Lemma Weakening for State Machine Invariant Proofs

Duong Dinh Tran*, Dang Duy Bui*, Parth Gupta†, and Kazuhiro Ogata*
*School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
Email: {duongtd, bddang, ogata}@jaist.ac.jp
†Indian Institute of Technology Kharagpur
Kharagpur, West Bengal 721302, India
Email: parthgupta.iitkgp@gmail.com

***Abstract*—Lemma conjecture is one of the most challenging tasks in theorem proving. The paper focuses on invariant properties (or invariants) of state machines. Thus, lemmas are also invariants. To prove that a state predicate p is an invariant of a state machine M, in general, we need to find an inductive invariant q of M such that $q(s)$ implies $p(s)$ for all states s of M. q is often in the form $p \wedge p'$, and p' is often in the form $q_1 \wedge \ldots \wedge q_n$. q_1, …, q_n are the lemmas of the proof that p is an invariant of M. The paper proposes a technique called Lemma Weakening (LW). LW replaces q_i with q'_i such that $q_i(s)$ implies $q'_i(s)$ for all states s of M, which can make the proof reasonably tractable that may become otherwise unreasonably hard. MCS mutual exclusion protocol is used as an example to demonstrate the power of LW.**

***Index Terms*—lemma weakening, lemma conjecture, MCS protocol, proof score, algebraic specification language, lemma strengthening**

I. Introduction

State machines can be used to formalize various kinds of systems as mathematical models and systems requirements can be expressed as state machine properties. Therefore, it is possible to check if systems satisfy their requirements by formally verifying that state machines enjoy properties. There are two major formal verification techniques: model checking and theorem proving. The former can be automatically conducted but basically cannot be used for systems that have an infinite number of states (infinite-state systems). The latter can directly deal with infinite-state systems but requires human interaction, especially for lemma conjecture that is one of the most challenging tasks in theorem proving. The paper focuses on invariant properties (or invariants) of state machines. Thus, lemmas are also invariants.

To prove that a state predicate p is an invariant of a state machine M, in general, we need to find an inductive invariant q of M such that $q(s)$ implies $p(s)$ for all states s of M [1]. q is often in the form $p \wedge p'$, and p' is often in the form $q_1 \wedge \ldots \wedge q_n$. $q_1, \ldots, q_n$ are the lemmas of the proof that p is an invariant of M. q (or $p \wedge p'$) is more generic or stronger than p, and p' (or $q_1 \wedge \ldots \wedge q_n$) is more generic or stronger than each q_i for $i = 1, \ldots, n$. Accordingly, invariant proofs can be regarded as strengthening lemmas, which is called Lemma Strengthening (LS) in this paper. For example, starting with q_1, we need to have $q_1 \wedge \ldots \wedge q_n$ that is stronger than q_1, $q_1 \wedge q_2$, etc. We also need to use LS for proofs of data properties. For instance, when we prove $\mathrm{rev}(\mathrm{rev}(L)) = L$ for all lists L by structural induction on L, where rev is the reverse function of lists, we encounter the situation at the induction case such that we need to use a lemma. The most straightforward lemma would be $\mathrm{rev}(\mathrm{rev}(L) \text{ @ } (E \mid \mathrm{nil})) = E \mid L$ for all lists L and all elements E, where @ is the concatenation function of lists, $\mid$ is the constructor of lists and nil is the empty list. The proof of the lemma requires us to use another lemma $\mathrm{rev}(\mathrm{rev}(L) \text{ @ } (E_1 \mid E_2 \mid \mathrm{nil})) = E_2 \mid E_1 \mid L$ for all lists L and all elements E_1, E_2. If we only use the most straightforward lemmas, we cannot make the proof converge. To make the proof converge, we need to strengthen such lemmas. One possible lemma obtained by strengthening such lemmas is $\mathrm{rev}(\mathrm{rev}(L_1) \text{ @ } L_2) = \mathrm{rev}(L_2) \text{ @ } L_1$ for all lists L_1, L_2 that is stronger than all of the most straightforward lemmas. $\mathrm{rev}(\mathrm{rev}(L)) = L$ corresponds to p, while $\mathrm{rev}(\mathrm{rev}(L_1) \text{ @ } L_2) = \mathrm{rev}(L_2) \text{ @ } L_1$ corresponds to p' or $q_1 \wedge \ldots \wedge q_n$.

While proving that MCS mutual exclusion protocol (MCS protocol or simply MCS) [2] enjoys the mutual exclusion property, we have encountered a situation where the use of only LS did not seem to make the proof converge. We got over the situation by weakening some lemmas. The paper proposes a technique called Lemma Weakening (LW). LW replaces q_i with q'_i such that $q_i(s)$ implies $q'_i(s)$ for all states s of M, which can make the proof reasonably tractable that may become otherwise unreasonably hard. MCS is used as an example to demonstrate the power of LW.

In this paper, we use observational transition systems (OTSs) [3] as state machines and CafeOBJ [4] as a formal specification language and system. Formal proofs are conducted by writing what are called proof scores [3] in CafeOBJ and executing them with CafeOBJ. All specifications and proof scores in CafeOBJ presented in this paper are available at http://gitlab.com/duongtd23/mcs/.

The work was partially supported by JSPS KAKENHI Grant Number JP19H04082.

The rest of the paper is organized as follows. Sect. II gives some preliminaries, such as OTSs and CafeOBJ. Sect. III describes LS and LW. Sect. IV describes MCS and its formal specification in CafeOBJ. Sect. V reports on the case study in which MCS is used to demonstrate the power of LW. Sect. VI mentions some related work. Sect. VII concludes the paper with some pieces of our future work.

II. Preliminaries

This section presents some basic notions which are requirements for the rest of the paper. We first give the definition of OTSs. Then, through a simple example, we show how to specify the OTS in CafeOBJ as well as how to write proof scores to conduct formal verification.

A. Observational Transition Systems (OTSs)

We suppose that there exists a universal state space denoted Υ and that each data type used in OTSs is provided. The data types include Bool for Boolean values. A data type is denoted D with a subscript such as D_{o1} and D_o.

Definition 1: An OTS $\mathcal{S}$ is $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that

- $\mathcal{O}$: A finite set of observers. Each *observer* $o : \Upsilon\, D_{o1} \ldots D_{om} \to D_o$ is a function that takes one state and m (≥ 0) data values and returns one data value. The equivalence relation ($\upsilon_1 =_{\mathcal{S}} \upsilon_2$) between two states $\upsilon_1, \upsilon_2 \in \Upsilon$ is defined as $(\forall o \in \mathcal{O})(\forall x_1 \in D_{o1}) \ldots (\forall x_m \in D_{om})\, (o(\upsilon_1, x_1, \ldots, x_m) = o(\upsilon_2, x_1, \ldots, x_m))$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A finite set of transitions. Each *transition* $t : \Upsilon\, D_{t1} \ldots D_{tn} \to \Upsilon$ is a function that takes one state and n (≥ 0) data values and returns one state, provided that $t(\upsilon_1, y_1, \ldots, y_n) =_{\mathcal{S}} t(\upsilon_2, y_1, \ldots, y_n)$ for each $[\upsilon] \in \Upsilon / =_{\mathcal{S}}$, each $\upsilon_1, \upsilon_2 \in [\upsilon]$ and each $y_k \in D_{tk}$ for $k = 1, \ldots, n$. Each transition t has the condition $c\text{-}t : \Upsilon\, D_{t1} \ldots D_{tn} \to \text{Bool}$, which is called *the effective condition* of t. If $c\text{-}t(\upsilon, y_1, \ldots, y_n)$ does not hold, then $t(\upsilon, y_1, \ldots, y_n) =_{\mathcal{S}} \upsilon$.

