International Journal of Software Engineering and Computer Systems (IJSECS)
ISSN: 2289-8522, Volume 4 Issue 2, pp. 1-18, August 2018
©Universiti Malaysia Pahang
DOI: https://doi.org/10.15282/ijsecs.4.2.2018.1.0045

GUESSING, MODEL CHECKING AND THEOREM PROVING OF STATE MACHINE PROPERTIES – A CASE STUDY ON QLOCK

May Thu Aung[1], Tam Thi Thanh Nguyen[2], Kazuhiro Ogata[2]

[1]Faculty of Information Science,
University of Information Technology, Parami Road, Hlaing Campus, Yangon, Myanmar.
[2]School of Information Science,
Japan Advanced Institute of Science and Technology, 1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan.

ABSTRACT

It is worth understanding state machines better because various kinds of systems can be formalized as state machines and therefore understanding state machines has something to do with comprehension of systems. Understanding state machines can be interpreted as knowing properties they enjoy and comprehension of systems is interpreted as knowing whether they satisfy requirements. We (mainly the second author) have developed a tool called SMGA that basically takes a finite sequence of states from a state machine and generates a graphical animation of the finite sequence or the state machine. Observing such a graphical animation helps us guess properties of the state machine. We should confirm whether the state machine enjoys the guessed properties because such guessed properties may not be true properties of the state machine. Model checking is one possible technique to do so. If the state machine has a fixed small number of reachable states, model checking is enough. Otherwise, however, it is not. If that is the case, we should use some other techniques to make sure that the system enjoys the guessed properties. Interactive theorem proving is one such technique. The paper reports on a case study in which a mutual exclusion protocol called Qlock is used as an example to exemplify the abovementioned idea or methodology.

Keywords: graphical animations of state machines, model checking, theorem proving, invariant properties

INTRODUCTION

A state machine is a mathematical structure, which can be used to formalize various kinds of systems, such as concurrent systems, distributed systems and real-time systems. It is worth comprehending a system under development better, which could be reduced to understanding a state machine that formalizes the system. Understanding a state machine could be interpreted as knowing properties the state machine enjoys. The more state machine properties we know, the better we understand the state machine. We (mainly the second author) have developed a tool called SMGA (Nguyen & Ogata, 2017a) that basically takes a finite sequence of states from a state machine and generates a graphical animation of the finite sequence or the state machine, where SMGA stands for State Machine Graphical Animation. Observing such a graphical animation helps us guess properties of the state machine. We should confirm whether the state machine enjoys the guessed properties because such guessed properties may

not be true properties of the state machine. Model checking is one possible technique to do so. If the state machine has a fixed small number of reachable states, model checking is enough. Otherwise, however, it is not. If that is the case, we should use some other techniques to make sure that the system enjoys the guessed properties. Interactive theorem proving is one such technique.

This paper reports on a case study in which Qlock, a mutual exclusion protocol, is used to exemplify the abovementioned idea or methodology. The paper is an extended and revised version of the paper (Aung et al. 2018) presented and published at ICSCA 2018. Qlock can be regarded as an abstract version of the Dijkstra Binary Semaphore. Qlock is first formalized as a state machine, which is specified in Maude (Clavel et al., 2007), a rewriting logic-based computer language and tool. Finite sequences of states can be generated from the Maude specification of Qlock. From finite sequence of states, SMGA produces graphical animations of the state machine formalizing Qlock. The graphical animations help humans guess interesting characteristics occurring in the graphical animations. Maude is equipped with model checking facilities, one of which is the search command that can be used as an invariant model checker. This paper only focuses on invariant properties of state machines because invariant properties are the most fundamental and often used as lemmas to prove other classes of properties, such as leads-to properties. Guessed characteristics are formalized as invariant properties so that the search command can be used to check if the state machine enjoys them. We then use theorem proving to formally verify that the state machine surely enjoys the invariant properties by writing what are called proof scores (Goguen, 1990; Ogata & Futatsugi, 2003) in CafeOBJ (Diaconescu & Futatsugi, 1998), an algebraic specification language and tool. Note that both Maude and CafeOBJ are direct successor of OBJ3 (Goguen, et al., 2000), the most famous algebraic specification language and tool, and then are sibling languages and tools.

The rest of the paper is organized as follows: RELATED WORK in which some related work is mentioned; PRELIMINARIES in which some preliminaries, such as state machines, are mentioned; METHODOLOGY in which we report on the case study where Qlock is used to exemplify the methodology; CONCLUSION in which we conclude the paper.

All files, such as specification of Qlock in Maude, used in the paper are available on the webpage:

http://www.jaist.ac.jp/~ogata/code/gmctp-qlock/

The files are (1) a template picture of Mod3 (an example used in PRELIMINARIES) for SMGA, (2) a template picture of Qlock for SMGA, (3) an/ input file to SMGA for Mod3, (4) an input file to SMGA for Qlock, (5) specification of Mod3 in Maude, (6) specification of Mod3 in CafeOBJ (including proof score), (7) specification of Qlock in Maude, (8) specification of Qlock in CafeOBJ, and (9) seven files of proof scores for seven Qlock properties. On the website, you can find the link to SMGA.

RELATED WORK

Qlock has been used as one example to demonstrate how to prove that systems formalized as state machines enjoy properties by writing proof scores in CafeOBJ. How to write proof scores in CafeOBJ showing that Qlock enjoys the mutual exclusion property (referred as Prop. 1 later in this paper) and a lemma needed (referred as Prop. 2 later in this paper) is described in (Ogata & Futatsugi, 2008, 2013). Liveness properties

as well as invariant properties can be formally verified by writing proof scores in CafeOBJ (Ogata & Futatsugi, 2013; Preining, 2014; Yoshida, 2015).

A graphical user interface for Maude-NPA has been developed (Santiago, 2009). Maude-NPA is a high-level security protocol analysis language and system implemented on the top of Maude. The graphical user interface is dedicated to Maude-NPA and then cannot be used for our purpose, namely that graphical animations of state machines can be displayed, helping humans guess properties of the state machines based on the graphical animations.

