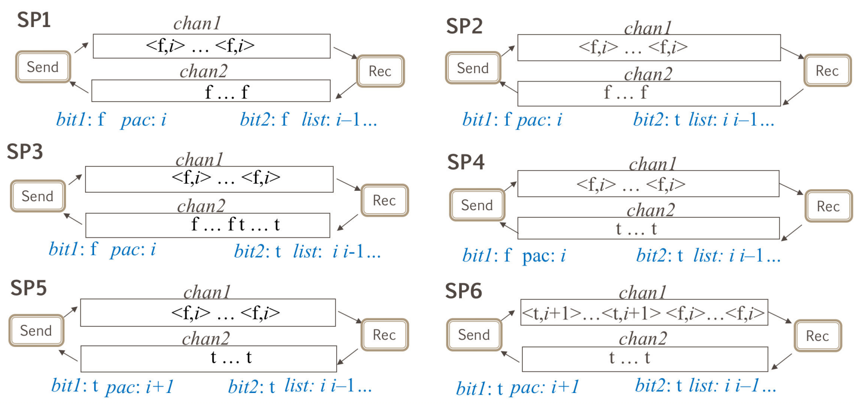


Fig. 2. Six state patterns of $R_{M_{\mathrm{ABP}}}$

```
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 ones of 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`.

3 Motivating Example

When we were formally verifying that ABP satisfies a desired property, we found that $R_{M_{\mathrm{ABP}}}$ is classified into six patterns shown in Fig. 2. From the six state patterns, we were able to conjecture several useful lemmas to complete the formal verification. For example, since SP3 is the only pattern such that *chan*2 consists of two different Booleans, SP3 allows us to conjecture the following lemma:

> if *chan*2 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 *chan*2 later than $b2$ is the same as $b2$ and $b2$ is the same as *bit*2;

and since SP6 is the only pattern such that *chan*1 consists of two different pairs, 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 efforts, 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 patterns in them 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 best use of this ability so as to conjecture lemmas in this research[1].

4 Requirements

There are several requirements the graphical animation tool of state machines should satisfy, among which are as follows:

1. Any state machines can be graphically animated.
2. Human users are intuitively allowed to design the frames (or pictures) of a graphical animation of a state machine.
3. Human users do not need to write any non-trivial pieces of code (or programs).
4. The speed of animations can be freely adjusted.
5. Very long animations can be made.

Requirement 1 implies that any applications can be graphically animated, provided that they can be formalized as state machines. Since it has been known that various kinds of systems can be formalized as state machines, the tool should be able to deal with various kinds of systems.

Each application must have an appropriate frame design of graphical animations of the state machine formalizing the application, and each human user must have his/her own preference. Frames (or pictures) should be able to be intuitively designed. Therefore, requirement 2 is mandatory.

[1] Our group has also been attempting [3] to automatically extract state patterns of a given state machine with Inductive Logic Programming, a combination of machine learning and logic programming.

It is fun to write programs, which is however error-prone as well as time-consuming and should be avoided if unnecessary. Hence, requirement 3 is reasonable.

If the speed of graphical animations of state machines is low, novice users could comprehend basic transitions of the state machines. Our goal to be achieved is, however, to help experienced engineers as well as researchers capture some non-trivial characteristics of the reachable states of state machines. To achieve this goal, the speed should be able to be adjusted. It would be essential to play a graphical animation of a state machine very quickly as well as smoothly so that some non-trivial characteristics of the reachable states could be observed. Therefore, requirement 4 is the most important for our goal.

Moreover, it would be necessary to play a very long animation so that some non-trivial characteristics of the reachable states could be observed. Accordingly, we need to satisfy requirement 5 as well.

5 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 one. 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 one. 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 fulfill requirement 1. 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. This is how requirement 1 is achieved.

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 to achieve requirement 2.

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). This is how we achieve requirement 3.

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. Requirement 4 is achieved in this way.

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 traversed $R_{M_{\mathrm{ABP}}}$ 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.

6 Implementation

6.1 Drawing state machine pictures

It would be possible to implement the tool from scratch, but take many efforts 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, text, and graphic images. It is useful to use

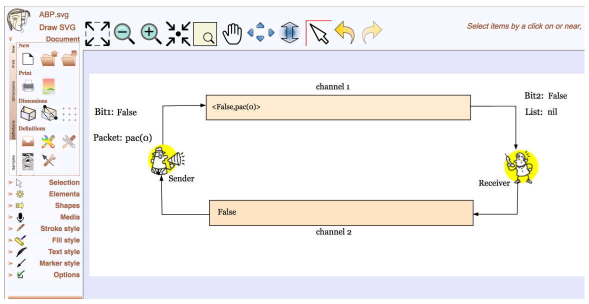


Fig. 3. A picture of M_{ABP}

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 [4], which we have used in this research. DRAW-SVG is a free online drawing application for designers and developers, making it possible to create fully standard compliant SVG. We use it as an integrated drawing tool within our website (`https://tamntt.bitbucket.io/Research/GraphicalAnimation/`) by using API based on Mozilla jsSchannel. The display of DRAW-SVG is supported by all currently available browsers except for Internet Explorer.

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, $chan1$ is set as the ID of the observable component ($chan1 : prq$) so that the queue prq of Boolean-packet pairs is displayed at the designated place on a state machine picture. Fig. 3 shows a picture of M_{ABP} drawn with the tool.

6.2 Input file format

The graphical animation tool does not deal with state machines themselves internally to achieve requirement 1. Instead, what are 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
chanl chan2 bitl bit2 pac list
###textDisplay
chanl::::REV::::<_,_>++++empty
###states
(chanl: empty chan2: empty bitl: false bit2: false pac: pac(0) list: nil) ||
(chanl: (< false,pac(0) > empty) chan2: empty bitl: false bit2: false pac: pac(0) list: nil) ||
(chanl: empty chan2: empty bitl: false bit2: true pac: pac(0) list: (pac(0) nil)) ||
(chanl: empty chan2: (true empty) bitl: false bit2: true pac: pac(0) list: (pac(0) nil)) ||
chanl: empty chan2: empty bitl: true bit2: true pac: pac(l) 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. These keys are used as IDs described in the last sub-section. 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
```

