Specification and Verification of Some Classical Mutual Exclusion Algorithms with CafeOBJ

Kazuhiro Ogata and Kokichi Futatsugi

JAIST ({ogata, kokichi}@jaist.ac.jp)

Abstract. We have specified and verified some classical mutual exclusion algorithms with CafeOBJ by adopting UNITY computational model and its logic. Two properties of each mutual exclusion algorithm have been proven with CafeOBJ, together with UNITY logic and simulation relations. One property is a safety property that more than one process can never enter their critical section simultaneously, and the other a liveness property that a process wanting to enter a critical section eventually enters there. In this paper, we describe the specification and verification of mutual exclusion algorithms.

1 Introduction

UNITY [2] is a parallel computational model, and a specification and programming logic. It also provides a proof system based on the logic that is an extension of Floyd-Hoare logic [4, 7] and is influenced by temporal logic [11]. On the other hand, CafeOJB [3, 5, 9], an algebraic specification language, provides notational machinery so as to specify labeled transition systems, which is similar to UNITY computational model, and the corresponding semantics, namely, *hidden algebra* [6].

We have adopted the UNITY computational model to specify mutual exclusion algorithms in CafeOBJ, and used UNITY logic and simulation relations [8] to verify a safety and a liveness properties of mutual exclusion algorithms with CafeOBJ. A safety property is often interpreted as saying that some particular bad thing never happens, while a liveness property is often informally understood as saying that some particular good thing eventually happens. The safety property of each mutual exclusion algorithm, which has been proven, is that more than one process can never enter their critical section simultaneously, and the liveness property of each of them, which has also been proven, is that a process wanting to enter a critical section eventually enters there. The mutual exclusion algorithms specified in CafeOBJ are Peterson algorithm [10], Ticket algorithm, and Anderson algorithm [1]. In this paper, we describe the specification and verification of the mutual exclusion algorithms with CafeOBJ.

The rest of the paper is organized as follows. Section 2 briefly explains UNITY. In Sect. 3, we introduce how CafeOBJ specifies UNITY computational model, and verifies safety and liveness properties using some small example. In Sect. 4, Peterson algorithm is treated, and in Sect. 5, Ticket and Anderson algorithms are treated. Finally, Sect. 6 gives a conclusion.

2 UNITY

UNITY [2] is a parallel computational model, and a specification and programming logic. It also provides a proof system based on the logic that is an extension of Floyd-Hoare logic [4, 7] to parallel programs, and is also influenced by temporal logic [11]. UNITY has a minimum notational machinery to represent the parallel computational model. In this section, a brief explanation of UNITY is given.

2.1 Computational Model

The parallel computational model of UNITY is basically a labeled transition system. It has some initial states and finitely many transition rules. Application or execution of one transition rule may simultaneously change (possibly nothing) some of the components that the state of a labeled transition system is composed of. An execution starts from one initial state and goes on forever; in each step of execution some transition rule is chosen nondeterministically and executed. Nondeterministic choice is constrained by the following *fairness* rule: every transition rule is chosen infinitely often.

UNITY has a minimum notational machinery or a programming language to represent the parallel computational model. A program consists of a declaration of variables, a specification of their initial values, and a set of multiple-assignment statements. Since some variables may not be initialized, a labeled transition system has more than one initial state. A multiple-assignment statement corresponds to a transition rule.

Now a small example is given. The following UNITY program *gcd* has the two variables x and y whose initial values are M and N, respectively. It also consists of the two statements; one statement sets x to x minus y provided that x is greater then y, and the other y to y minus x provided that y is greater than x.

```
Program gcd
  declare x, y : integer
  initially x, y = M, N {M and N are positive integers}
  assign x := x − y if x > y [] y := y − x if x < y
end {gcd}
```

2.2 Specification and Programming Logic

The logic is based on assertions of the form $\{p\}\ s\ \{q\}$, denoting that an execution of statement s in any state that satisfies predicate p results in a state that satisfies predicate q, if execution of s terminates. Properties of a UNITY program are expressed using assertions of the form $\{p\}\ s\ \{q\}$, where s is universally or existentially quantified over the statements of the program. The properties are classified into a *safety* or a *liveness* property. Examples of safety properties are that variable x is always positive, and that two processes can never enter their critical sections simultaneously. Examples of liveness properties are that the difference between x and y eventually becomes nothing, and that a process wanting to enter a critical section eventually enters there. Existential quantification over program statements is essential in stating liveness properties, whereas safety properties can be stated using only universal quantifications over statements (and using the initial condition).

Although all properties of a program can be expressed directly using assertions, a few additional terms are introduced for conveniently describing properties of programs: *unless*, *stable*, *invariant*, *ensures*, and *leads-to*. The first three terms are used to state safety properties, and the latter two ones to state liveness properties. The definitions of these terms are given below.

Unless For a given program F, p *unless* q is defined as follows:

$$p \textit{ unless } q \equiv \langle \forall s : s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{p \vee q\} \rangle\,.$$

In other words, if p holds at any point during the execution of F, then either q never holds and p continues to hold forever, or q holds eventually (it may hold initially when p holds) and p continues to hold at least until q holds.

Stable *stable* p is defined as a special case of *unless*:

$$\textit{stable } p \equiv p \textit{ unless } \text{false}\,.$$

A stable predicate remains true once it becomes true (though it may never become true).

Invariant *invariant* p is defined as a special case of *stable*:

$$\textit{invariant } p \equiv (\text{initial condition} \Rightarrow p) \wedge \textit{stable } p\,.$$

An invariant is always true: all states of the program that arise during any execution of the program satisfy all invariants.

Ensures For a given program F, p *ensures* q is defined as follows:

$$p \textit{ ensures } q \equiv (p \textit{ unless } q) \wedge \langle \exists s : s \text{ in } F :: \{p \wedge \neg q\}\ s\ \{q\} \rangle .$$

In other words, if p is true at some point in the computation, p remains true as long as q is false (from p *unless* q), and eventually q becomes true.

Leads-to ($\mapsto$) A given program has the property $p \mapsto q$ if and only if this property can be derived by a finite number of applications of the following inference rules:

- (**Basis**) $\dfrac{p \textit{ ensures } q}{p \ \mapsto \ q}$,
- (**Transitivity**) $\dfrac{p \ \mapsto \ q,\ q \ \mapsto \ r}{p \ \mapsto \ r}$,
- (**Disjunction**) For any set W, $\dfrac{\langle \forall m : m \in W :: p(m) \ \mapsto \ q \rangle}{\langle \exists m : m \in W :: p(m) \rangle \ \mapsto q}$.

3 UNITY in CafeOBJ

3.1 UNITY Computational Model in Hidden Algebra

CafeOBJ [3,5,9] provides a notational machinery to describe labeled transition systems, and the corresponding semantics, namely, *hidden algebra* [6]. In hidden algebra, a *hidden sort* represents (states of) a labeled transition system. *Action operations* or *actions*, which take the state of a labeled transition system and more than zero data such as integers, and returns another (possibly the same) state of the system, can change the sate of a labeled transition system. The state of a labeled transition system can be observed only using *observation operations* or *observations* that take the state of a labeled transition system and returns the value of a data component in the system.