A pair (υ, υ') of states is called a *transition instance* if there exists $t \in \mathcal{T}$ such that $\upsilon' =_{\mathcal{S}} t(\upsilon, y_1, \ldots, y_n)$ for some $y_i \in D_{ti}$ for $i = 1, \ldots, n$. Such a pair (υ, υ') may be denoted $\upsilon \to_{\mathcal{S}} \upsilon'$ (or $\upsilon \to \upsilon'$) to emphasize that υ directly goes to υ' by one step.

Each state that is reachable from an initial state through transitions is called a reachable state.

Definition 2: Given an OTS $\mathcal{S}$, *reachable states* with respect to (wrt) $\mathcal{S}$ are inductively defined:

- Each $\upsilon \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.
- For each $t \in \mathcal{T}$ and each $y_k \in D_{tk}$ for $k = 1, \ldots, n$, $t(\upsilon, y_1, \ldots, y_n)$ is reachable wrt $\mathcal{S}$ if $\upsilon \in \Upsilon$ is reachable wrt $\mathcal{S}$.

Let $\mathcal{R}_{\mathcal{S}}$ be the set of all reachable states wrt $\mathcal{S}$.

Predicates whose types are $\Upsilon \to \text{Bool}$ are called *state predicates*. State predicates may have universally quantified variables. State predicates that hold in all reachable states wrt $\mathcal{S}$ are invariants wrt $\mathcal{S}$.

Definition 3: A state predicate $\rho : \Upsilon \to \text{Bool}$ is called an *invariant* wrt $\mathcal{S}$ if $\rho(\upsilon)$ is true for all $\upsilon \in \mathcal{R}_{\mathcal{S}}$, i.e. $(\forall \upsilon \in \mathcal{R}_{\mathcal{S}})\, \rho(\upsilon)$.

State predicates that are preserved by all transitions are inductive invariants.

Definition 4: A state predicate $\rho : \Upsilon \to \text{Bool}$ is called an *inductive invariant* wrt $\mathcal{S}$ if it satisfies the following two conditions:

1) $(\forall \upsilon \in \mathcal{I})\, \rho(\upsilon)$
2) $(\forall t \in \mathcal{T})(\forall \upsilon \in \Upsilon)(\forall y_1 \in D_{t1}) \ldots (\forall y_n \in D_{tn})\, (\rho(\upsilon) \Rightarrow \rho(t(\upsilon, y_1, \ldots, y_n)))$

Inductive invariants wrt $\mathcal{S}$ are invariants wrt $\mathcal{S}$ but not vice versa.

B. CafeOBJ and Proof Scores

A mutual exclusion protocol is used as an example to describe how to specify OTSs in CafeOBJ. The mutual exclusion protocol written in Algol-like pseudo-code is as follows:

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

test&set(*locked*) atomically does the following: if *locked* is false, then it sets *locked* to true and returns false; otherwise it just returns true. Since the protocol uses test&set, it is called TAS protocol or simply TAS. Each process is located at either rs (Remainder Section) or cs (Critical Section) and initially at rs. *locked* is a Boolean variable shared by all processes and initially false.

To formalize TAS as an OTS $\mathcal{S}_{\text{TAS}}$, we use two observers with which we observe the location of each process and the value stored in *locked*. The two observers are expressed as the CafeOBJ operators declared as follows:

```
op pc : Sys Pid -> Label .
op locked : Sys -> Bool .
```

where `Sys` is the sort (or type) representing Υ, `Pid` is the sort of process IDs and `Label` is the sort of locations such as rs and cs. For `s` of `Sys` and `p` of `Pid`, `pc(s,p)` is the location at which `p` is located in state `s` and `locked(s)` is the value stored in *locked* in state `s`. Observers and CafeOBJ operators that express observers are interchangeably used in this paper.

We use two transitions that are expressed as CafeOBJ operators. An arbitrary initial state of $\mathcal{S}_{\text{TAS}}$ is expressed as a CafeOBJ operator. Those CafeOBJ operators are declared as follows:

```
op init : -> Sys {constr} .
op enter : Sys Pid -> Sys {constr} .
op exit : Sys Pid -> Sys {constr} .
```

where `constr` stands for constructors. The three operators `init`, `enter` and `exit`, together with process IDs, construct $\mathcal{R}_{\mathcal{S}_{\text{TAS}}}$. `init` and `enter` are defined in terms of equations as follows:

```
eq pc(init,P) = rs .
eq locked(init) = false .
ceq pc(enter(S,P),Q) = (if P = Q then cs
  else pc(S,Q) fi) if c-enter(S,P) .
ceq locked(enter(S,P)) = true if c-enter(S,P) .
ceq enter(S,P) = S if not c-enter(S,P) .
```

where `S` is a CafeOBJ variable of `Sys` and `P` & `Q` are CafeOBJ variables of `Pid`. `c-enter(S,P)` is `pc(S,P) = rs and not locked(S)`. `if` c `then` a `else` b `fi` is a if c equals `true` and b if c equals `false`. Note that in CafeOBJ, all variables are universal quantifier. Transitions and CafeOBJ operators that express transitions are interchangeably used in this paper.

One desired property TAS should enjoy is the mutual exclusion property whose informal description is that there is always at most one process in Critical Section. Let `mutex(S,P,Q)` be `pc(S,P) = cs and pc(S,Q) = cs implies P = Q`. The property is formalized as the following invariant: $(\forall v \in \mathcal{R}_{\mathcal{S}_{\mathrm{TAS}}})$ $(\forall p, q \in \texttt{Pid})$ $\texttt{mutex}(v,p,q)$. Let us use "the proof of `mutex`" (and "to prove `mutex`") to mean the proof of $(\forall v \in \mathcal{R}_{\mathcal{S}_{\mathrm{TAS}}})(\forall p, q \in \texttt{Pid})$ $\texttt{mutex}(v,p,q)$ (and to prove $(\forall v \in \mathcal{R}_{\mathcal{S}_{\mathrm{TAS}}})(\forall p, q \in \texttt{Pid})$ $\texttt{mutex}(v,p,q)$) not only for `mutex` but also for any other similar operators `inv`i that takes one state and zero or more data values and returns a Boolean value. The invariant is proved by (simultaneous) structural induction on v (or `S`) by writing proof scores in CafeOBJ. The proof score fragment for the base case `init` is as follows:

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

where `open` makes the given module (or specification) available, `close` stops the use of the module and `red` reduces (computes) the given term. `p` & `q` are fresh constants of `Pid` representing arbitrary process IDs. CafeOBJ returns `true` for the proof score fragment meaning that the case is discharged.