Tow 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.

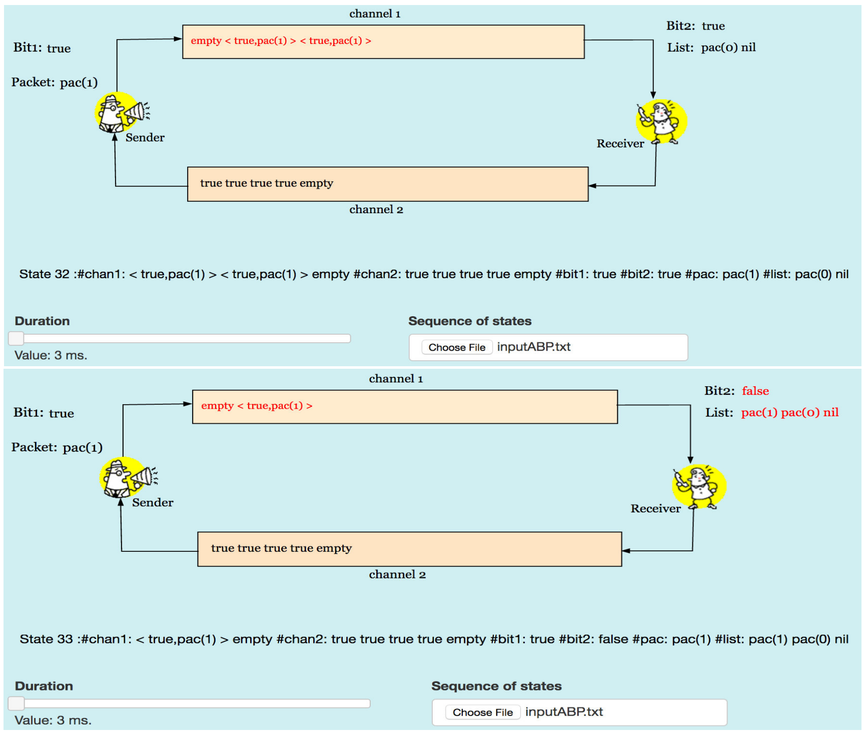
6.3 Running tool

A picture of a state machine has been drawn and/or got and an input file has been prepared and/or imported, and then 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 shows a state transition (done by rec2) from state 32 to state 33 in a finite computation of M_{ABP}.