Specification animation has been actively studied. Specification animation means making formal specification executable by translating formal specifications into executable programs because most formal specification languages are not executable. Specification animations have been used to inspect formal specifications (Li & Liu, 2016), to monitor software through formal specification animation (Liang, et al., 2016), to validate formal models by refinement animation (Hallerstede, et al., 2016) and to make a specification-based testing better (Nagoya & Liu, 2017). Maude is inherently executable and then it is unnecessary to translate Maude specifications into executable programs.

Some model checkers, such as Alloy (Jackson, 2012) and PAT (Sun, 2009), are equipped with some graphical facilities such that counterexamples are graphically displayed. Human users are allowed to interact with such graphically displayed counterexamples, such as forward and backward step execution and investigating each state. Their graphical animations of counterexamples, however, have not been used to help humans guess properties of state machines.

Few researches have been conducted in which graphical animations of state machines are used to help human users guess or conjecture lemmas needed to complete formal proofs. The case study reported in this paper exhibits a positive potential that graphical animations of state machines could be used for that purpose. There are, however, a lot to do left so as to make sure that our claim that graphical animations of state machines can help human users conjecture useful lemmas for theorem proving would be true. One of them is to conduct more case studies. We (mainly the second author) have been conducting a case study in which it is theorem proved that MCS (Mellor-Crummey & Scott, 1991), a list-based queuing mutual exclusion protocol, enjoys the mutual exclusion property (that corresponds to Prop. 1 of Qlock). The proof requires many lemmas. As a preliminary research, the second and third authors guessed and confirmed some MCS properties with SMGA and Maude (Nguyen & Ogata, 2017b). MCS is not just a laboratory-level mutual exclusion protocol but has been really used in many Java virtual machines. This is why Mellor-Crummey & Scott were awarded the 2006 Edsger W. Dijkstra Prize in Distributed Computing.

PRELIMINARIES

A state machine $M \triangleq \langle S, I, T \rangle$ consists of a set S of states, the set $I \subseteq S$ 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 and may be written as $s \rightarrow s'$, where s' is called a successor state of s with respect to (wrt) M. The set R of reachable states wrt M is inductively defined as follows: $I \subseteq R$ and if $s \in R$ and $(s, s') \in T$, then $s' \in R$. A state predicate p is called an invariant property wrt M if and only if (iff) p holds in all reachable states wrt M, namely $(\forall s \in R) p(s)$. States can be expressed in various ways. States are characterized by some values and then it suffices to observe those values in a way to express states. We

use two ways to express states. The first way to do so in to record those values in name-value pairs called observable components, which are similar to entries of maps or dictionaries and states are expressed as associative-commutative collections (called soups) of observable components. For example,

$$(\mathrm{pc[p1]}:\mathrm{rs})(\mathrm{pc[p2]}:\mathrm{rs})(\mathrm{pc[p3]}:\mathrm{rs})(\mathrm{pc[r4]}:\mathrm{rs})(\mathrm{pc[r5]}:\mathrm{rs})(\mathrm{queue}:\mathrm{empty})$$

is a soup of observable components used for Qlock, in which there are six observable components. For example, $(\mathrm{pc[p3]}:\mathrm{rs})$ is an observable component where $\mathrm{pc[p3]}:$ is the name and rs is the value, meaning that the process p3 is at label rs. The second way to do so is to use functions (called observers) that take states and some other parameters and return values that characterize states. For example, we can use an observer pc that takes a state s and a process ID p and returns the label $\mathrm{pc}(s, p)$ at which the process is.

Let us consider a simple system called Mod3, which will be formalized as a state machine M_{Mod3}. Mod3 has one value referred as val that is a natural number and whose initial value is 0. Each state of Mod3 can be expressed as one observable component $(\mathrm{val}: val)$, where val: is the name and val is the value. S_{Mod3} is $\{(\mathrm{val}: val) \mid val \in \mathrm{Nat}\}$, where Nat is the set of natural numbers. I_{Mod3} is $\{(\mathrm{val}: 0)\}$, where there is one initial state $(\mathrm{val}: 0)$. T_{Mod3} is $\{((\mathrm{val}: 0), (\mathrm{val}: 1)), ((\mathrm{val}: 1), (\mathrm{val}: 2)), ((\mathrm{val}: 2), (\mathrm{val}: 0))\}$. Note that S_{Mod3} is not finite, while R_{Mod3} is $\{(\mathrm{val}: 0), (\mathrm{val}: 1), (\mathrm{val}: 2)\}$. We suppose that the function get takes $(\mathrm{val}: val)$ and returns val and we can guess that $\mathrm{get}((\mathrm{val}: val)) < 3$ is an invariant property wrt M_{Mod3}.

Maude makes it possible to use any user's preferred notation to express states. For example, we can use exactly the same notation to express states of M_{Mod3} as we used in the last paragraph: $(\mathrm{val}: val)$. As described, generally, soups of observable components are used to express states. Note that a single observable component is a singleton soup that only consists of the observable component. Transitions are specified in terms of rewrite rules. For example, T_{Mod3} is specified as the following rewrite rule:

$$\mathrm{rl\ [upd]} : \{(\mathrm{val}: \mathrm{X})\} \Rightarrow \{(\mathrm{if\ X} < 2\ then\ X + 1\ else\ 0\ fi)\}\ .$$

where X is a Maude variable of natural numbers. Given a state s, {s} obtained by enclosing s with { and } is called a configuration. Rewrite rules specifying transitions are written as those from configurations to configurations so as to avoid some subtle issues. The rule upd says that if $\mathrm{X} < 2$, then X is incremented. Otherwise, the value of the val: observable component is 0. The Maude search command can be used to model check invariant properties wrt a state machine. It is in the form:

$$\mathrm{search}\ [n]\ \mathrm{in}\ Module : s \Rightarrow\ast\ p\ \mathrm{such\ that}\ c\ .$$

where $Module$ is a Maude specification of a state machine under model checking, s is a given state (typically an initial state of the state machine), p is a pattern and c is a condition. The search command searches the reachable states from s for at most n states that can match p and make c true. Typically, n is 1 and the nagation of the state predicate used to express the invariant property concerned is expressed as p and c. The condition part "such that c" can be omitted. The invariant property $\mathrm{get}((\mathrm{val}: val)) < 3$ wrt M_{Mod3} can be model checked with the search command as follows:

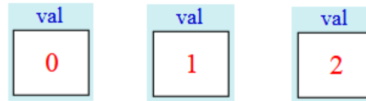
$$\mathrm{search\ [1]in\ Mod3} : \{\mathrm{init}\} \Rightarrow\ast \{(\mathrm{val}: \mathrm{X})\}\ \mathrm{such\ that\ (not\ X} < 3)\ .$$

where init equals (val: 0). Maude does not find any counterexamples, and then because there are only three reachable states from init, which is the only initial state of M_{Mod3}, we have formally verified that M_{Mod3} enjoys the invariant property.

From the Maude specification of M_{Mod3}, a finite sequence of states can be generated. For example, the following is an example:

$$(\mathrm{val}: 0)\ ||\ (\mathrm{val}: 1)\ ||\ (\mathrm{val}: 2)\ ||\ (\mathrm{val}: 0)\ ||\ (\mathrm{val}: 1)\ ||\ (\mathrm{val}: 2)\ ||\ (\mathrm{val}: 0)$$

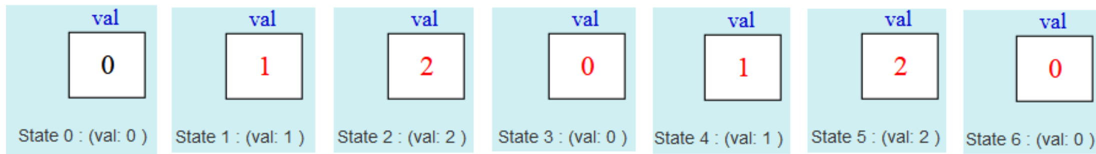
where the sequence consists of seven states, the leftmost one is the first (initial) state and the rightmost is the final one. For each state, a picture could be designed and drawn. For (val: 0), (val: 1) and (val: 2), for example, the following three pictures are designed and drawn, respectively:



Let sp0, sp1 and sp2 be the three pictures, respectively. From the finite sequence of states, then, we can generate the finite sequence of pictures:

$$\text{sp0} \mid\mid \text{sp1} \mid\mid \text{sp2} \mid\mid \text{sp0} \mid\mid \text{sp1} \mid\mid \text{sp2} \mid\mid \text{sp0}$$

Such a finite sequence of pictures can be regarded as a movie film. Playing such a movie film, we can watch a graphical animation of state machines. This is the basic idea on which SMGA, a state machine graphical animation tool, produces graphical animations of state machines. Note that we do not need to draw all pictures for all possible states but it suffices to design and draw one template picture, for example, for (val: val), and then SMGA automatically produces each concrete state picture. Once a template picture is designed and drawn, basically feeding a finite sequence of states, SMGA produces a graphical animation of the finite sequence. For example, from the finite sequence of states, the graphical animation in which the following seven pictures appear sequentially in the order is produced:



Observing the animation, we can guess that the value never becomes three or greater, namely that the value is always less than three, although we have already confirmed (and actually verified) this with Maude.

CafeOBJ can be used to theorem prove that a state machine enjoys an invariant property by writing what are called proof scores. To this end, it is necessary to specify a state machine in CafeOBJ as what is called an observational transition system (OTS) style. As briefly mentioned, in the OTS style, each value that characterizes states is observed by applying a function called an observer to the states, a set of transitions is represented by a function called an action (or a transition), and how to change each value that characterizes states by applying an action to the states is specified in terms of equations. M_{Mod3} is specified in CafeOBJ as an OTS style as follows:

```
op init : -> Sys {constr} .
op upd : Sys -> Sys {constr} .
op val : Sys -> Nat .
eq val(init) = 0 .
eq val(upd(S)) = (if val(S)
          < 2 then val(S) + 1 else 0 fi) .
```

Sys is the sort (or type) representing the set of reachable states that are constructed from the constant init and the function upd as indicated by constr that stands for constructor. The constant init represents an arbitrary initial state. Nat is the sort representing the set of natural numbers. In this paper, a sort and the set denoted by the sort are interchangeably used. In this example, there is one initial state. The function upd is an action, which is the only one in this example. The function val is an observer, which is the only one in this example. The first equation says that the value observed by the observer val in the initial state is 0. S is a variable of Sys. The second equation says that in the successor state upd(S) obtained by applying the action upd to S, the value observed by the observer val becomes val(S) + 1 if val(S) is less than 2, and it becomes 0 otherwise.

Let us define the following state predicate:

eq inv1(S) = (val(S) < 3) .

We can theorem prove that inv1 is an invariant property wrt M_{Mod3} by structural induction of the reachable state. The proof is first divided into the base case and the induction case. For the base case, we write the following program in CafeOBJ that is called (a fragment of) proof score:

```
open MOD3 .
  red inv1(init) .
close
```

where MOD3 is the CafeOBJ specification of M_{Mod3}, the open command makes a given specification available, the close command indicates the end of the use of the module and the red(uction) command reduces (or simplifies) a given expression (term) by using equations as left-to-right rewrite rules. CafeOBJ returns true for this proof score, meaning that the base case has been successfully proved. For the induction case, we write the following proof score in CafeOBJ:

```
open MOD3 .
  op s : -> Sys .
  eq [IH : nonexec] : inv1(s) = true .
  red inv1(s) implies inv1(upd(s)).
close
```

where the constant s of Sys represents an arbitrary state. The equation is the induction hypothesis but is not used as a left-to-right rewrite rules as indicated by the :nonexec attribute. Instead, the induction hypothesis is used as inv1(s) in "inv1(s) implies inv1(upd(s))." CafeOBJ does not return true for this proof score, meaning that we need to do some more to complete the proof. Typically, there are two kinds of things to do: (1) case splitting and (2) lemma conjecture and use. For this specific example, the induction case is split into three sub-cases based on val(s) < 3 and val(s) < 2, which correspond to the following three fragments of proof score:

```
open MOD3 .
 op s : -> Sys .
 eq [IH : nonexec] : inv1(s) = true .
 eq val(s) < 3 = true . eq val(s) < 2 = true .
 red inv1(s) implies inv1(upd(s)).
close
```

```
open MOD3 .
 op s : -> Sys .
 eq [IH : nonexec] : inv1(s) = true .
 eq val(s) < 3 = true . eq val(s) < 2 = false .
 red inv1(s) implies inv1(upd(s)).
close
```

```
open MOD3 .
 op s : -> Sys .
 eq [IH : nonexec] : inv1(s) = true .
 eq val(s) < 3 = false .
 red inv1(s) implies inv1(upd(s)).
close
```