In the induction cases, we want to prove that if `mutex` holds in a state v, it also holds in the successor states v' and v'' of v, where v' and v'' are made by transitions when an arbitrary process r tries to move to cs from rs, and moves to rs from cs, respectively. Accordingly, two induction cases need to be proved. Let us consider the induction case in which `enter` is taken into account. In this paper, we use the induction case t to mean the induction case in which a transition t is taken into account. The case is first split into two sub-cases: (1) `pc(s,r) = rs` and (2) `(pc(s,r) = rs) = false`. For the sub-case (2), the following proof score fragment is written:

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

where `s` is a fresh constant of `Sys` representing an arbitrary state. The equation characterizes the sub-case. `mutex(s,p,q)` is an instance of the induction hypothesis. Feeding the proof score fragment into CafeOBJ, CafeOBJ returns `true`, indicating that the sub-case is discharged. For the sub-case (1), it is necessary to conduct case splitting several more times. Let us consider a sub-case of (1), which has the following proof score fragment:

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

CafeOBJ returns `false` for this fragment. We need to conjecture a lemma to discharge the sub-case. The lemma is as follows:

```
eq inv1(S,P) =
  (pc(S,P) = cs implies locked(S)) .
```

Then, in the above open-close fragment, `inv1` is used as a lemma to discharge the sub-case of (1) as follows:

```
red inv1(s,q) implies mutex(s,p,q)
  implies mutex(enter(s,r),p,q) .
```

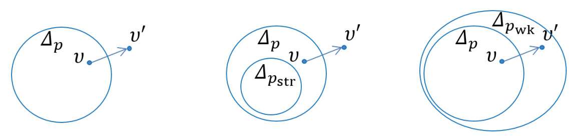
CafeOBJ now returns `true` for the proof score fragment.

Since `inv1` is used in the proof of `mutex`, we need to prove that `inv1(S,P)` for all process IDs `P` is also an invariant wrt $\mathcal{S}_{\mathrm{TAS}}$ to complete the verification. The proof of `inv1` needs to use `mutex` as a lemma. Although the proof of `mutex` uses `inv1` and vice versa, our proof is not circular. The reason is that we use simultaneous (structural) induction to develop our proof. The correctness of this method has been formally proved in the paper [3]. The complete specification of TAS as well as the proof scores can be found at the webpage presented in Sect. I.

III. Lemma Strengthening (LS) and Lemma Weakening (LW)

Let us suppose that we want to prove that a state predicate p is an invariant wrt an OTS $\mathcal{S}$. It is often the case that p is not inductive and then there is a transition instance that does not preserve p such that $p(v)$ holds but $p(v')$ does not, where $v \to v'$ is a transition instance, which is shown in Fig. 1 (a), where Δ_p is $\{v \in \Upsilon \mid p(v)\}$. This is the reason why invariant proofs become non-trivial or even can become very hard. If we can successfully show that the source v is not reachable wrt $\mathcal{S}$, then we do not need to consider the transition instance, being able to discharge the case. One possible way to do so is to find p_{str} that is stronger than p such that $p_{\mathrm{str}}(v)$ does not hold and to prove that p_{str} is an invariant wrt $\mathcal{S}$, which is shown in Fig. 1 (b). If p_{str} is inductive wrt $\mathcal{S}$, we do not need to use any more lemmas. This approach has been summarized as the proof rule Inv by Manna and Pnueli [1].

To prove that p is an invariant of $\mathcal{S}$ (or a state machine), in general, we need to find an inductive invariant q wrt $\mathcal{S}$ such that $q(v) \Rightarrow p(v)$ for all states $v \in \Upsilon$. In practice, q is often in the form $p \wedge p'$ and p' is often in the form $q_1 \wedge \ldots \wedge q_n$. q_1, $\ldots$, q_n are the lemmas of the proof that p is an invariant wrt



(a) The source making the proof non-trivial (b) $p_{\rm str}$ obtained by strengthening p (c) $p_{\rm wk}$ obtained by weakening p

Fig. 1. The reason why invariant proofs become non-trivial and two approaches to tackling the non-trivial situation

$\mathcal{S}$. It is often the case that we do not know any of $q_1, \ldots, q_n$ in advance. We need to gradually conjecture $q_1, \ldots, q_n$ one by one when we encounter the situation shown in Fig. 1 (a). For example, while proving that p is an invariant wrt $\mathcal{S}$, we may conjecture q_1. $q_1(v)$ for the source v needs not to hold but does not need to be properly stronger than p because $p \wedge q_1 \wedge \ldots \wedge q_n$ is surely stronger than p. In general, when we have conjectured up to q_k, where $k = 1, \ldots, n$, we do not know how many more lemmas we need to conjecture. We may move toward the direction such that our proof attempt never converges as we only use the most straightforward lemmas for the proof of $\mathrm{rev}(\mathrm{rev}(L)) = L$ for all lists L.

The reason why invariant proofs become non-trivial or even can become very hard is because there exists a transition instance $v \to v'$ as shown in Fig. 1 (a). The proof rule Inv gets rid of such a transition instance as shown in Fig. 1 (b). Another possible way to get rid of such a transition instance is to find $p_{\rm wk}$ that is weaker than p such that $p_{\rm wk}(v')$ holds and to prove that $p_{\rm wk}$ is an invariant wrt $\mathcal{S}$, which is shown in Fig. 1 (c). Even though $p_{\rm wk}$ is an invariant wrt $\mathcal{S}$, however, it does not guarantee that p is an invariant wrt $\mathcal{S}$. This is because Δ_p may not contain all reachable states in $\mathcal{R}_{\mathcal{S}}$. Therefore, the second approach is not used to prove that p is an invariant wrt $\mathcal{S}$. This might be the reason why the second approach has been rarely used. Although the second approach is not very useful for p, it may be useful for some q_i, a lemma of the proof that p is an invariant wrt $\mathcal{S}$. In this paper, strengthening lemmas q_i is called Lemma Strengthening (LS), while weakening lemmas q_i is called Lemma Weakening (LW). We have already shown a concrete case that demonstrates the usefulness of LS for the proof of $\mathrm{rev}(\mathrm{rev}(L)) = L$ for all lists L.

While proving that MCS enjoys the mutual exclusion property, we realized that LW can make the proof attempt converge that otherwise did not seem to converge in a reasonable amount of time. Note that we have used LS as well as LW for the proof that MCS enjoys the mutual exclusion property. We will describe in which way LW makes the proof attempt converge in Sect. V.