6.4 The algorithm of graphical animation

The algorithm used in the tool is as follows:

```
Function: animation(svg, states, keys, textDisplay, duration)
 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 = empty;
   if(value1 != value2) attr is changed red color for text.
   else attr is changed black color for text.
   setTransition(svgText, attr, value2, duration, textDisplay[key])
```



```
State 32 :#chan1: < true,pac(1) > < true,pac(1) > empty #chan2: true true true true empty #bit1: true #bit2: true #pac: pac(1) #list: pac(0) nil
```

```
State 33 :#chan1: < true,pac(1) > empty #chan2: true true true true empty #bit1: true #bit2: false #pac: pac(1) #list: pac(1) pac(0) nil
```

Fig. 4. A step running of an animation

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/or got. 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'.

6.5 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 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 the constraint is as follows:

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

where `key` is the name of an observable component, `regex1`, `regex2`, ..., `regexn` 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 (called Const$i$ for $i = 1, 2, \ldots, 8$, respectively) 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
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 occurs more than once in the finite computation, the tool also finds seven different states in it. One of them is shown in Fig. 5.

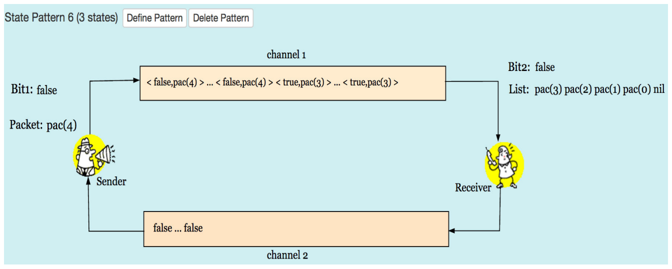

State Pattern 6 (3 states)
Define Pattern
Delete Pattern
channel 1
< false,pac(4) > ... < false,pac(4) > < true,pac(3) > ... < true,pac(3) >
Bit1: false
Packet: pac(4)
Sender
Bit2: false
List: pac(3) pac(2) pac(1) pac(0) nil
Receiver
false ... false
channel 2

Fig. 5. A state that satisfies Cond1, Const4 and Const6

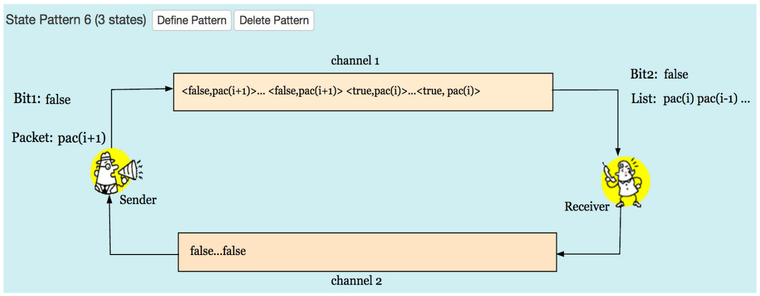

State Pattern 6 (3 states)
Define Pattern
Delete Pattern
channel 1
<false,pac(i+1)>... <false,pac(i+1)> <true,pac(i)>...<true, pac(i)>
Bit2: false
List: pac(i) pac(i-1) ...
Bit1: false
Packet: pac(i+1)
Sender
Receiver
false...false
channel 2

Fig. 6. A state pattern

6.6 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. 5, 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. 6 that is essentially equivalent to SP6 shown in Fig. 2.

7 Generation of Long Computations

Maude provides metaprorgamming functionalities. 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.

This algorithm is implemented in Maude as follows:

```
op selNxt : TermList Nat -> Term .
op subGenSeq : Module TermList Nat Bound -> TermList .
op genSeq : Module Term Nat Bound -> TermList .
eq selNxt(TList,R) = getTermByIndex(TList,R rem termListLength(TList)) .
eq subGenSeq(Mod,empty,R,B) = empty .
eq subGenSeq(Mod,TList,R,B)
 = genSeq(Mod,selNxt(TList,R),random(R quo 100000),B) .
eq genSeq(Mod,S,R,0) = empty .
eq genSeq(Mod,S,R,s(B))
 = S , subGenSeq(M,getAllSuccessors(Mod,S,getLabels(Mod)),R,B) .
```

where Mod, S, R, B and TList are Maude variables of the sorts Module, Term, Nat, Nat and TermList, respectively.