CafeOBJ returns true for each of the three fragments, but the first sub-case needs one lemma about natural numbers, which is as follows:

$$\text{ceq X} + 1 < 3 = \mathit{true\ if\ X} < 2 \,.$$

where X is a CafeOBJ variable of natural numbers. The lemma is written as a conditional equation, which says that if X $<$ 2 is true, then X $+$ 1 $<$ 3 is true. Accordingly, we have successfully proved that inv1 is an invariant property wrt M_{Mod3}.

METHODOLOGY

Qlock is a mutual exclusion protocol and can be regarded as an abstract version of the Dijkstra Binary Semaphore. The pseudo-code for each process i as follows:

```
Loop: "Remainder Section"
 rs: enq(queue, i);
 ws: repeat until top(queue) = i;
      "Critical Section"
 cs: dequeue(queue);
```

where $queue$ is a queue of process IDs shared by all processes participating in Qlock and is atomic in that the functions enq, top and deq for queues are atomic. We suppose

that queue is used in neither "Remainder Section" and "Critical Section." Each process is located at rs (remainder section), ws (waiting section) or cs (critical section). Initially, each process i is located at rs and queue is empty. When a process i wants to enter "Critical Section," i first enqueues its ID into queue, next waits at ws until the top of $queue$ is i, then enters "Critical Section" if the top of $queue$ is i, and finally goes back to "Remainder Section," which is repeated.

One desired property Qlock should enjoy is what is called the mutual exclusion property, which is that there is always at most one process in "Critical Section." Some properties of a state machine M_{Qlock} formalizing Qlock including the mutual exclusion property will be guessed based on graphical animations of M_{Qlock}, confirmed by model checking and theorem proved.

Specification of Qlock in Maude

The values that characterize states of M_{Qlock} are the value (a queue of process IDs) stored in $queue$ and the each process location (rs, ws or cs). When there are five processes p1, p2, p3, p4 and p5, each state is expressed as

(queue: q) (pc[p1]: l1) (pc[p2]: l2) (pc[p3]: l3) (pc[p4]: l4) (pc[p5]: l5)

where queue: and each pc[pi]: are names, q and each li are values, q is a queue of process IDs, and each li is rs, ws or cs. Initially, q is empty and each li is rs. Let init be the expression (term) denoting the initial state.

T_{Qlock} is specified as the following rewrite rules:

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

where Q is a Maude variable of process ID queues, I is a Maude variable of process IDs and S is a Maude variable of states (or state fragments). I Q denotes a queue such that I is the top element and Q is deq(I Q). For example, p1 p2 p3 empty is the queue such that p1, p2 and p3 are the first, second and third elements and empty is the empty queue. Let QLOCK refer to the Maude specification of M_{Qlock}.

Guessing and Confirmation of some Qlock Properties

The picture of the initial state used by SMGA is shown in Fig. 1. There are three rectangles that correspond to rs, ws and cs, respectively. The picture allows us to immediately recognize that there are five processes in rs and no process in both ws and cs. Because a process moves to ws, cs and rs from rs, ws and cs, respectively, there are three arrows from the former to the latter, respectively. $queue$ is represented as a long pentagon laid down such that its head faces right. In the initial state, $queue$ is empty and then there is nothing in the pentagon. If there are some in the pentagon, what is located at the right-most is the top element of $queue$.

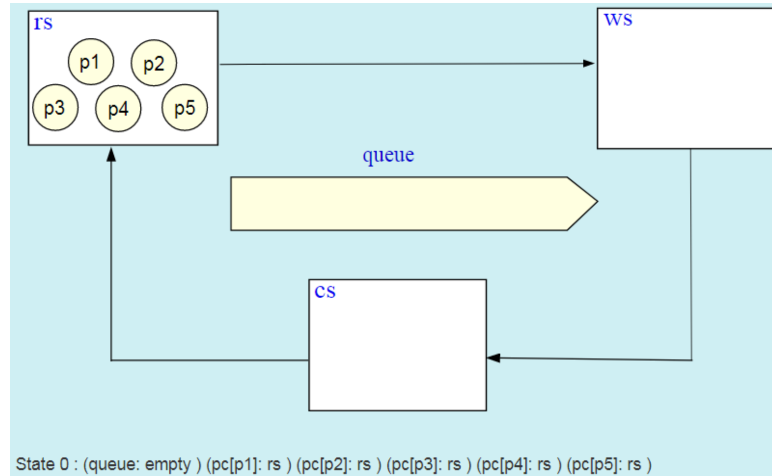


Figure 1. Initial state of Qlock.

We generate a finite sequence of states such that it consists of 1000 states based on the Maude specification of Qlock with Maude, feeding it to SMGA that produces the graphical animation of the sequence of states. The ith state in the sequence is expressed as si. Fig. 2 shows the six states in row from s993 to s998.