IV. MCS Protocol and its Formal Specification in CafeOBJ

Before going to present the usefulness of LW in making the proof that MCS protocol enjoys the mutual exclusion property attempt converge, we first need the specification of the protocol. This section describes MCS and the formal specification of $\mathcal{S}_{\rm MCS}$ that formalizes the protocol in CafeOBJ.

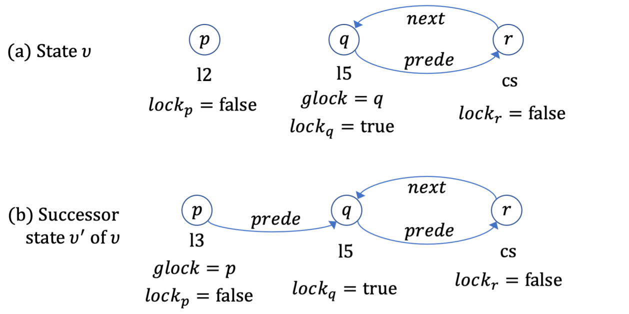


Fig. 2. The change of state of MCS when a process p moves to l3 from l2

A. MCS Protocol

MCS is a mutual exclusion protocol invented by Mellor-Crummey and Scott [2]. Variants of MCS have been used in Java VMs and therefore the 2006 Edsger W. Dijkstra Prize in Distributed Computing went to their paper [2]. The algorithm inside MCS protocol is a scalable algorithm for spin locks that generates $O(1)$ remote references per lock acquisition, independent of the number of processes attempting to acquire the lock.

MCS written in Algol-like pseudo-code is as follows:

```
rs  : "Remainder Section"
l1  : $next_p$ := nop;
l2  : $prede_p$ := fetch&store($glock, p$);
l3  : **if** $prede_p \neq$ nop {
l4  :     $lock_p$ := true;
l5  :     $next_{prede_p}$ := $p$;
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 : $lock_{next_p}$ := false;
l12 : **goto** rs;
```

MCS uses one global variable $glock$ and three local variables $next_p$, $prede_p$ and $lock_p$ for each process p. Process IDs are stored in $glock$, $next_p$ and $prede_p$, while a Boolean value is stored in $lock_p$. There is one special (dummy) process ID nop that is different from any real process IDs. Initially, each of $glock$, $next_p$ and $prede_p$ is set to nop and $lock_p$ is set to false. We suppose that each process is located at one of the labels, such as rs, l1 and cs. Initially, each process is located at rs. When a process wants to enter "Critical Section," it first moves to l1 from rs.

MCS uses two non-trivial atomic instructions: fetch&store and comp&swap. For a variable x and a value a, fetch&store(x, a) atomically does the following: x is set to a and the old value of x is returned. For a variable x and values a, b, comp&swap(x, a, b) atomically does the following: if x equals a, then x is set to b and true is returned; otherwise false is just returned.

Fig. 2 graphically visualizes the change of state of MCS when a process p moves to l3 from l2. In the state υ, which is represented by Fig. 2 (a), processes p, q, and r, located at l2, l5, and cs, respectively; $glock$ is q; $next$ of r is q; and $prede$ of q is r. When process p moves to $l3$, $glock$ is set to itself, and its $prede$ is set to q (Fig. 2 (b)).

B. Formal Specification of MCS Protocol

Each state of $\mathcal{S}_{\mathrm{MCS}}$ is characterized by $glock$, $next_p$, $prede_p$, $lock_p$ and p's location for each process p. Then, we use the following observers:

```
op glock : Sys -> Pid&Nop .
op pc : Sys Pid -> Label .
op next  : Sys Pid -> Pid&Nop .
op prede : Sys Pid -> Pid&Nop .
op lock  : Sys Pid -> Bool .
```

`Pid` is the sort for real process IDs, while `Pid&Nop` is the sort for real process IDs plus nop. The other sorts can be understood as those used in Sect. II.

An arbitrary initial state of $\mathcal{S}_{\mathrm{MCS}}$ is expressed as the following operator:

```
op init : -> Sys {constr} .
```

`init` is defined as follows:

```
eq glock(init) = nop .
eq pc(init,P) = rs .
eq next(init,P) = nop .
eq prede(init,P) = nop .
eq lock(init,P) = false .
```

where `P` is a CafeOBJ variable of `Pid`.

We use the following 14 transitions:

```
-- moves to l1 from rs
op want   : Sys Pid -> Sys {constr} .
-- moves to l2 from l1
op stnxt  : Sys Pid -> Sys {constr} .
-- moves to l3 from l2
op stprd  : Sys Pid -> Sys {constr} .
-- moves to l4 or cs from l3
op chprd  : Sys Pid -> Sys {constr} .
-- moves to l5 from l4
op stlck  : Sys Pid -> Sys {constr} .
-- moves to l6 from l5
op stnpr  : Sys Pid -> Sys {constr} .
-- tries to move to cs from l6
op chlck  : Sys Pid -> Sys {constr} .
-- moves to l7 from cs
op exit   : Sys Pid -> Sys {constr} .
-- moves to l8 or l11 from l7
op chnxt  : Sys Pid -> Sys {constr} .
-- moves to l9 or l10 from l8
op chglk  : Sys Pid -> Sys {constr} .
-- moves to rs from l9
op go2rs  : Sys Pid -> Sys {constr} .
-- tries to move to l11 from l10
op chnxt2 : Sys Pid -> Sys {constr} .
-- moves to l12 from l11
op stlnx  : Sys Pid -> Sys {constr} .
-- moves to rs from l12
op go2rs2 : Sys Pid -> Sys {constr} .
```

A comment starts with `--` and stops by the line end. The comments briefly explain what part of MCS is formalized by each transition. For example, if a process p is at l2 in a state s, then `stprd(`s,p`)` denotes the state just after p has executed the statement at l2 and moved to l3 from l2; if p is at l6 in s, then `chlck(`s,p`)` denotes the state just after p has exited the loop at l6 and moved to cs if $lock_p$ is false and the state just after p has done one iteration of the loop at l6 if $lock_p$ is true.

The transitions are defined in terms of equations that specify how the values observed by the five observers change. For example, `stprd` is defined as follows:

```
ceq glock(stprd(S,P)) = P if pc(S,P) = l2 .
ceq pc(stprd(S,P),Q) = (if P = Q then l3 else
  pc(S,Q) fi) if pc(S,P) = l2 .
eq next(stprd(S,P),Q) = next(S,Q) .
eq lock(stprd(S,P),Q) = lock(S,Q) .
ceq prede(stprd(S,P),Q)
  = (if P = Q then glock(S) else prede(S,Q) fi)
  if pc(S,P) = l2 .
ceq stprd(S,P) = S if (pc(S,P) = l2) = false .
```

where `S` is a CafeOBJ variable of `Sys` and `P` & `Q` are CafeOBJ variables of `Pid`. The other transitions can be defined likewise.

V. Formal Verification by Proof Scores

This section describes the proof score approach to formally verify that MCS enjoys the mutual exclusion property. The complete proof scores are available at the webpage presented in Sect. I. This section focuses on how to use LW as well as LS in the formal verification.