s(B) represents B+1 and can be used as the pattern as any positive natural number. getLabels(Mod) returns a collection of all labels of the rewrite rules in Mod. The function getAllSuccessors takes Mod, S representing a state and a collection of rewrite rule labels, 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 random generates a pseudo-random number based on the given seed. Based on the pseudo-random number generated, the next state is chosen by selNxt. 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

Table 1. Experimental results with FS150

–	Cnd1	+Cst1,5	+Cst2,6	+Cst1,7	+Cst2,8	+Cst3,5	+Cst4,6	+Cst3,7	+Cst4,8
#OS	50	37	37	0	0	18	18	0	0
#DSP	40	24	11	0	0	16	7	0	0
SP	–	–	SP1,5	–	–	–	SP6	–	–

–	Cnd2	+Cst1,5	+Cst2,6	+Cst1,7	+Cst2,8	+Cst3,5	+Cst4,6	+Cst3,7	+Cst4,8
#OS	39	18	18	21	21	0	0	0	0
#DSP	32	14	8	18	10	0	0	0	0
SP	–	–	SP2,4	–	SP3	–	–	–	–

it into a meta-represented term of the term. The way to generate finite computations can generate a finite computations up to about 100000 for M_{ABP}. Note that a finite computation made by `downTermList` has `||` `nil` as the end that should be deleted when it is used in an input file.

8 Experiment

We have used the finite computation FC150 of M_{ABP}.. Observing the animation from FC150 has made us find out some of the six state patterns shown in Fig. 2. 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 Const$j$ for $j = 1, 2, \ldots, 8$. When we used Cond1, the tool found 50 occurrences of the states that satisfied Cond1 among which there were 40 different states. When we used Const1 and Const5 as well, the tool found 37 occurrences of the states that satisfied Cond1, Const1 and Const5 among which there were 24 different states. When we instead used Const2 and Const6 as well, the tool found 37 occurrences of the states that satisfied Cond1, Const2 and Const6 among which there were 11 different state patterns. Taking a close look at those 11 different state patterns made us recognize SP1 and SP5 shown in Fig. 2. Table 1 shows the experimental results in which Cnd$i$, Cst$j$,k, #OS, #DSP, SP and SPj(,k) stand for Cond$i$, Const$j$ and Const$k$, the number of occurrences of states, the number of different states or state patterns, state patterns, and SPj (and SPk), respectively. 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.

9 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 have been used to improve some other activities, such as refinement [5, 6], inspection and formal specification construction [7, 8], and software monitoring [9]. Although specification animation does not necessarily mean visual and graphical animations, some tools make it possible to play graphical animations [8]. The formal specification language we have used is Maude. Since Maude is executable, we do not need to develop any translators.

Some model checkers, such as Alloy and PAT, 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.

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.

10 Conclusion

The tool is available at the website (`https://tamntt.bitbucket.io/Research/GraphicalAnimation/`). The experiment demonstrates the tool could help human users find out interesting state patterns. We also conducted another case study in which animations of a state machine formalizing Qlock, a mutual exclusion protocol, were observed and some interesting state patterns were found out. We will 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.

References

1. Ogata, K.: 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)
2. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. LNCS 4350. Springer (2007)
3. Ho, D.T., Zhang, M., Ogata, K.: 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). (2015) 33–47
4. Liard, J.: Draw SVG website. `http://www.drawsvg.org/` (2015)
5. Hallerstede, S., Leuschel, M., Plagge, D.: Refinement-animation for Event-B - towards a method of validation. In: ABZ 2010. LNCS 5977, Springer (2010) 287–301
6. Hallerstede, S., Leuschel, M., Plagge, D.: Validation of formal models by refinement animation. Sci. Comput. Program. **78** (2013) 272–292
7. Liu, S.: Validating formal specifications using testing-based specification animation. In: FormaliSE@ICSE 2016. (2016) 29–35
8. Li, M., Liu, S.: Integrating animation-based inspection into formal design specification construction for reliable software systems. IEEE Trans. Reliability **65** (2016) 88–106
9. Liang, H., Dong, J.S., Sun, J., Wong, W.E.: Software monitoring through formal specification animation. ISSE **5** (2009) 231–241