As an example, a CafeOBJ specification that corresponds to the UNITY program *gcd* is given. The specification consists of one module whose name is GCD. GCD imports another module INT where some sorts and operations relating to integers are declared. By importing INT, integers and the related operators such as addition and subtraction can be used. GCD declares one hidden sort `State` that represents (states of) the labeled transition system that the UNITY program *gcd* represents. The main part of the signature is as follows:

```
-- initial state
  op  init : -> State
-- observations
  bops x y  : State -> NzNat
-- actions
  bops update-x update-y : State -> State
```

GCD has two (atomic) actions `update-x` and `update-y` that correspond to the first and second assignment statements in the UNITY program *gcd*, respectively. It also has two observations `x` and `y` with which we can observe the values of the variables x and y, respectively. `init` is one initial state of GCD. GCD has three sets of equations: one for initial state `init`, and the others for the two states after one of the two actions `update-x` and `update-y` have been executed, respectively. We are giving the three sets of equations.

```
  eq  x(init) = M .  eq  y(init) = N .
```

The constants `M` and `N` are the initial values of the variables x and y, respectively.

```
  ceq x(update-x(S)) = x(S) - y(S) if x(S) > y(S) .
  ceq x(update-x(S)) = x(S)        if x(S) <= y(S) .
  eq  y(update-x(S)) = y(S) .
```

`S` is a variable for `State`. The three equations correspond to the first assignment statement in the UNITY program *gcd*.

```
  eq  x(update-y(S)) = x(S) .
  ceq y(update-y(S)) = y(S) - x(S) if x(S) < y(S) .
  ceq y(update-y(S)) = y(S)        if x(S) >= y(S) .
```

The three equations correspond to the second assignment statement in the UNITY program *gcd*.

3.2 Verification of Safety and Liveness Properties with CafeOBJ

The following two properties are proven from the CafeOBJ specification GCD with CafeOBJ. One is a safety property, and the other a liveness property. The liveness property means that the difference between x and y eventually becomes nothing.

1. *invariant* $x > 0 \land y > 0$,
2. true $\mapsto x = y$.

Proof sketch 1: Since the initial values `M` and `N` of x and y are positive, the predicate clearly holds in the initial state. Thus, suppose that the predicate holds, we confirm that it is preserved after execution of every action. First the following module is declared, in which the needed precondition is given.

```
mod GCD-PROOF1 {
   pr( GCD )
   op s : -> State  ops m n : -> NzNat  eq x(s) = m.  eq y(s) = n .
}
```

We can show the predicate (i.e. $x > 0 \land y > 0$) is preserved after the action `update-x` is executed by having the CafeOBJ system executes the following two proof scores: one for the case $x > y$, and the other for the case $x \le y$.

```
open GCD-PROOF1
  eq m > n = true .  eq m + (- n) > 0 = true .
  red x(update-x(s)) > 0 and y(update-x(s)) > 0 .
close
open GCD-PROOF1
  eq m <= n = true .  red x(update-x(s)) > 0 and y(update-x(s)) > 0 .
close
```

As is the above case, we can show the predicate is preserved after `update-y` is executed. Therefore, it follows that the safety property holds in GCD. □

Proof sketch 2: First suppose that the following two subproperties hold in GCD. In this paper, properties are often written without explicit quantifications; these are universally quantified over all values of the free variables such as m and n occurring in them.

i. $x > y \land x = m \land y = n$ *ensures* $x < m \land y = n$,
ii. $x < y \land x = m \land y = n$ *ensures* $x = m \land y < n$.

From the two subproperties using Basis inference rule for *leads-to* and Finite Disjunction (see Appendix), the following property is obtained: $x \neq y \land (x, y) = (m, n) \mapsto (x, y) \prec (m, n)$, where $\prec$ is the lexicographic ordering relation on the set of pairs of natural numbers. By applying Corollary of Induction for Leads-to (see Appendix) on the pair (m, n) to this property, the desired property "true $\mapsto x = y$" is derived. □

Next we show the assumed subproperties actually hold in GCD.

Proof sketch 2.1: First the following module is declared, in which the needed precondition is given.

```
mod GCD-PROOF2 {
  pr( GCD )
  op s : -> State  ops m n : -> NzNat
  eq x(s) = m .  eq y(s) = n .
  eq m > n = true .  eq m >= n = true .  eq m + (- n) < m = true .
}
```

We can show the first subproperty holds in `GCD` by executing the following proof score.

```
open GCD-PROOF2
  red x(update-x(s)) < m and y(update-x(s)) == n .
  red x(update-y(s)) == m and y(update-y(s)) == n .
close
```

The second subproperty can be shown almost the same way as the first subproperty. □

4 Two-Process Mutual Exclusion

In this and the next sections, two-process and N-process mutual exclusion algorithms are treated. For each algorithm in this and the next sections, the following two properties are proven:

ME1 Two processes (or more generally more than one process) can never enter their critical section simultaneously.
ME2 A process wanting to enter a critical section eventually enters there.

4.1 Simplified Peterson Algorithm

Peterson algorithm [10] need not sophisticated atomic operations. It can be implemented using usual `load` and `store` instructions. In Peterson algorithm, one shared variable *turn* is used, and each process $i \in \{0,1\}$ has its own local variable $\mathit{flag}[i]$ that can be only read by the opponent process $\hat{i}$ $(= i+1 \bmod 2)$. The code, in a traditional style, is given below. The initial value of $\mathit{flag}[i]$ is false; the initial value of *turn* is arbitrary.

```
                  ...
$\mathit{flag}[i]$ := true
$\mathit{turn}$ := $i$
repeat while $\mathit{flag}[\hat{i}] \wedge \mathit{turn} = i$
Critical Section
$\mathit{flag}[i]$ := false
                  ...
```

We first specify a simplified Peterson algorithm with large granularity in CafeOBJ. The simplified version can be easily vefiied w.r.t. the two properties **ME1** and **ME2**. After the verification, more realistic version is described.

Specification We describe the CafeOBJ specification of the simplified Peterson algorithm. It is called SS-PETERSON2P. SS-PETERSON2P divides the simplified Peterson algorithm into three atomic actions: `try`, `enter`, and `leave`. `try` corresponds to the first two assignments $\mathit{flag}[i] :=$ true and $\mathit{turn} := i$, and does the two assignments simultaneously. `enter` corresponds to the **repeat while** statement, and `leave` to the last assignment.

In SS-PETERSON2P, each process has three possible states: `t0`, `h0`, and `e0`. That a process is in `t0` means it is executing any code but the simplified Peterson algorithm, a process changes its state to `h0` whenever it executes `try` only if it is in `t0`, and that a process is in `e0` means it is executing the critical section.

Besides the three atomic actions, there are three observations: `p`, `flag`, and `turn`. `p` and `flag` are used to observe the state and the *flag*'s value of each process, respectively, and `turn` to observe the *turn*'s value. The main part of the signature of the specification is as follows:

```
-- initial state
  op init0 : -> SState0
-- observations
  bop p    : Nat SState0 -> PState0
  bop flag : Nat SState0 -> Bool
  bop turn : SState0 -> Nat
-- actions
  bop try   : Nat SState0 -> SState0
  bop enter : Nat SState0 -> SState0
  bop leave : Nat SState0 -> SState0
```

`SState0` is a hidden sort that represents the state of the simplified Peterson algorithm, and `Nat`, `Bool`, and `PState0` are visible sorts that represent natural numbers, booleans, and the states of processes. `init0` is one initial state of the simplified Peterson algorithm.