A. Use of Lemma Strengthening (LS)

We prepare the following module:

```
mod INV {
pr(MCS) .  var S : Sys .  vars P Q : Pid
op mutex : Sys Pid Pid -> Bool
eq mutex(S,P,Q) = ((pc(S,P) = cs and
  pc(S,Q) = cs) implies (P = Q)) .
}
```

where `pr(MCS)` says that another module `MCS` in which MCS is specified is used, `S` is a CafeOBJ variable of `Sys` and `P` & `Q` are CafeOBJ variables of `Pid`. In `INV`, we write state predicates we would like to prove as invariants wrt $\mathcal{S}_{\mathrm{MCS}}$. Initially, we only have `mutex` that is used to formalize the mutual exclusion property. While conducting the formal verification, we gradually conjecture lemmas and add them to `INV` on the fly.

Let us consider a sub-case of the induction case `chprd` in the proof of `mutex`. The proof score fragment of the sub-case is as follows:

```
open INV .
op s : -> Sys .  ops p r : -> Pid .
eq pc(s,r) = l3 .  eq p = r .
eq (q = r) = false .
eq prede(s,r) = nop .  eq pc(s,q) = cs .
red mutex(s,p,q)
  implies mutex(chprd(s,r),p,q) .
close
```

CafeOBJ returns `false` for the proof score fragment. The pair of states `s` and `chprd(s,r)` is a transition instance as shown in Fig. 1 (a). We need to conjecture and use a lemma to discharge the sub-case.

One possible lemma conjectured most straightforwardly can be constructed by combining the five equations that characterize the sub-case with conjunction, negating the whole formula and replacing the fresh constants `s`, `p`, `q` & `r` with variables `S`, `P`, `Q` & `R` [5]. Since the formula constructed is in the form `(not P = R) or` F that is equivalent to `(P = R) implies` F. Thus, the lemma constructed is F in which R is replaced with P, which is equivalent to the following:

```
eq inv1'(S,P,Q) = ((prede(S,P) = nop and
  pc(S,P) = l3 and (Q = P) = false)
  implies (pc(S,Q) = cs) = false) .
```

Since `inv1'(s,p,q)` reduces to `false`, in the above open-close fragment, we can use `inv1'` as a lemma to discharge the sub-case as follows:

```
red inv1'(s,p,q) implies mutex(s,p,q)
  implies mutex(chprd(s,r),p,q) .
```

In the proof of `inv1'`, however, we encounter a sub-case in which the proof reduces to `false`. The proof score fragment of the sub-case is as follows:

```
open INV .
op s : -> Sys .  ops p r : -> Pid .
eq pc(s,r) = l6 .  eq p = r .
eq (q = r) = false .
eq lock(s,r) = false .  eq pc(s,q) = l3 .
eq prede(s,q) = nop .
red inv1'(s,p,q)
  implies inv1'(chlck(s,r),p,q) .
close
```

By applying the same technique that has been explained above to conjecture `inv1'`, we can conjecture another lemma to discharge the sub-case in the proof of `inv1'` as follows:

```
eq inv1''(S,P,Q) = ((prede(S,Q) = nop and
  pc(S,Q) = l3 and (P = Q) = false) implies
(pc(S,P) = l6 and lock(S,P) = false) = false) .
```

The conditional part of `inv1''` is exactly the same as that of `inv1'`. The reason why we use the forms of `inv1'` and `inv1''` is because we emphasize what are shared by `inv1'` and `inv1''`. The proof of `inv1''` needs yet another lemma whose conditional part is exactly the same as that of `inv1'`. Although we do not need an infinite series of similar but different lemmas as is the case when we only use the most straightforward lemmas for the proof of $\mathrm{rev}(\mathrm{rev}(L)) = L$ for all lists L, we need several such ones. Therefore, we strengthen them to obtain the following lemma:

```
eq inv1(S,P,Q) = ((prede(S,P) = nop and pc(S,P)
= l3 and (P = Q) = false) implies (((pc(S,Q)
= l6 and lock(S,Q) = false) or pc(S,Q) = cs or
pc(S,Q) = l7 or pc(S,Q) = l8 or pc(S,Q) = l10
or pc(S,Q) = l11) = false)) .
```

The proof of `inv1` needs totally fewer lemmas than that of `inv1'`. More precisely, the set of lemmas that need to be used in the proof of `inv1` is a subset of those in the proof of `inv1'`.

B. *The Other Lemmas*

The other lemmas used to prove that `mutex(S,P,Q)` for all process IDs `P` & `Q` is an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$ are as follows:

```
eq inv2(S,P,Q) = ((pc(S,P) = l3 and prede(S,P)
= nop and pc(S,Q) = l3 and (P = Q) = false)
implies (prede(S,Q) = nop) = false) .
```

```
eq inv3(S,P) = (pc(S,P) = l5
implies lock(S,P) = true) .
```

```
eq inv4(S,P) = (((pc(S,P) = l3 and prede(S,P)
= nop) or (pc(S,P) = l6 and lock(S,P) = false)
or pc(S,P) = cs or pc(S,P) = l7 or pc(S,P)
= l8 or pc(S,P) = l10 or pc(S,P) = l11)
implies (glock(S) = nop) = false) .
```

```
eq inv5(S,P,Q) = ((next(S,Q) = P and (P = Q)
= false and (pc(S,Q) = l12 or pc(S,Q) = l1 or
pc(S,Q) = rs) = false) implies (prede(S,P) = Q
and pc(S,P) = l6 and lock(S,P) = true)) .
```

```
eq inv6(S,P,Q) = ((pc(S,Q) = l6 and lock(S,Q)
= false and (P = Q) = false) implies
((pc(S,P) = l6 and lock(S,P)= false) or
pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8
or pc(S,P) = l10 or pc(S,P) = l11) = false) .
```

```
eq inv7(S,P,Q) = (((pc(S,Q) = l11 or pc(S,Q)
= l10 or pc(S,Q) = l8 or pc(S,Q) = l7 or
pc(S,Q) = cs) and (P = Q) = false) implies
((pc(S,P) = l6 and lock(S,P) = false) or
pc(S,P) = cs or pc(S,P) = l7 or pc(S,P) = l8
or pc(S,P) = l10 or pc(S,P) = l11) = false) .
```

Let us partially give the explanation for `inv4`. `inv4` says that if there exists a process `P` located at `cs`, or `l7`, or `l8`, or `l10`, or `l11`, or `l6` and `lock` of `P` is `false`, or `l3` and `prede` of `P` is `nop`; then `glock` can not be `nop`. The remaining lemmas can be understood likewise. Let us repeat again that we did not come up with the seven lemmas from the beginning, but we have gradually constructed each of them when conduct formal verification. For example, in the proof of `mutex`, `inv1` is constructed; or `inv2` together with `inv3` and `inv4` are constructed when we try to prove `inv1`.