Observing the graphical animation produced by SMGA makes us guess that there is always at most one process at cs. For example, among the states shown in Fig. 2 there is one process in s993 and s997, and there is no process in the other four states. This guessed property is called Prop. 1 (which is actually the mutual exclusion property). The Maude search command can be used to confirm Prop. 1 as follows:

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

The search tries to find a state such that there are two processes I and J whose locations are L1 and L2, respectively, and it is not the case that if L1 is cs, then L2 is not cs, namely that both L1 and L2 are cs. Maude checks all possible combination of two processes I and J but not some specific combinations, such as p1 and p2. No counterexample is found and then Qlock enjoys Prop. 1 (the mutual exclusion property) when there are five processes. Note that the model checking result does not guarantee that Qlock surely enjoys the Prop. 1. For example, the result does not guarantee that Qock enjoys Prop. 1 when there are 100 processes.
The animation makes us recognize that whenever there is a process at cs, its process ID is the top of queue. For example, please take a look at s995 and s997 in Fig. 2. The guessed property is called Prop. 2, which is confirmed by the following search command:

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

The search tries to find a state such that there is a process I whose location is L1, the content of queue is Q, and it is not the case that if L1 is cs, then the top of Q is I. No counterexample is found and then Qlock enjoys Prop. 2 when there are five processes.

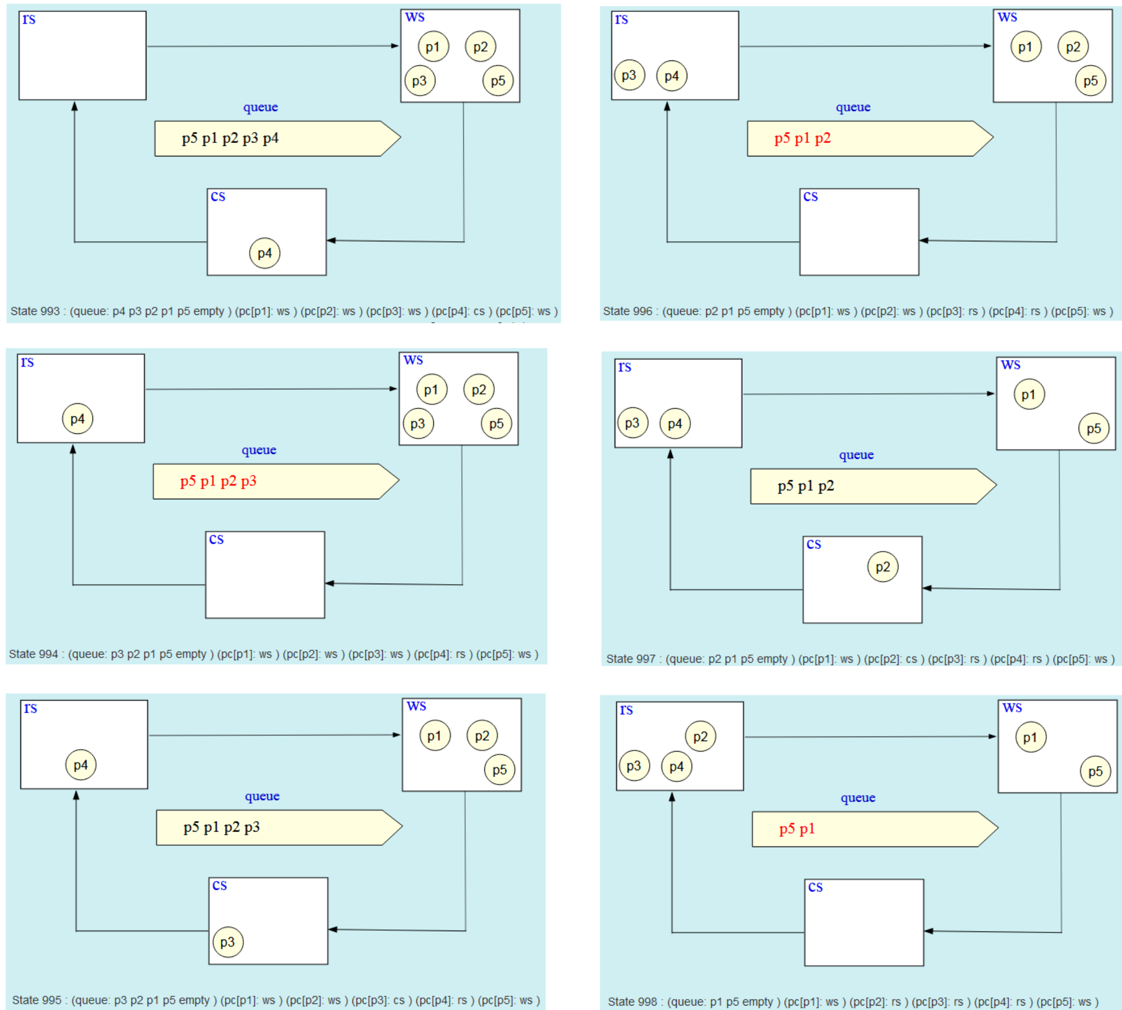


Figure 2. Six states in row from s993 to s998.

The animation also makes us recognize that whenever a process is at rs, it is not in *queue*. For example, please take a look at the five states from s994 to s998. The guessed property is called Prop. 3, which is confirmed by the following search command:

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

The search tries to find a state such that there is a process I whose location is L1, the content of queue is Q, and it is not the case that if L1 is rs, then I is not in Q. No counterexample is found and then Qlock enjoys Prop. 3 when there are five processes.

The animation makes us guess some more properties as well, among which are as follows:

- Prop. 4: Whenever a process is not in queue, it is at rs.
- Prop. 5: Whenever a process is at ws or cs, it is in queue.
- Prop. 6: Whenever a process is in queue, it is at ws or cs.

The three guessed properties are confirmed by the following three search commands:

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

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

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

Each search does not find any counterexamples. Therefore, Qlock enjoys Prop. 4, Prop. 5 and Prop. 6 when there are five processes.

Specification of Qlock in CafOBJ

The Maude search command can be used to find counterexamples of invariant properties but basically does not guarantee that systems, such as Qlock, enjoys invariant properties in all possible situations. To make it sure that Qlock enjoys the six guessed properties, CafOBJ could be used. We first specify Qlock in CafeOBJ as an OTS style. Let QLOCK be the specification.