SS-PETERSON2P has four sets of equations: one for one initial state `init0`, and the others for the three states after a process has executed `try`, `enter`, and `leave`, respectively. We are giving the four sets of equations.

```
-- 0. in the initial state.
  eq  p(I, init0) = t0 .
  eq  flag(I, init0) = false .
```

I is a variable for natural numbers that are used to identify each process. In SS-PETERSON2P, only two natural numbers 0 and 1 are used. In the initial state, every process are in `t0`, and every *flag* is set false. The value of *turn* is arbitrary.

```
-- 1. after execution of 'try'
  ceq p(I, try(I, S)) = h0      if p(I, S) == t0 .
  ceq p(J, try(I, S)) = p(J, S) if J =/= I or p(J, S) =/= t0 .
  ceq flag(I, try(I, S)) = true       if p(I, S) == t0 .
  ceq flag(J, try(I, S)) = flag(J, S) if J =/= I or p(J, S) =/= t0 .
  ceq turn(try(I, S)) = I       if p(I, S) == t0 .
  ceq turn(try(I, S)) = turn(S) if p(I, S) =/= t0 .
```

J is also a variable for natural numbers, and S for the states of the simplified Peterson algorithm. The first two equations, the next two ones, and the last two ones prescribe how each process's state, each process's *flag*, and *turn* change after a process executes `try`. The set of equations corresponds to the following two UNITY assignment statements:

$$\langle [\!]\, i \ : \ 0 \leq i \leq 1 \ :: \ p_i,\ \mathit{flag}_i,\ \mathit{turn} := \mathrm{h0},\ \mathrm{true},\ 0 \quad \mathrm{if}\ p_i = \mathrm{t0} \rangle\,,$$

where p_i and flag_i are variables for the state and the *flag* of a process $i \in \{0, 1\}$, respectively.

```
-- 2. after execution of 'enter'
  ceq p(I, enter(I, S)) = e0
      if p(I, S) == h0 and flag(I, S) and (not (flag(opponent(I), S) and turn(S) == I)) .
  ceq p(J, enter(I, S)) = p(J, S)
      if J =/= I or p(I, S) =/= h0 or not flag(J, S) or (flag(opponent(J), S) and turn(S) == J) .
  eq  flag(J, enter(I, S)) = flag(J, S) .
  eq  turn(enter(I, S)) = turn(S) .
```

`opponent` is a function that takes as arguments 0 (or 1) and returns 1 (or 0). The first two equations prescribe a process $i \in \{0, 1\}$ changes its state to `e0` whenever it executes `enter` only if it is in `h0`, its *flag* is true, and its opponent process's *flag* is false or *turn* is its opponent process's ID, and otherwise the process i does not change its state. The last two equations say *flag*'s and *turn* keep remaining even if any process executes `enter`. The corresponding UNITY assignment statements are as follows:

$$\langle [\!]\, i \ : \ 0 \leq i \leq 1 \ :: \ p_i := \mathrm{h0} \quad \mathrm{if}\ p_i = \mathrm{h0} \wedge \mathit{flag}_i \wedge \neg(\mathit{flag}_{\hat{i}} \wedge \mathit{turn} = i) \rangle\,,$$

where $\hat{i}$ is $i \bmod 2$.

```
-- 3. after execution of 'leave'
  ceq p(I, leave(I, S)) = t0      if p(I, S) == e0 .
  ceq p(J, leave(I, S)) = p(J, S) if J =/= I or p(J, S) =/= e0 .
  ceq flag(I, leave(I, S)) = false      if p(I, S) == e0 .
  ceq flag(J, leave(I, S)) = flag(J, S) if J =/= I or p(J, S) =/= e0 .
  eq  turn(leave(I, S)) = turn(S) .
```

The first two equations and the next two ones prescribe a process i changes its statet to `t0` and the i's *flag* is set false whenever the process i executes `leave` only if it is in `e0`. The last one says *turn* keeps remaining even if any process executes `leave`. The corresponding UNITY assignment statements are as follows:

$$\langle [\!]\, i \ : \ 0 \leq i \leq 1 \ :: \ p_i,\ \mathit{flag}_i := \mathrm{h0},\ \mathrm{false} \quad \mathrm{if}\ p_i = \mathrm{e0} \rangle\,.$$

Verification We prove the simplified Peterson algorithm w.r.t. the two properties **ME1** and **ME2**. The properties are restated in more formal way as follows:

1. *invariant* $\neg(p_i = \mathrm{e0} \wedge p_{\hat{i}} = \mathrm{e0})$,
2. $p_i = \mathrm{h0} \mapsto p_i = \mathrm{e0}$.

Proof sketch 1: In the initial state, the predicate is vacuously true because every process is in `t0`. Thus we confirm that the predicate is preserved after any process executes any action in a state satisfying the predicate. We prove the property by a case analysis. However, a little thought shows that we need to consider only the cases, $(p_0, p_1) = (\mathrm{e0}, \mathrm{h0})$ and $(\mathrm{h0}, \mathrm{e0})$, because the predicate is clearly preserved after any process executes any action in the other cases. Besides only execution of `enter` by p_1 (or p_0) needs considering in the former case (or in the latter case). We can show the predicate is preserved after p_1 executes `enter` in the state where p_0 and p_1 are in `e0` and `h0`, respectively, by having the CafeOBJ system execute the following proof score:

```
open SS-PETERSON2P
  op s : -> SState0 .
  eq p(0,s) = e0 .  eq p(1,s) = h0 .
  eq flag(0,s) = true .  eq flag(1,s) = true .
  eq turn(s) = 1 .
  red p(0,enter(1,s)) == e0 and p(1,enter(1,s)) == h0 .
close
```

In the above proof score, we use the fact that the two *flag*'s are true and *turn* is 1 if p_0 and p_1 are in `e0` and `h0`, respectively. If the two processes are in `h`, it is trivial that the two *flag*'s are true. Therefore the fact can be shown by confirming that the `h0`-to-`e0` transition of p_0, in the state where both p_0 and p_1 are in `h0`, occurs only if *turn* is 1. The execution of the following two proof scores can show the fact.

```
mod SS-PETERSON2P-PROOF {
  pr(SS-PETERSON2P)
  op s : -> SState0
  eq p(0,s) = h0 .  eq p(1,s) = h0 .
  eq flag(0,s) = true .  eq flag(1,s) = true .
}
open SS-PETERSON2P-PROOF
  eq turn(s) = 0 .
  red p(0,enter(0,s)) == h0 and p(1,enter(0,s)) == h0 .
close
open SS-PETERSON2P-PROOF
  eq turn(s) = 1 .
  red p(0,enter(0,s)) == e0 and p(1,enter(0,s)) == h0 .
close
```

The proof of the latter case can be done the same way as the former case. □

Proof sketch 2: We show the property holds in the case $i = 0$. First suppose the following four subproperties hold:

i. $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{t0}$ *ensures* $p_0 = \mathrm{e0} \vee (p_0 = \mathrm{h0} \wedge p_1 = \mathrm{h0} \wedge \mathit{turn} = 1)$,
ii. $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{h0} \wedge \mathit{turn} = 0$ *ensures* $p_0 = \mathrm{h0} \wedge p_0 = \mathrm{e0}$,
iii. $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{h0} \wedge \mathit{turn} = 1$ *ensures* $p_0 = \mathrm{e0}$,
iv. $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{e0}$ *ensures* $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{t0}$.