In addition to `inv1`, the proof of `mutex` also requires the use of `inv6`. The proof of `inv1` uses `inv2`, `inv3` and `inv4` as lemmas. The proof of `inv2` uses `inv4` as a lemma. The proof of `inv3` uses `inv5` as a lemma and vice versa. The proofs of `inv4` and `inv6` use `inv1`, `inv3` and `inv7` as lemmas. `inv1`, `inv3` and `inv6` are required in the proof of `inv7`.

C. *Use of Lemma Weakening (LW)*

While proving that MCS enjoys the mutual exclusion property, use of LS (but not use of LW), together with case splitting, etc., did not seem to make our proof attempt converge. Use of LW made it converge. There are two cases where we used LW. We describe how to use LW in the two cases in detail.

1) Case 1: Initially, we used the following `inv40` instead of `inv4`:

```
eq inv40(S,P) = ((pc(S,P) = l3 or pc(S,P) = l4
or pc(S,P) = l5 or pc(S,P) = l6 or pc(S,P) = cs
or pc(S,P) = l7 or pc(S,P) = l8 or
pc(S,P) = l10 or pc(S,P) = l11)
implies (glock(S) = nop) = false .
```

`inv40` is obtained by strengthening `inv4`. Let us note that `40` in the notation `inv40` does not mean that there are 40 or more invariants that have been conjectured. We put confidence in that whenever there exists a process `P` located at `l3` or `l6` (or `cs`, or `l7`, or `l8`, or `l10`, or `l11` as well), `glock` can not be `nop`. Thus, we strongly believe that strengthening `inv4` to obtain `inv40` is the correct way to complete the formal verification (similar to the way we strengthen `inv1'` and `inv1''` to obtain `inv1`). Accordingly, we believe that `inv40` is truly an invariant wrt $\mathcal{S}_{\mathrm{MCS}}$. Let us consider a sub-case of the induction case `chglk` for the proof attempt of `inv40`. The open-close fragment of the sub-case is as follows:

```
open INV .
op s : -> Sys . ops p r : -> Pid .
eq pc(s,r) = l8 . eq (p = r) = false .
eq glock(s) = r . eq pc(s,p) = l3 .
red inv40(s,p) implies inv40(chglk(s,r),p) .
close
```

Let v_{40} be an arbitrary state in which the four equations used in the fragment hold. CafeOBJ returns `false` for the fragment. This is why we need a lemma to discharge the sub-case. By strengthening the lemma constructed straightforwardly from the four equations used in the fragment, we obtain the following lemma:

```
eq inv41(S,P,Q) = ((pc(S,P) = l3 or pc(S,P)
= l4 or pc(S,P) = l5 or pc(S,P) = l6 or pc(S,P)
= cs or pc(S,P) = l7 or pc(S,P) = l8 or pc(S,P)
= l10 or pc(S,P) = l11) and glock(S) = Q and
(P = Q) = false) implies (pc(S,Q) = cs or
pc(S,Q) = l7 or pc(S,Q) = l8 or pc(S,Q) = l10
or pc(S,Q) = l11 or (pc(S,Q) = l6 and
lock(S,Q) = false)) = false .
```

`inv41` can be used as a lemma to discharge the sub-case. While proving `inv41`, we encounter three sub-cases of the induction case `stlnx`. One of the three sub-cases is characterized by the following equations: `pc(s,r) = l11`, `next(s,r) = q`, `glock(s) = q`, `pc(s,p) = l3`, `pc(s,q) = l6`, `lock(s,q) = true`, `(p = r) = false`, `(q = r) = false` and `(p = q) = false`. The other two sub-cases are characterized by almost the same equations. The only difference is `pc(s,p) = l3`, instead of which `pc(s,p) = l4` and `pc(s,p) = l5` hold for the other two sub-cases, respectively. Let v_{41} be an arbitrary state that corresponds to any of the three sub-cases. `inv41(s,p,q) implies inv41(stlnx(s,r),p,q)` reduces to `false` and then we need a lemma to discharge the three sub-cases. One possible lemma is as follows:

```
eq inv42(S,P,Q,R) = (glock(S) = Q and next(S,R)
= Q and (pc(S,R) = cs or pc(S,R) = l7 or
pc(S,R) = l8 or pc(S,R) = l10 or pc(S,R) = l11
or (pc(S,R) = l6 and lock(S,R) = false)) and
(P = R) = false and (Q = R) = false and (P = Q)
= false) implies (pc(S,P) = l3 or pc(S,P) = l4
or pc(S,P) = l5 or pc(S,P) = l6) = false .
```

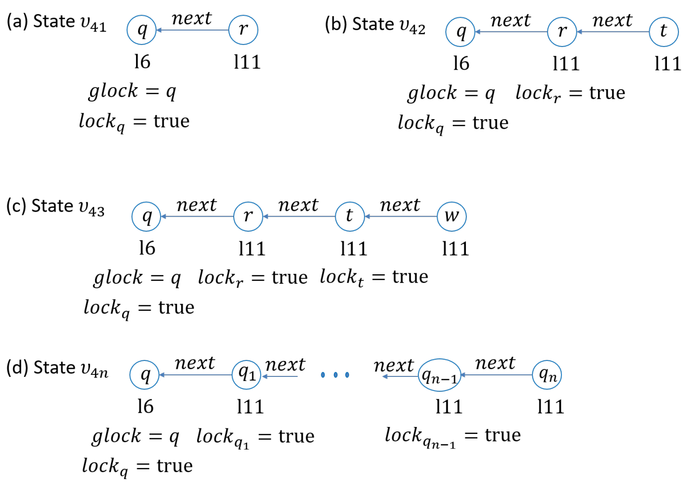


Fig. 3. States v_{41}, v_{42}, v_{43} & v_{4n}

`inv42` can be used as a lemma to discharge the three sub-cases. While proving `inv42`, we encounter three sub-cases of the induction case `stlnx`. One of the three sub-cases is characterized by the following equations: `pc(s,t) = l11`, `(p = t) = false`, `(q = t) = false`, `(r = t) = false`, `(p = r) = false`, `(q = r) = false`, `(p = q) = false`, `next(s,t) = r`, `next(s,r) = q`, `glock(s) = q`, `pc(s,q) = l6`, `lock(s,q) = true`, `pc(s,r) = l6`, `lock(s,r) = true` and `pc(s,p) = l3`. The other two sub-cases are characterized by almost the same equations. The only difference is `pc(s,p) = l3`, instead of which `pc(s,p) = l4` and `pc(s,p) = l5` hold for the other two sub-cases, respectively. Let v_{42} be an arbitrary state that corresponds to any of the three sub-cases. `inv42(s,p,q,r) implies inv42(stlnx(s,t),p,q,r)` reduces to `false` and then we need a lemma to discharge the three sub-cases.