We use the following two observers with which we observe the values that characterize states of M_{Qlock}:

```
op pc    : Sys Pid -> Label .
op queue : Sys -> Queue .
```

where Sys is the sort representing the set of reachable states wrt M_{Qlock}, Pid is the sort representing the set of process IDs, Label is the sort representing the set of labels (rs, ws and cs) and Queue is the sort representing the set of process ID queues. We use the following three actions to specify the transitions of M_{Qlock}:

```
op want : Sys Pid ->
 Sys {constr} .
op try  : Sys Pid -> Sys {constr} .
op exit : Sys Pid -> Sys {constr} .
```

A set of equations that s[...]ved by pc and queue is as follows:

```
ceq pc(exit(S, I), J) = (if I = J then rs else pc(S, J) fi) if pc(S, I) = cs .
ceq queue(exit(S, I)) = deq(queue(S)) if pc(S, I) = cs .
ceq exit(S, I) = S if not pc(S, I) = cs .
```

The two sets of equations for the other two actions want and exit can be defined likewise. An arbitrary initial state is represented as init, which is declared as follows and the values observed by pc and queue in init are defined as follows:

```
op init : -> Sys {constr} .
eq pc(init, I)  =  rs .
eq queue(init)  =  empty .
```

We also specify the six guessed properties in the specification of M_{Qlock} as follows:

```
eq inv1(S, I, J)  =  ((pc(S, I)  =  cs and pc(S, J)  =  cs) implies I  =  J) .
eq inv2(S, I)  =  (pc(S, I)  =  cs implies top(queue(S))  =  I) .
eq inv3(S, I)  =  (pc(S, I)  =  rs implies (not I \in queue(S))) .
eq inv4(S, I)  =  ((not I \in queue(S)) implies pc(S, I)  =  rs) .
eq inv5(S, I)  =  (pc(S, I)  =  ws or pc(S, I)  =  cs implies I \in queue(S)) .
eq inv6(S, I)  =  (I \in queue(S) implies pc(S, I)  =  ws or pc(S, I)  =  cs) .
```

where S is a CafeOBJ variable of Sys and I & J are CafeOBJ variables of Pid.

Theorem Proving of Guessed Properties

We prove that the guessed and confirmed properties hold for Qlock. Precisely, we prove that the state predicates for all $\mathrm{I}, \mathrm{J} \in \mathrm{Pid}$ are invariant properties wrt M_{Qlock}. The main proof technique is structural induction on R_{Qlock}. Let us consider the proof of inv3(S, I). Applying the structural induction to inv3(S, I), four cases are generated: one base case for init and three induction cases for try, want and exit. Let us consider the induction case for exit. What to show in the induction case is inv3(exit(s, k), i), where s is a fresh constant of Sys representing an arbitrary state and k & i are fresh constants of Pid representing arbitrary process IDs. The induction case is first split into two cases based on the condition of exit: (1) $\mathrm{ps}(\mathrm{s}, \mathrm{k}) = \mathrm{cs}$ and (2) $\mathrm{ps}(\mathrm{s}, \mathrm{k}) \neq \mathrm{cs}$. For case (2), the following proof score fragment is written:

```
open QLOCK .
 op s : -> Sys . ops i k : -> Pid .
 eq [: nonexec] : inv3(s, I: Pid)  =  true .
 eq (pc(s, k)  =  cs)  =  false .
 red inv3(s, i) implies inv3(exit(s, k), i) .
close
```

The first equation is the induction hypothesis, where I is a CafeOBJ variable of Pid declared on-the-fly. The equation is annotated as : nonexec, meaning that the equation is not used as a left-to-right rewrite rule by CafeOBJ. Instead, inv3(s, i) is used as the premise of the implication. red stands for reduction and simplifies a given term by

using all equations except for those annotated as :nonexec as left-to-right rewrite rules. Feeding the proof score fragment into CafeOBJ, CafeOBJ returns true as the result, meaning that the case is discharged.

To discharge case (1), the case needs to be split into multiple cases. First case (1) is split into two cases: (1.1) i = k and (1.2) i ≠ k. Case (1.1) also needs to be split into two cases: (1.1.1) queue(s) = empty (meaning that queue(s) is empty) and (1.1.2) queue(s) = j q, where j is a fresh constant of Pid and q is a fresh constant of Queue (meaning that queue(s) is not empty). For case (1.1.1), the following proof score fragment is written:

```
open QLOCK .
 op s : -> Sys . ops i k : -> Pid .
 eq [:nonexec] : inv3(s,I:Pid) = true .
 eq pc(s,k) = cs . eq i = k . eq queue(s) = empty .
 red inv3(s,i) implies inv3(exit(s,k),i) .
close
```

Feeding this into CafeOBJ, CafeOBJ returns true.

Case (1.1.2) also needs to be split into two cases: (1.1.2.1) j = k and (1.1.2.2) j ≠ k. For case (1.1.2.1), the proof score fragment is as follows:

```
open QLOCK .
 op s : -> Sys . ops i k j : -> Pid . op q : -> Queue .
 eq [:nonexec] : inv3(s,I:Pid) = true .
 eq [:nonexec] : inv7(s,I:Pid) = true .
 eq pc(s,k) = cs . eq i = k . eq queue(s) = j q . eq j = k .
 red inv3(s,i) implies inv3(exit(s,k),i) .
close
```

Feeding this into CafeOBJ, CafeOBJ returns (true xor (k \in q)), where xor is the exclusive or operator. If k \in q is false, then the result becomes true. Observing the graphical animation of M_{Qlock}, there is at most one occurrence of each process ID i in queue. Therefore, we can guess the 7th property:

- Prop. 7: For each process ID i, let q be the queue obtained by deleting i from $queue$ and then it is always the case that there is no occurrence of i in q. If there is no occurrence of i in $queue$, then q is the same as $queue$.