From the subpropperties i and ii using Basis inference rule for *leads-to* and Cancelation Theorem (see Appendix), the following subproperty is obtained:

v. $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{t0} \mapsto p_0 = \mathrm{e0}$.

From the subproperties ii and iii using Basis inference rule for *leads-to* and Finite Disjunction (see Appendix), the following subproperty is obtained:

vi. $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{h0} \mapsto p_0 = \mathrm{e0}$.

From the subproperties iv and v using Basis and Transitivity inference rules for *leads-to*, the following subproperty is obtained:

vii. $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{e0} \mapsto p_0 = \mathrm{e0}$.

Then from the three subproperties v, vi and vii using Finite Disjunction twice, the desired property "$p_0 = \mathrm{hungry} \mapsto p_0 = \mathrm{eating}$" is derived. □

We have to prove the four assumed subproperties.

Proof sketch 2.1: In the case that $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{t0}$, none of the actions but `enter` by p_0 and `try` by p_1 can change the system state. Therefore only the execution of `enter` by p_0 and `try` by p_1 needs considering. Besides "*invariant* $p_0 = \mathrm{h0} \wedge p_1 = \mathrm{t0} \Leftrightarrow p_0 = \mathrm{h0} \wedge p_1 = \mathrm{t0} \wedge \mathit{flag}_0 \wedge \neg \mathit{flag}_1 \wedge \mathit{turn} = 0$" is used to prove the subproperty. The execution of the following proof score can show the first subproperty holds in `SS-PETERSON2P`.

```
open SS-PETERSON2P
  op s : -> SState0 .
  eq p(0, s) = h0 .  eq p(1, s) = t0 .
  eq flag(0, s) = true .  eq flag(1, s) = false .
  eq turn(s) = 0 .
  red p(0, enter(0, s)) == e0 .
  red p(0, try(1, s)) == h0 and p(1, try(1, s)) == h0 and turn(try(1, s)) == 1 .
close
```

The other three subproperties can be shown by executing the following three proof scores. As is the case for the first subproperty, a little thought can reduce the number of cases.

```
open SS-PETERSON2P
  op s : -> SState0 .
  eq p(0, s) = h0 .  eq p(1, s) = h0 .
  eq flag(0, s) = true .  eq flag(1, s) = true .
  eq turn(s) = 0 .
  red p(0, enter(0, s)) == h0 and p(1, enter(0, s)) == h0 and turn(enter(0, s)) == 0 .
  red p(0, enter(1, s)) == h0 and p(1, enter(1, s)) == e0 .
close
open SS-PETERSON2P
  op s : -> SState0 .
  eq p(0, s) = h0 .  eq p(1, s) = h0 .
  eq flag(0, s) = true .  eq flag(1, s) = true .
  eq turn(s) = 1 .
  red p(0, enter(0, s)) == e0 .
  red p(0, enter(1, s)) == h0 and p(1, enter(1, s)) == h0 and turn(enter(1, s)) == 1 .
close
open SS-PETERSON2P
  op s : -> SState0 .
  eq p(0, s) = h0 .  eq p(1, s) = e0 .
  eq flag(0, s) = true .  eq flag(1, s) = true .
  eq turn(s) = 0 .
  red p(0, enter(0, s)) == h0 and p(1, enter(0, s)) == e0 .
  red p(0, leave(1, s)) == h0 and p(1, leave(1, s)) == t0 .
close
```

□

4.2 Peterson Algorithm

Next more realistic (refined) version of Peterson algorithm is specified and verified.

Specification Synchronous assignment to *flag*$[i]$ and *turn* can be decoupled provided that *turn* is set to i only after *flag*$[i]$ is set true. A local boolean variable *reg*$[i]$ of p_i is introduced to perform these two assignments in order; *reg*$[i]$ holds if *flag*$[i]$ has been set true and *turn* is yet to be set to i.

In the specification that is called PETERSON2P, there are five atomic actions: `try`, `setflag`, `setturn`, `enter`, and `leave`. Execution of `try` by a process means that the process starts to execute Peterson algorithm. `setflag` corresponds to *flag*$[i]$:= true, `setturn` to *turn* := i, `enter` to the **repeat while** statement, and `leave` to *flag*$[i]$:= false.

In PETERSON2P, each process has five possible states: `t2`, `h2-1`, `h2-2`, `h2-3`, and `e2`. For example, that a process is in `h2-2` means it has executed *flag*$[i]$:= true and is yet to be execute *turn* := i.

There are also four observations in PETERSON2P: `p`, `flag`, `reg`, and `turn`. The main part of the signature is as follows:

```
-- initial state
  op init2 : -> SState2
-- observations
  bop p    : Nat SState2 -> PState2
  bop flag : Nat SState2 -> Bool
  bop reg  : Nat SState2 -> Bool
  bop turn : SState2 -> Nat
-- actions
  bop try     : Nat SState2 -> SState2
  bop setflag : Nat SState2 -> SState2
  bop setturn : Nat SState2 -> SState2
  bop enter   : Nat SState2 -> SState2
  bop leave   : Nat SState2 -> SState2
```

`SState2` is a hidden sort that represents the state of Peterson algorithm, and `PState2` is a visible sort that represents the states of processes. `init2` is one initial state of Peterson algorithm.

PETERSON2P has six sets of equations: one for one initial state `init2`, and the others for the five states after a process has executed `try`, `setflag`, `setturn`, `enter`, and `leave`, respectively. We are giving the six sets of equations.

```
-- 0. in the initial state.
  eq  p(I, init2) = t2 .
  eq  flag(I, init2) = false .
  eq  reg(I, init2) = false .
```