What if we keep on doing the proof attempt as we did? Let us partially visualize v_{41} as shown in Fig. 3 (a). Fig. 3 (a) visually says that `q` is located at l6, `r` is located at l11, $next_{\mathtt{r}}$ is `q`, $lock_{\mathtt{q}}$ is true and *glock* is `q`. Let us partially visualize v_{42} as shown in Fig. 3 (b). The difference between v_{41} and v_{42} can be visually observed from Fig. 3 (a) and (b). One process located at l6 is inserted between the two processes in Fig. 3 (a) and its *lock* is true, although `t` is used in Fig. 3 (b) instead of `r` in Fig. 3 (a). If we conjecture a lemma, say `inv43`, that can be used to discharge the three sub-cases that correspond to v_{42} as we conjecture `inv41` and `inv42`, we encounter some sub-cases in which `inv43(s,p,q,r,t) implies inv42(stlnx(s,w), p,q,r,t)` reduces to `false` while proving `inv43`. Let v_{43} be an arbitrary state that corresponds to any of the sub-cases. Fig. 3 (c) shows the diagram that partially visualizes v_{43}. The difference between Fig. 3 (b) and (c) is essentially the same as that of Fig. 3 (a) and (b). One more process located at l6 such that its $lock$ is true is inserted into the structure constructed with $next$ variables. The structure virtually forms the queue in which processes that want to enter the critical section wait. If we repeat what we did, we will encounter the situation that can be partially visualized as shown in Fig. 3 (d), which suggests that this way to conjecture lemmas never converges.

There must be a generic lemma that is stronger than `inv41`, `inv42`, etc. like $\text{rev}(\text{rev}(L_1) \mathbin{@} L_2) = \text{rev}(L_2) \mathbin{@} L_1$ for the proof of $\text{rev}(\text{rev}(L)) = L$, but we could not construct such a generic one. Instead, we made `inv40` weaker, constructing `inv4`. By observing some graphical animations of MCS, we realized that there exists at most one process p except for processes q such that (1) q is located at l3 and $prede_q$ is not nop and (2) q is located at l6 and $lock_q$ is true in extended CS region, where extended CS region consists of cs, l7, l8, l10, l11, l3 and l6 [6]. From this observation, we conjectured that whenever there exists such a process p in extended CS region, the virtual queue at least consists of p as an element, which implies that $glock$ is not nop. This is `inv4` that is weaker than `inv40`. Tackling the induction case `chglk` for the proof of `inv4`, we encounter a sub-case in which `inv4(s,p) implies inv4(chglk(s,r),p)` reduces to `false`. The proof score fragment of the sub-case is as follows:

```
open INV .
op s : -> Sys . ops p r : -> Pid .
eq pc(s,r) = l8 . eq (p = r) = false .
eq glock(s) = r . eq pc(s,p) = l3 .
eq prede(s,p) = nop .
red inv4(s,p) implies inv4(chglk(s,r),p) .
close
```

The only difference between the sub-cases of the `inv4` and `inv40` is the existence of `prede(s,p) = nop` in the sub-case of `inv4`. The sub-case of `inv4` can be discharged by using `inv1` as a lemma. The proof of `inv4` needs `inv3` and `inv7` as lemmas as well. Note that v_{40} is not only the sub-case in which `inv40(s,p) implies inv40(`t_{MCS}`(s, r),p)` reduces to `false`, where t_{MCS} is a transition of $\mathcal{S}_{\text{MCS}}$, but also there are eight more sub-cases such that the term (or formula) reduces to `false`. The eight more sub-cases are characterized by almost the same equations of v_{40}, except the only difference is `pc(s,p) = l4`, `pc(s,p) = l5`, `pc(s,p) = l6`, `pc(s,p) = cs`, `pc(s,p) = l7`, `pc(s,p) = l8`, `pc(s,p) = l10` and `pc(s,p) = l11` hold for the eight sub-cases, respectively, instead of `pc(s,p) = l3` in v_{40}. We need to conjecture new lemmas for the first three sub-cases like v_{40}, while the latter five cases can be discharged by using `inv7` as a lemma.

We strongly believe that `inv40` as well as `inv41` and `inv42` are invariants wrt $\mathcal{S}_{\text{MCS}}$. We were, however, not able to construct any generic lemma that is stronger than all of `inv41`, `inv42`, etc., and therefore we have not successfully completed the proof of `inv40`. Accordingly, we cannot guarantee that `inv40` is actually an invariant wrt $\mathcal{S}_{\text{MCS}}$ when we submit the present paper to the conference.

2) Case 2: Let us consider the lemma `inv50` obtained by deleting `(P = Q) = false` from `inv5`:

```
eq inv50(S,P,Q) = ((next(S,Q) = P and (pc(S,Q)
= l12 or pc(S,Q) = l1 or pc(S,Q) = rs) = false)
implies (pc(S,P) = l6 and lock(S,P) = true
and prede(S,P) = Q)) .
```

`inv50` is stronger than `inv5` or equivalently `inv5` is weaker than `inv50`. Since $next$ variables are used to virtually construct a queue of process IDs, we put confidence in that $next_p$ never has p as its value. Therefore, we also strongly believe that `inv50` is an invariant wrt $\mathcal{S}_{\text{MCS}}$ if `inv5` is an invariant wrt $\mathcal{S}_{\text{MCS}}$. We realized, however, that their proofs are totally different when we tried to prove that `inv50` is an invariant wrt $\mathcal{S}_{\text{MCS}}$.

The proof of `inv5` only uses `inv3` as a lemma and the proof of `inv3` only uses `inv5` as a lemma (again, this is not a circular as explained in Sect. II). On the other hand, the proof of `inv50` requires two more lemmas `inv51` and `inv53` in addition to `inv3`. The proof of `inv51` requires `inv3`, `inv50`, `inv52` and `inv53` as lemmas. The proof of `inv52` requires `inv53` as a lemma. `inv51`, `inv52` and `inv53` are as follows:

```
eq inv51(S,P,Q) = ((pc(S,P) = l12 or pc(S,P) =
l1 or pc(S,P) = rs) = false and (pc(S,Q) = l12
or pc(S,Q) = l1 or pc(S,Q) = rs) = false and
(next(S,Q) = nop) = false and (P = Q) = false)
implies (next(S,P) = next(S,Q)) = false .

eq inv52(S,P) = ((next(S,P) = P) = false) .

eq inv53(S,P) = ((prede(S,P) = P) = false) .
```

Since we suppose that each of $next_p$ and $prede_p$ for every process p is initially set to nop, $next_p$ variables are used to virtually construct a queue and $prede_p$ variables are used to enqueue process IDs as elements into the virtual queue, we are sure that $next_p$ never has p as its value and $prede_p$ never has p as its value. We realized, however, that it is not that straightforward to prove `inv53`. The proof of `inv53` requires a new lemma `inv54` whose proof needs five more new lemmas `inv55`, `inv56`, `inv57`, `inv58` and `inv59` to complete the proof of `inv53`. Please refer to http://gitlab.com/duongtd23/mcs/ for `inv`i for $i = 54, 55, 56, 57, 58, 59$. Let us repeat again that `59` in the notation `inv59` does not mean that there are 59 invariants that have been conjectured.