This guessed property can be confirmed by Maude as follows:

```
search [1] in QLOCK : {init} =
            >* {(pc[I]: L1) (queue: Q) S}
such that not (not I \in del(Q,I)) .
```

where del takes a queue and a process ID and returns the one obtained by deleting the first occurrence of the ID from the queue. The search does not find any counterexamples. Prop. 7 is specified in CafeOBJ as follows:

```
eq inv7(S,I) = (not I \in del(queue(S),I)) .
```

inv7 is used in the proof score fragment for case (1.1.2.1) as follows:

```
open QLOCK .
 op s : -> Sys . ops i k j : -> Pid . op q : -> Queue .
 eq [:nonexec] : inv3(s,I:Pid) = true .
 eq [:nonexec] : inv7(s,I:Pid) = true .
 eq pc(s,k) = cs . eq i = k . eq queue(s) = j q . eq j = k .
 red inv7(s,k) and inv3(s,i) implies inv3(exit(s,k),i) .
close
```

Feeding this into CafeOBJ, CafeOBJ returns true.

For case (1.1.2.2), the proof fragment is as follows:

```
open QLOCK .
 op s : -> Sys . ops i k j : -> Pid . op q : -> Queue .
 eq [:nonexec] : inv2(s,I:Pid) = true .
 eq [:nonexec] : inv3(s,I:Pid) = true .
 eq pc(s,k) = cs . eq i = k . eq queue(s) = j q . eq (j = k) = false .
 red inv3(s,i) implies inv3(exit(s,k),i) .
close
```

Feeding this into CafeOBJ, CafeOBJ returns (true xor (k \in q)). For this case, we cannot use inv7 because we assume $j \neq k$. But, we also assume that j is the top of *queue*. Therefore, we can use inv2 as a lemma in the proof score fragment as follows:

```
open QLOCK .
 op s : -> Sys . ops i k j : -> Pid . op q : -> Queue .
 eq [:nonexec] : inv2(s,I:Pid) = true .
 eq [:nonexec] : inv3(s,I:Pid) = true .
 eq pc(s,k) = cs . eq i = k . eq queue(s) = j q . eq (j = k) = false .
 red inv2(s,k) and inv3(s,i) implies inv3(exit(s,k),i) .
close
```

Feeding this into CafeOBJ, CafeOBJ returns true.

Case (1.2) can be discharged likewise. It needs to use inv2 as a lemma but does not need to use inv7. The base case can be straightforwardly discharged, and the other two induction cases can be discharged only by case splitting. The other six including inv7 can be proved by structural induction on R_{Qlock}. The proof of inv1 uses inv2 as a lemma. The proof of inv2 uses as inv1 as a lemma. The proof of inv4 uses inv2 as a

lemma. The proof of inv5 uses inv2 as a lemma. The proof of inv6 uses inv2 and inv7 as lemmas. The proof of inv7 uses inv2 and inv3 as lemmas.

In addition to those lemmas on M_{Qlock}, to complete the formal proofs, we need to use the following four lemmas on queues:

```
eq X \in enq(Q, Y) = (if X = Y then true else X \in Q fi) .
ceq X \in del(enq(Q, X), X) = false if not X \in Q .
ceq X \in del(enq(Q, Y), X) = X \in del(Q, X) if not X = Y .
ceq X \in del(Q, X) = false if not X \in Q .
```

where X and Y are CafeOBJ variables of Pid and Q is a CafeOBJ variable of Queue. The first equation says that if X equals Y, then X \in enq(Q, Y) is true and otherwise it is the same as X \in Q. The second (conditional) equation says that if X \in Q is not true, then X \in del(enq(Q, X), X) is false. The third equation says that if X is different from Y, then X \in del(enq(Q, Y), X) is the same as X \in del(Q, X). The fourth equation says that if X \in Q is false, then X \in del(Q, X) is false.

CONCLUSION

We conjecture that graphical animations of state machines help human users guess or conjecture non-trivial properties of state machines, which could be used to complete formal proofs of theorems. Although model checking is convenient as well as useful to confirm guessed properties, it is not enough because state machine may not have a fixed small number of reachable states. If that is the case, interactive theorem proving is one possible technique to tackle the situation. The case study reported in the paper supports our claim to some extent. To support our claim more, we should conduct some more case studies as mentioned in the last section.

ACKNOWLEDGEMENT

We are grateful to the editor to handle our paper and to the anonymous reviewers to carefully read an earlier version of the paper and give us valuable comments to make it possible for us to properly revise it.

REFERENCES

Aung, M. T., Nguyen, T. T. T. & Ogata, K. (2018). Guessing properties of the Qlock mutual exclusion protocol based on its graphical animations and confirming the properties by model checking. 8th International Conference on Software and Computer Applications, 153-157.

Clavel, M., Duran, F., Eker, S., Lincoln, P., Marti-Oliet, N., Meseguer, J. & Talcott, C. (2007). *All About Maude*. Berlin: Springer-Verlag.

Diaconescu, R. & Kokichi, F. (1997). *CafeOBJ Report*. Singapore: World Scientific.

Goguen, J. A. (1990). Proving and rewriting. 2nd International Conference on Algebraic and Logic Programming, 1–24.

Goguen, J. A., Winkler, T., Meseguer, J., Futatsugi, K., Jouannaud, J. P. (2000). Introducing OBJ. Software Engineering with OBJ. Goguen & Malcolm (Ed.). New York: Springer US.

Hallerstede, S., Leuschel, M., & Plagge, D. (2013). Validation of formal models by refinement animation. Science of Computer Programming, 78, 272–292, 2013. doi:10.1016/j.scico.2011.03.005

Jackson, D. (2012). *Software Abstraction (Revised edition)*. Cambridge:The MIT Press.