In the initial state, all processes' states, all *flag*'s, and all *reg*'s are `t2`, false, and false, respectively. *turn* is arbitrary.

```
-- 1. after execution of 'try'
  ceq p(I, try(I, S)) = h2-1    if p(I, S) == t2 .
  ceq p(J, try(I, S)) = p(J, S) if J =/= I or p(J, S) =/= t2 .
  eq  flag(J, try(I, S)) = flag(J, S) .
  eq  reg(J, try(I, S)) = reg(J, S) .
  eq  turn(try(I, S)) = turn(S) .

-- 2. after execution of 'setflag'
  ceq p(I, setflag(I, S)) = h2-2    if p(I, S) == h2-1 .
  ceq p(J, setflag(I, S)) = p(J, S) if J =/= I or p(I, S) =/= h2-1 .
  ceq flag(I, setflag(I, S)) = true       if p(I, S) == h2-1 and not flag(I, S) .
  ceq flag(J, setflag(I, S)) = flag(J, S) if J =/= I or p(J, S) =/= h2-1 or flag(J, S) .
  ceq reg(I, setflag(I, S)) = true      if p(I, S) == h2-1 and not flag(I, S) .
  ceq reg(J, setflag(I, S)) = reg(J, S) if J =/= I or p(J, S) =/= h2-1 or flag(J, S) .
  eq  turn(setflag(I, S)) = turn(S) .

-- 3. after execution of 'setturn'
  ceq p(I, setturn(I, S)) = h2-3    if p(I, S) == h2-2 .
  ceq p(J, setturn(I, S)) = p(J, S) if J =/= I or p(I, S) =/= h2-2 .
  eq  flag(J, setturn(I, S)) = flag(J, S) .
  ceq reg(I, setturn(I, S)) = false     if p(I, S) == h2-2 and reg(I, S) .
  ceq reg(J, setturn(I, S)) = reg(J, S) if J =/= I or p(I, S) =/= h2-2 or not reg(J, S) .
  ceq turn(setturn(I, S)) = I       if p(I, S) == h2-2 and reg(I, S) .
  ceq turn(setturn(I, S)) = turn(S) if p(I, S) =/= h2-2 or not reg(I, S) .

-- 4. after execution of 'enter'
  ceq p(I, enter(I, S)) = e2
      if p(I, S) == h2-3 and flag(I, S) and not reg(I, S) and
         not (flag(opponent(I), S) and not reg(opponent(I), S) and turn(S) == I) .
  ceq p(J, enter(I, S)) = p(J, S)
      if J =/= I or p(I, S) =/= h2-3 or not flag(I, S) or reg(I, S) or
         (flag(opponent(I), S) and not reg(opponent(I), S) and turn(S) == I) .
  eq  flag(J, enter(I, S)) = flag(J, S) .
  eq  reg(J, enter(I, S)) = reg(J, S) .
  eq  turn(enter(I, S)) = turn(S) .

-- 5. after execution of 'leave'
  ceq p(I, leave(I, S)) = t2      if p(I, S) == e2 .
  ceq p(J, leave(I, S)) = p(J, S) if J =/= I or p(J, S) =/= e2 .
  ceq flag(I, leave(I, S)) = false      if p(I, S) == e2 .
  ceq flag(J, leave(I, S)) = flag(J, S) if J =/= I or p(J, S) =/= e2 .
  eq  reg(J, leave(I, S)) = reg(J, S) .
  eq  turn(leave(I, S)) = turn(S) .
```

Prior to verifying PETERSON2P w.r.t. **ME1** and **ME2**, SS-PETERSON2P gets altered a little so as to ease the verification. The action `try` in SS-PETERSON2P is decoupled into two actions `try` and `set`. According to this decoupling, the state `h0` is decoupled into two states `h1-1` and `h1-2`, and `t0` and `e0` are renamed `t1` and `e1`, respectively. The new `try` changes nothing but the state of a process that executes `try` to `h1-1` provided that it is in `t1`, and `set` sets *flag* of a process that executes `set` true and *turn* to its ID, and changes the process's state to `h1-2` provided that the process is in `h1-1`. The altered version of SS-PETERSON2P is called S-PETERSON2P. The verification of S-PETERSON2P w.r.t. ME1 and ME2 can be done almost the same way as that of SS-PETERSON2P.

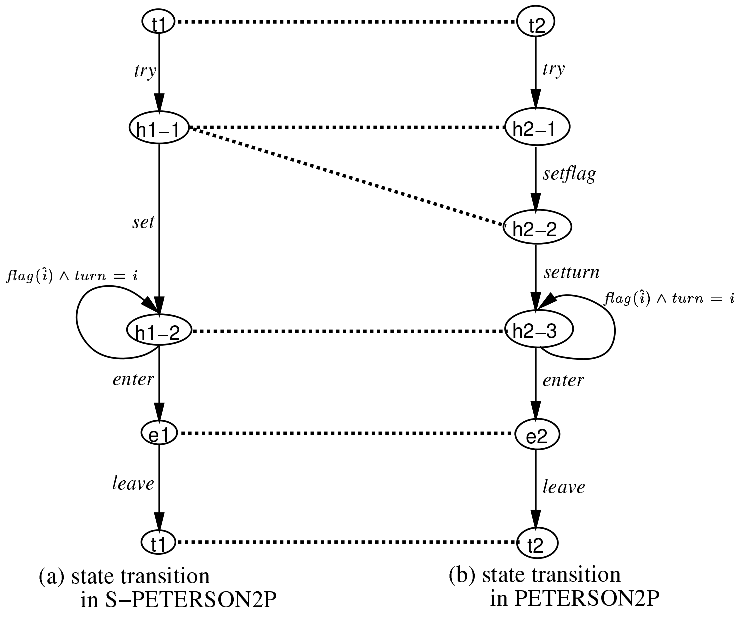

Fig. 1. Correspondance between states in S-PETERSON2P and PETERSON2P

Verification We first verify PETERSON2P w.r.t. **ME1** by showing there exists a simulation relations from PETERSON2P to S-PETERSON2P, and then verify PETERSON2P w.r.t. **ME2** using the simulation relation and UNITY logic. By showing there exists a simulation relation from PETERSON2P to S-PETERSON2P, we can say PETERSON2P also has the safety properties such as **ME1** satisfied by S-PETERSON2P [8]. We can prove PETERSON2P has liveness properties such as **ME2** using the fact that PETERSON2P satisfies the same safety properties and UNITY logic.

Proof sketch 1: It is necessary to classify actions into two types: external and internal actions. In S-PETERSON2P, `try`, `enter` and `leave` are external, and `set` is internal; in `PETERSON2P`, `try`, `enter` and `leave` are external, and `setflag` and `setturn` are internal.

First of all, a simulation function mapping each state of PETERSON2P to some state of S-PETERSON2P is defined so as to define a candidate for a simulation relatoin. The mapping from each process's state in PETERSON2P to some process's state in S-PETERSON2P is defined as shown in Fig. 1.

The following module SIMFUNC defines the simulation function `sim`.

```
mod SIMFUNC {
  pr( S-PETERSON2P + PETERSON2P )
  op sim : SState2 -> SState1
  var S : SState2
  var I : Nat
  ceq p(I, sim(S)) = t1   if p(I, S) == t2 .
  ceq p(I, sim(S)) = h1-1 if p(I, S) == h2-1 or p(I, S) == h2-2 .
  ceq p(I, sim(S)) = h1-2 if p(I, S) == h2-3 .
  ceq p(I, sim(S)) = e1   if p(I, S) == e2 .
  ceq flag(I, sim(S)) = true  if flag(I, S) and not reg(I, S) .
  ceq flag(I, sim(S)) = false if not flag(I, S) or reg(I, S) .
  eq  turn(sim(S)) = turn(S) .
}
```

A candidate for a simulation relation from PETERSON2P to S-PETERSON2P is defined. The following module SIMREL defines the candidate `R`.

```
mod SIMREL {
  pr( SIMFUNC )
  op _R_ : SState1 SState2 -> Bool
  var S1 : SState1
  var S2 : SState2
  eq S1 R S2 = p(0, S1) == p(0, sim(S2)) and p(1, S1) == p(1, sim(S2)) and
               flag(0, S1) == flag(0, sim(S2)) and flag(1, S1) == flag(1, sim(S2)) and
               turn(S1) == turn(sim(S2)) .
}
```