VI. Related Work

Rushby [7] has demonstrated that use of disjunctive invariants $q_1 \vee \ldots \vee q_n$ makes invariant verification easier for synchronous concurrent (and/or distributed) systems. His technique proves that $p \wedge (q_1 \vee \ldots \vee q_n)$ is an inductive invariant wrt a system so as to prove that p is an invariant wrt the system. LW, together with LS, can be regarded as a generalized version of his technique. Instead of $p \wedge q_1 \wedge \ldots \wedge q_i \wedge \ldots \wedge q_n$, we prove that $p \wedge q_1 \wedge \ldots \wedge q'_i \wedge \ldots \wedge q_{n'}$ is an inductive invariant wrt a system, where q'_i is weaker than q_i (and n' is much less than n in our case study). q'_i may be in the form $q'_{i1} \vee \ldots \vee q'_{im}$. We have demonstrated that LW can be effective for asynchronous concurrent (and/or distributed) systems as well.

Wang [8] has proved that it is impossible to automatically prove that concurrent software systems in which multiple processes run algorithms on data structures with pointers enjoy desired properties if there are an arbitrary number of processes. Then, a new approximation method has been proposed to formally verify such software systems. The key idea is to construct a finite collective image set (CIS) whose elements are reachable state representations (or global data-structure image - GDSI). The verification can be done by enumerating all GDSIs reachable from the initial state. He has used the proposed method to prove that a revised version of MCS protocol enjoys desired properties. The proofs described in the paper, however, are in Mathematical argumentation but not formal. It would at least not be straightforward to develop a tool that fully supports his verification technique.

Kim, et al. [9] have used the methodology of certified concurrent abstraction layers to conduct a case study in which they prove that MCS enjoys the lockout freedom property (a liveness property) as well as the mutual exclusion property (a safety property). They have defined five layers such that the lowest one is the implementation of MCS in C/assembly languages and the higher ones are more abstract than the implementation. They have formally proved with Coq, a proof assistant, that each layer except for the highest one contextually refines the one-step higher layer. Their paper mainly focuses on their contextual refinement approach to integration of the verified algorithm, such as MCS, into a larger system, such as an OS.

To prove that p is an invariant wrt $\mathcal{S}$, our method tries to find an inductive invariant q wrt $\mathcal{S}$ such that $q(\upsilon) \Rightarrow p(\upsilon)$ for all states $\upsilon \in \Upsilon$. The well-known model checking algorithm IC3 [10] also can be used for discovering the inductive invariants. Given a finite-state transition system S and a property P that we want to check whether P is invariant for the system S or not, IC3 will gradually refine P, eventually producing either an inductive invariant P' that is stronger than P or a counterexample trace. However, IC3 can not be used to check that the state machine formalizes MCS satisfies the mutual exclusion property or not. The reason is that IC3 only can accept finite-state systems, can not deal with infinite-state systems such as the state machine formalizes MCS. Basically, that is the disadvantage of any model checking techniques/tools that we mentioned at the very beginning of the paper. Our method presented in this paper bases on theorem proving that can get rid of this disadvantage.

Ogata and Futatsugi [11] have reported on a case study in which they have *semi-formally* (but not formally) verified that MCS enjoys the mutual exclusion property and the lockout freedom property in CafeOBJ, although they claimed that their verification is formal. Since their proofs are semi formal, however, they may have overlooked several subtle cases and then do not discuss anything about what we have encountered. In addition to formal verification by writing proof scores in CafeOBJ, we have used CiMPA and CiMPG [12], a proof assistant and a proof generator for CafeOBJ, to confirm that our proofs (or proof scores) are correct. Based on the proof scores we wrote, we manually wrote proof scripts for CiMPA proving that MCS enjoys the mutual exclusion property. CiMPG was used to infer proof scripts from our proof scores and the proof scripts inferred by CiMPG were successfully deal with by CiMPA. In summary, we triple-checked our proof scores.

VII. Conclusion

We have demonstrated the power of LW by using MCS as an example. We were not able to complete the formal proof that MCS enjoys the mutual exclusion property without use of LW. We had stuck in the proof attempt of `inv40` for several months until the first author of the present paper came up with `inv4` that is weaker than `inv40`. `inv4`, together with `inv5`, made us successfully complete the formal proof that MCS enjoys the mutual exclusion property.

For each non-trivial invariant proof, we need to conjecture lemmas that are also invariants on the fly during the proof. We cannot always conjecture the best lemma every time we need to use a lemma. The first lemma we construct may be too weak or strong. Therefore, we may need to strengthen or weaken it. Accordingly, it is natural that it is necessary to use LW as well as LS. To the best knowledge of ours, however, LW has been rarely used in formal methods.

One piece of our future work is to conduct more case studies that demonstrate the power of LW. Another piece of our future work is to come up with a systematic (or hopefully automatic) way to use LW, which will be conducted by consulting some systematic and/or automatic ways to use LS, such as those used by Creme [13], an automatic invariant prover for OTSs specified in Maude.

References

[1] Z. Manna and A. Pnueli, *Temporal verification of reactive systems - safety*. Springer, 1995.

[2] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, no. 1, pp. 21–65, Feb. 1991.

[3] K. Ogata and K. Futatsugi, "Proof scores in the OTS/CafeOBJ method," in *FMOODS 2003*, 2003, pp. 170–184.

[4] R. Diaconescu and K. Futatsugi, *Cafeobj Report*, ser. AMAST Series in Computing. World Scientific, 1998, vol. 6.

[5] K. Ogata and K. Futatsugi, "A combination of forward and backward reachability analysis methods," in *12th ICFEM*, 2010, pp. 501–517.

[6] D. D. Bui and K. Ogata, "Better state pictures facilitating state machine characteristic conjecture," in *Submitted for publication*, 2020.

[7] J. M. Rushby, "Verification diagrams revisited: Disjunctive invariants for easy verification," in *12th CAV*, 2000, pp. 508–520.

[8] F. Wang, "Automatic verification of pointer data-structure systems for all numbers of processes," in *FM '99*, 1999, pp. 328–347.

[9] J. Kim, V. Sjöberg, R. Gu, and Z. Shao, "Safety and liveness of MCS lock - layer by layer," in *15th APLAS*, 2017, pp. 273–297.

[10] A. R. Bradley, "Sat-based model checking without unrolling," in *Verification, Model Checking, and Abstract Interpretation*, R. Jhala and D. Schmidt, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2011, pp. 70–87.

[11] K. Ogata and K. Futatsugi, "Formal verification of the MCS list-based queuing lock," in *5th ASIAN*, 1999, pp. 281–293.

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

[13] M. Nakano, K. Ogata, M. Nakamura, and K. Futatsugi, “Crème: an automatic invariant prover of behavioral specifications,” *IJSEKE*, vol. 17, no. 6, pp. 783–804, 2007.