Li, M. & Liu, S. (2016). Integrating animation-based inspection into formal design specification construction for reliable software systems. IEEE Transactions on Reliability, 65, 88–106. doi:10.1109/TR.2015.2456853

Liang, H., Dong, J. S., Sun, J. & Wong, W. E. (2016). Software monitoring through formal specification animation. Innovations in Systems and Software Engineering, 5, 231–241. doi:10.1007/s11334-015-0269-z

Mellor-Crummey, J. M. & Scott, M. L. (1991). Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9, 21-65. doi: 10.1145/103727.103729

Nagoya, F. & Liu, S.: An Investigation of Integrating a GUI-Aided Approach and a Specification-Based Testing, 8th International Workshop on SOFL+MSVL, 24-35

Nguyen, T. T. T. & Ogata, K. (2017a). Graphical Animations of State Machines. 15th IEEE International Conference on Dependable, Autonomous and Secure Computing, 604-611.

Nguyen, T. T. T. & Ogata, K. (2017b). Graphically perceiving characteristics of the MCS lock and model checking them, 8th International Workshop on SOFL+MSVL, 3-23.

Ogata, K. & Futatsugi, K. (2003). Proof Scores in the OTS/CafeOBJ Method. 6th International Conference on Formal Methods for Open Object-Based Distributed Systems, 170-184.

Ogata, K. & Futatsugi, K. (2008). Proof Score Approach to Verification of Liveness Properties. IEICE TRANSACTIONS on Information and Systems, E91-D, 2804-2817, 170-184. doi:10.1093/ietisy/e91-d.12.2804

Ogata, K. & Futatsugi, K. (2013). Compositionally Writing Proof Scores of Invariants in the OTS/CafeOBJ Method. The Journal of Universal Computer Science, 19, 771-804. doi: 10.3217/jucs-019-06-0771

Preining, N., Ogata, K. & Futatsugi, K. (2014). Liveness Properties in CafeOBJ - A Case Study for Meta-Level Specifications. 24th International Symposium on Logic-Based Program Synthesis and Transformation, 182-198.

Santiago, S., Talcott, C. L., Escobar, S., Meadows, C. A. & Meseguer, J. (2009). A Graphical User Interface for Maude-NPA. 9th Spanish Conference on Programming and Languages, 3-20.

Sun, J., Liu, Y., Dong, J. S., & Pang, J. (2009). PAT: Towards flexible verification under fairness. 21st International Conference on Computer Aided Verification, 709–714.

Yoshida, H., Ogata, K. & Futatsugi, K. (2015). Formalization and Verification of Declarative Cloud Orchestration. 24th International Symposium on Logic-Based Program Synthesis and Transformation, 33-49.

APPENDIX

A glossary of symbols and terminologies used in the paper is given:

e \in q: A membership predicate for queues, where e is an element and q is a queue; The queue that consists of the elements e_1, e_2 & e_3 in this order is expressed as e_1 e_2 e_3 empty, where empty is the empty queue

p and q, p implies q & not p: p ∧ q, p ⇒ q & ¬p, respectively.

CafeOBJ: An algebraic specification language and tool that is used to specify state machines in equations as OTSs and to formally verify that state machines enjoy invariant properties by theorem proving.

constr: It is used to declare that operators are data constructors.

Command close: It indicates the end of use of a module started with open.

Equations for state transitions: They are in the form "eq $[l]$: $obs(act(S,X),Y) = val(S,X,Y)$.", where l is the label given to the equation, obs is an observer and act is an action; It specifies how the value observed by *obs* together with a parameter X changes if act is applied (or taken) in a state S together Y; Equations can have conditions; If so, ceq is used instead of eq, "if" is written after the right-hand side and conditions are written between "if" and the full stop; Equations are used as left-to-right rewrite rules to reduce terms unless : nonexec is given.

Invariant properties wrt M: State predicates p that hold in all reachable states wrt M, namely $(\forall s \in R) p(s)$.

$M \triangleq \langle S, I, T \rangle$: A state machine; S is a set of states; $I \subseteq S$ is the set of initial states; $T \subseteq S \times S$ is a binary relation over S; elements $(s, s') \in T$ are called state transitions; (s, s') may be written as $s \to s'$; M_{Mod3} is the state machine formalizing Mod2; M_{Qlock} is the state machine formalizing Qlock.

Maude: A rewriting-logic based programming and specification language and tool that is used to specify state machines in terms of rewrite rules and model check that state machines enjoy linear temporal logic (LTL) properties as well as invariant properties.

Mod3: A system that only has one value val whose initial value is 0 and that changes to 1, 2 and 0 repeatedly in this order.

Observable components: Name-value pairs, such as (pc[p1]: cs), where pc[p1]: is the name and cs is the value.

Observational Transition System (OTS): State machines described such that each value that characterizes states is observed by applying a function called an observer to the states (together with some parameters if any), a set of transitions is represented by a function called an action (or a transition), and how to change each value that characterizes states by applying an action to the states is specified in terms of equations.

Command open: A CafeOBJ command that takes a module and make it possible to use the module

Proof scores: Programs written in an algebraic specification language, such as CafeOBJ, to conduct theorem proving.

Qlock: A mutual exclusion protocol; an abstract version of the Dijkstra Binary Semaphore.

R: The set of reachable states wrt M; R_{Mod3} is the set of reachable states wrt M_{Mod3}; R_{Qlock} is the set of reachable states wrt M_{Qlock}.

Command red : A CafeOBJ (and Maude) command that takes a term and reduces (or computes) it by using equations as left-to-right rewrite rules.

Rewrite rules for state transitions: They are in the form rl $[l]$: $\{StatePattern_1\}$ => $\{StatePattern_2\}$. that says that $StatePattern_1$ changes to $StatePattern_2$,

where l is the label given to the rule; Rewrite rules can have conditions; If so, crl is used instead of rl, "if" is written after the right-hand side and conditions are written between "if" and the full stop.

Search command: A Maude (and CafeOBJ) command that is in the form "search $[n]$ in $Module : s =>* p$ such that c ." that searches all reachable states from s for at most n states that can match p and make c true.

Soups: Collections that satisfy associative and commutative laws; a soup that consist of e_1 , e_2 and e_3 is expressed as $e_1\ e_2\ e_3$; because it satisfies associative and commutative laws, $e_2\ e_1\ e_3$, $e_3\ e_2\ e_1$, etc. are exactly the same as $e_1 e_2 e_3$;

SMGA: A state machine graphical animation tool.