Then we prove the candidate is a simulation relation from PETERSON2P to S-PETERSON2P. For each initial state of PETERSON2P, we can easily find a corresponding initial state of S-PETERSON2P w.r.t. `R` because the two initial states are clearly under `R` if the two *turn*'s get set equal. The rest of the proof is done by a case analysis; there are 25 cases to be checked. For each case, in which the two states of S-PETERSON2P and PETERSON2P are under the candidate relation, we show there exists some action (possibly empty) sequence in S-PETERSON2P, corresponding to each action in PETERSON2P, s.t. the state after the action sequence in S-PETERSON2P and the state after the single action in PETERSON2P are still under the candidate relation, and the action sequence contains only (more than zero) internal actions, and moreover one external action that is the same as the single action provided that the single action is external. In this paper, we show only six cases plus one extra module in which the two states `s1` in S-PETERSON2P and `s2` in PETERSON2P are declared, which are under `R`. Each of the six cases is the state the process 1 in PETERSON2P has set *turn* to 1, and before it is yet to enter the critical section, i.e. its state is `h2-2`.

```
mod SIMREL-PROOF {
  pr( SIMREL )
  op s1 : -> SState1   op s2 : -> SState2
  eq p(0, s1) = p(0, sim(s2)) .  eq p(1, s1) = p(1, sim(s2)) .
  eq flag(0, s1) = flag(0, sim(s2)) .  eq flag(1, s1) = flag(1, sim(s2)) .
  eq turn(s1) = turn(sim(s2)) .
}
open SIMREL-PROOF
  eq p(0, s2) = t2 .  eq p(1, s2) = h2-3 .
  eq flag(0, s2) = false .  eq flag(1, s2) = true .
  eq reg(0, s2) = false .  eq reg(1, s2) = false .
  eq turn(s2) = 1 .
  red try(0, s1) R try(0, s2) and enter(1, s1) R enter(1, s2) .
close
open SIMREL-PROOF
  eq p(0, s2) = h2-1 .  eq p(1, s2) = h2-3 .
  eq flag(0, s2) = false .  eq flag(1, s2) = true .
  eq reg(0, s2) = false .  eq reg(1, s2) = false .
  eq turn(s2) = 1 .
  red s1 R setflag(0, s2) and enter(1, s1) R enter(1, s2) .
close
open SIMREL-PROOF
  eq p(0, s2) = h2-2 .  eq p(1, s2) = h2-3 .
  eq flag(0, s2) = true .  eq flag(1, s2) = true .
  eq reg(0, s2) = true .  eq reg(1, s2) = false .
  eq turn(s2) = 1 .
  red set(0, s1) R setturn(0, s2) and enter(1, s1) R enter(1, s2) .
close
open SIMREL-PROOF
  eq p(0, s2) = h2-3 .  eq p(1, s2) = h2-3 .
  eq flag(0, s2) = true .  eq flag(1, s2) = true .
  eq reg(0, s2) = false .  eq reg(1, s2) = false .
  eq turn(s2) = 0 .
  red enter(0, s1) R enter(0, s2) and enter(1, s1) R enter(1, s2) .
close
open SIMREL-PROOF
  eq p(0, s2) = h2-3 .  eq p(1, s2) = h2-3 .
  eq flag(0, s2) = true .  eq flag(1, s2) = true .
  eq reg(0, s2) = false .  eq reg(1, s2) = false .
  eq turn(s2) = 1 .
  red enter(0, s1) R enter(0, s2) and enter(1, s1) R enter(1, s2) .
close
open SIMREL-PROOF
  eq p(0, s2) = e2 .  eq p(1, s2) = h2-3 .
  eq flag(0, s2) = true .  eq flag(1, s2) = true .
  eq reg(0, s2) = false .  eq reg(1, s2) = false .
  eq turn(s2) = 1 .
  red leave(0, s1) R leave(0, s2) and enter(1, s1) R enter(1, s2) .
close
```

Now that we have shown there exists a simulation relation from PETERSON2P to S-PETERSON2P and the state `e2` in PETERSON2P is mapped to the state `e1` in S-PETERSON2P in the simulation relation, we have proven PETERSON2P has the safety property **ME1** that S-PETERSON2P satisfies as well. □

Proof sketch 2: We prove PETERSON2P satisfies the liveness property **ME2**, i.e. "p_i = h2-1 $\mapsto$ p_i = e2", where $i \in \{0,1\}$. Since it is easy to show "p_i = h2-1 $\mapsto$ p_i = h2-3" in PETERSON2P, it is sufficient to show "p_i = h2-3 $\mapsto$ p_i = e2" in PETERSON2P. In this paper, we show the case that $i = 0$, i.e. "p_0 = h2-3 $\mapsto$ p_0 = e2."

Suppose that the process 0 in PETERSON2P is in `h2-3`. If $\neg flag_1 \vee turn = 1$, it is easy to show "$p_0$ = h2-3 $\mapsto$ p_0 = e2." Thus suppose that $flag_1 \wedge turn = 1$. Since there is the simulation relation `R` from PETERSON2P and S-PETERSON2P, the process 0 must be in `h1-2` and $flag_1 \wedge turn = 1$ in the corresponding state in S-PETERSON2P. If so, the process 0 must be in `e1` in S-PETERSON2P. Since this is a safety property, so must the process 0 in PETERSON2P. It is easy to show that "p_1 = e2 $\mapsto$ p_1 = t2" in PETERSON2P. If p_1 in `e2` changes its state to `t2`, $\neg flag_1 \vee turn = 1$. Moreover even if p_1 executes any action sequence, $\neg flag_1 \vee turn = 1$ keeps holding unless p_0 in `h2-3` changes its state to `e2`. The following proof score proves this fact:

```
open PETERSON2P
  ops s s1 s2 s3 s4 s5 : -> SState2 .
  op test : SState2 -> Bool .
  eq p(0, s) = h2-3 .  eq p(1, s) = e2 .
  eq flag(0, s) = true .  eq flag(1, s) = true .
  eq reg(0, s) = false .  eq reg(1, s) = false .
  eq turn(s) = 1 .
  eq s1 = leave(1, s) .
  eq s2 = try(1, leave(1, s)) .
  eq s3 = setflag(1, try(1, leave(1, s))) .
  eq s4 = setturn(1, setflag(1, try(1, leave(1, s)))) .
  eq s5 = enter(1, setturn(1, setflag(1, try(1, leave(1, s))))) .
  eq test(S:SState2) = not flag(1, S) or turn(S) == 1 .
  red test(s1) and test(s2) and test(s3) and test(s4) and test(s5) .
close
```

Consequently we have shown "p_0 = h2-1 $\mapsto$ p_0 = e2." The case that $i = 1$ can be shown as the same way. □

5 *N*-Process Mutual Exclusion

In this section, Ticket and Anderson algorithms are treated.

5.1 Ticket Algorithm

Ticket algorithm is a mutual exclusion algorithm based on issuing tickets to a critical section. Some atomic operation has to be used to implement the algorithm. In the algorithm shown, `fetch&incmod` is used. It atomically reads a memory location, increments the value modulo N, writes the result into the memory location, and return the old value. It can be implemented (simulated) using simpler atomic operations such as `swap` or `ldstub` provided by SPARC architecture [12]. The algorithm in a traditional style is given below:

```
          ...
$ticket[i]$ := fetch&incmod($next$,$N$)
repeat while $ticket[i] \neq serve$
Critical Section
$ticket[i]$ := $N$
$serve$ := $serve + 1 \bmod N$
          ...
```

next and *serve* are shared variables of values in $\{0, \ldots, N-1\}$; initially both are 0, and *ticket*[*i*] is a local variable to a process whose ID is i in $\{0, \ldots, N-1\}$. *next* represents the next ticket to the critical section that is to be issued to a process, while *serve* represents the ticket whose owner is in the critical section or is allowed to enter it. When a process i tries to enter the critical section, it takes a ticket, that is, it indivisibly copies into its local variable *ticket*[*i*] and increments *next* modulo N using `fetch&incmod`. If a process's *ticket* is equal to *serve*, it enters the critical section. When a process leaves there, it increments *serve* modulo N.

Specification We describe the CafeOBJ specification of Ticket algorithm that is called TICKET. TICKET divides Ticket algorithm into three atomic actions: `try`, `enter`, and `leave` that correspond to the first assignment, the **repeat while** statement, and the last two assignments, respectively.

In TICKET, each process has three possible states: `t0`, `h0`, and `e0`. That a process is in `t0` means it is executing any code but Ticket algorithm, a process changes its state to `h0` whenever it executes `try` only if it is in `t0`, and that a process is in `e0` means it is executing the critical section.

Besides the three atomic actions, there are four observations: `p`, `ticket`, `next`, and `serve`. The main part of the signature of the specification is as follows:

```
-- initial state
  op init0 : -> SState0
-- observations
  bop p      : Nat SState0 -> PState0
  bop ticket : Nat SState0 -> Nat
  bop next   : SState0 -> Nat
  bop serve  : SState0 -> Nat
-- actions
  bop try   : Nat SState0 -> SState0
  bop enter : Nat SState0 -> SState0
  bop leave : Nat SState0 -> SState0
```

TICKET has four sets of equations: one for one initial state `init0`, and the others for the three states after a process has executed `try`, `enter`, and `leave`, respectively. We are giving the four sets of equations.

```
-- 0. in the initial state.
  eq  p(I, init0) = t0 .
  eq  ticket(I, init0) = N .
  eq  next(init0) = 0 .
  eq  serve(init0) = 0 .

-- 1 after execution of ‘try’
  ceq p(I, try(I, S)) = h0      if ticket(I, S) == N and p(I, S) == t0 .
  ceq p(J, try(I, S)) = p(J, S) if J =/= I or ticket(J, S) =/= N or p(J, S) =/= t0 .
  ceq ticket(I, try(I, S)) = next(S)      if ticket(I, S) == N and p(I, S) == t0 .
  ceq ticket(J, try(I, S)) = ticket(J, S) if J =/= I or ticket(J, S) =/= N or p(J, S) =/= t0 .
  ceq next(try(I, S)) = next(S) + 1 mod N if ticket(I, S) == N and p(I, S) == t0 .
  ceq next(try(I, S)) = next(S)           if ticket(I, S) =/= N or p(I, S) =/= t0 .
  eq  serve(try(I, S)) = serve(S) .

-- 2 after execution of ‘enter
  ceq p(I, enter(I, S)) = e0      if serve(S) == ticket(I, S) .
  ceq p(J, enter(I, S)) = p(J, S) if J =/= I or ticket(J, S) =/= serve(S) .
  eq  ticket(J, enter(I, S)) = ticket(J, S) .
  eq  next(enter(I, S)) = next(S) .
  eq  serve(enter(I, S)) = serve(S) .

-- 3 after execution of ‘leave
  ceq p(I, leave(I, S)) = t0      if p(I, S) == e0 .
  ceq p(J, leave(I, S)) = p(J, S) if J =/= I or p(J, S) =/= e0 .
  ceq ticket(I, leave(I, S)) = N           if p(I, S) == e0 .
  ceq ticket(J, leave(I, S)) = ticket(J, S) if J =/= I or p(J, S) =/= e0 .
  eq  next(leave(I, S)) = next(S) .
  ceq serve(leave(I, S)) = serve(S) + 1 mod N if p(I, S) == e0 .
  ceq serve(leave(I, S)) = serve(S)           if p(I, S) =/= e0 .
```

Verification We verify Ticket algorithm specified in CafeOBJ w.r.t. **ME1** and **ME2**. The two properties are formally restated:

1. $\textit{invariant}\ p_i = \mathrm{e0} \wedge p_j = \mathrm{e0} \Rightarrow i = j$,
2. $p_i = \mathrm{h0} \mapsto p_i = \mathrm{e0}$.

Proof sketch 1: The following two subproperties are easily checked to hold:

i. $\textit{invariant}\ p_i = \mathrm{e0} \Rightarrow \textit{ticket}[i] = \textit{serve}$,
ii. $\textit{invariant}\ \textit{ticket}[i] = \textit{ticket}[j] \Rightarrow i = j \vee \textit{ticket}[i] = N$.

From the first subproperty (in the following the term *invariant* is omitted), $p_i = \mathrm{e0} \wedge p_j = \mathrm{e0} \Rightarrow \textit{ticket}[i] = \textit{serve} \wedge \textit{ticket}[j] = \textit{serve}$. From this and $\textit{serve} < N$, $p_i = \mathrm{e0} \wedge p_j = \mathrm{e0} \Rightarrow (\textit{ticket}[i] = \textit{ticket}[j]) \wedge \textit{ticket}[i] < N$. From this and the second subproperty, $p_i = \mathrm{e0} \wedge p_j = \mathrm{e0} \Rightarrow i = j$. □

Proof sketch 2: Suppose that the following two subproperties hold in TICKET:

i. $(\langle \exists j :: p_j = \mathrm{e0}\rangle \wedge p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \wedge (\mathit{ticket}[i] - \mathit{serve} \bmod N) = k)$ *ensures* $(p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \wedge (\mathit{ticket}[i] - \mathit{serve} \bmod N) < k)$,

ii. $(\langle \forall j :: p_j \neq \mathrm{e0}\rangle \wedge p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \wedge (\mathit{ticket}[i] - \mathit{serve} \bmod N) = k)$ *ensures* $((p_i = \mathrm{h0} \wedge \mathit{ticket}[i] = \mathit{serve}) \vee (\langle \exists j :: p_j = \mathrm{e0}\rangle \wedge p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \wedge (\mathit{ticket}[i] - \mathit{serve} \bmod N) = k))$.

From the two subproperties using Basis inference rule for *leads-to*, Cancellation Theorem and Finite Disjunction, the following liveness property is obtained:

$$(p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \wedge (\mathit{ticket}[i] - \mathit{serve} \bmod N) = k) \mapsto ((p_i = \mathrm{h0} \wedge \mathit{ticket}[i] = \mathit{serve}) \vee (p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \wedge (\mathit{ticket}[i] - \mathit{serve} \bmod N) < k)) .$$

Then, by applying Induction for Leads-to (see Appendix) on k to this, the following is obtained: $p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \mapsto p_i = \mathrm{h0} \wedge \mathit{ticket}[i] = \mathit{serve}$. We can also easily show the following: $p_i = \mathrm{h0} \wedge \mathit{ticket}[i] = \mathit{serve}$ *ensures* $p_i = \mathrm{e0}$. From these two subproperties using Basis and Transitivity inference rules for *leads-to*, the following is obtained: $p_i = \mathrm{h0} \wedge \mathit{ticket}[i] \neq N \mapsto p_i = \mathrm{e0}$. Besides, "$p_i = \mathrm{h0} \Rightarrow \mathit{ticket}[i] \neq N$" holds clearly. Therefore, from these two subproperties using Implication Theorem (see Appendix) and Transitivity inference rule for *leads-to*, the desired property "$p_i = \mathrm{h0} \mapsto p_i = \mathrm{e0}$" is derived. □

We prove the assumed subproperties.

Proof sketch 2.1: Even if processes in `t0` execute any action, they do not change any *ticket* of other processes (in `h0`) and *serve*. Moreover, even if processes in `h0` execute any action in the state in which there is a process in `e0`, nothing changes. Therefore, only execution of `leave` by the process in `e0` needs considering because execution of every action, but `leave`, by the process in `e0` does not change the system state. Thus, the subproperty can be shown by executing the following proof score.

```
open TICKET
  op s : -> SState0 .
  ops i, j : -> Nat .
  op k : -> NzNat .
  eq p(i,s) = h0 .
  eq ticket(i,s) - serve(s) mod N = k .
  eq (ticket(i,s) - (serve(s) + 1 mod N) mod N < k) = true .
  eq (ticket(i,s) == N) = false .
  eq p(j,s) = e0 .
  eq ticket(j,s) = serve(s) .
  red p(i,leave(j,s)) == h0 and ticket(i,leave(j,s)) == ticket(i,s) and
      ticket(i,leave(j,s)) - serve(leave(j,s)) mod N < k .
close
```

□

Proof sketch 2.2: By the same discussion as in the proof of the first subproperty, all we have to do is to consider a process in `h0`, say p_j, whose *ticket* is equal to *serve* executes `enter`. If $k = 0$, the subproperty clearly holds because $\mathit{ticket}[i] = \mathit{serve}$. Hence, we consider only the case $k > 0$. The subproperty can be shown by executing the following proof score.

```
open TICKET
  op s : -> SState0 .
  ops i, j : -> Nat .
  op k : -> NzNat .
  eq p(i,s) = h0 .
  eq ticket(i,s) - serve(s) mod N = k .
  eq (ticket(i,s) == N) = false .
  eq p(j,s) = h0 .
  eq ticket(j,s) = serve(s)  .
  red p(j,enter(j,s)) == e0 and p(i,enter(j,s)) == h0 and
      not (ticket(i,enter(j,s)) == N) and (ticket(i,s) - serve(s) mod N) == k .
close
```

□

5.2 Anderson Algorithm

Anderson algorithm [1] is an array-based queuing mutual exclusion algorithm. It may be regarded as an improvement of Ticket algorithm. In Ticket algorithm, waiting processes have to repeatedly check if they are allowed to enter their critical section by accessing the same location, i.e. the shared variable *serve*. That might cause network or bus traffic high, declining overall performance. In Anderson algorithm, each process is waiting on a different location, in a different cache line, if some process is in their critical section. The algorithm in a traditional style is given below:

```
              ...
$place[i]$ := fetch&incmod($next$,$N$)
repeat while $\neg array[place[i]]$
Critical Section
$array[place[i]]$ := false
$array[place[i] + 1 \bmod N]$ := true
$place[i]$ := $N$
              ...
```

next is a shared variable of values in $\{0, \ldots, N-1\}$, initially 0, and *array* is a shared array of values in $\{0, \ldots, N-1\}$, whose size is N, initially *array*$[0]$ = true and *array*$[i]$ = false if $i \neq 0$. *place*$[i]$ is a local variable of values in $\{0, \ldots, N\}$ to a process whose ID is i in $\{0, \ldots, N-1\}$, initially N.

Specification We describe the CafeOBJ specification of Anderson algorithm that is called ANDERSON. ANDERSON divides Ticket algorithm into three atomic actions: `try`, `enter`, and `leave` that correspond to the first assignment, the **repeat while** statement, and the last three assignments, respectively.

In ANDERSON, each process has three possible states: `t1`, `h1`, and `e1`. That a process is in `t1` means it is executing any code but Anderson algorithm, a process changes its state to `h1` whenever it executes `try` only if it is in `t1`, and that a process is in `e1` means it is executing the critical section.

Besides the three atomic actions, there are four observations: `p`, `place`, `next`, and `array`. The main part of the signature of the specification is as follows:

```
-- initial state
  op init1 : -> SState1
-- observations
  bop p : Nat SState1 -> PState1
  bop place : Nat SState1 -> Nat
  bop next : SState1 -> Nat
  bop array : Nat SState1 -> Bool
-- actions
  bop try : Nat SState1 -> SState1
  bop enter : Nat SState1 -> SState1
  bop leave : Nat SState1 -> SState1
```

ANDERSON has four sets of equations: one for one initial state `init1`, and the others for the three states after a process has executed `try`, `enter`, and `leave`, respectively. We are giving the three sets of equations.

```
-- 0. in the initial state.
  eq  p(I, init1) = t1 .
  eq  place(I, init1) = N .
  eq  next(init1) = 0 .
  eq  array(0, init1) = true .
  ceq array(Idx, init1) = false if Idx =/= 0 .

-- 1 after execution of 'try'
  ceq p(I, try(I, S)) = h1  if place(I, S) == N and p(I, S) == t1 .
  ceq p(J, try(I, S)) = p(J, S) if J =/= I or place(J, S) =/= N or p(J, S) =/= t1 .
  ceq place(I, try(I, S)) = next(S)     if place(I, S) == N and p(I, S) == t1 .
  ceq place(J, try(I, S)) = place(J, S) if J =/= I or place(J, S) =/= N or p(J, S) =/= t1 .
  ceq next(try(I, S)) = next(S) + 1 mod N if place(I, S) == N and p(I, S) == t1 .
  ceq next(try(I, S)) = next(S)           if place(J, S) =/= N or p(J, S) =/= t1 .
  eq  array(Idx, try(I, S)) = array(Idx, S) .

-- 2 after execution of 'enter'
  ceq p(I, enter(I, S)) = e1 if array(place(I, S), S) .
  ceq p(J, enter(I, S)) = p(J, S)   if J =/= I or not array(place(I, S), S) .
  eq  place(J, enter(I, S)) = place(J, S) .
  eq  next(enter(I, S)) = next(S) .
  eq  array(Idx, enter(I, S)) = array(Idx, S) .
```

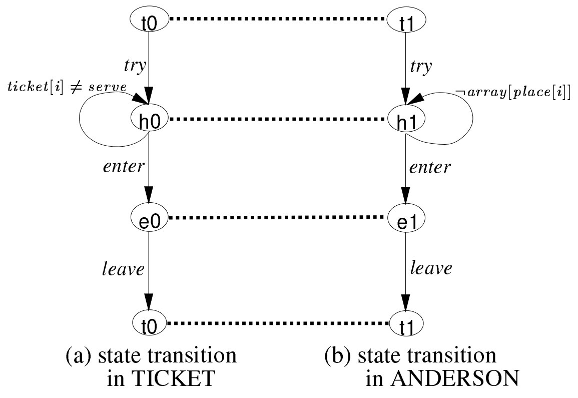

Fig. 2. Correspondance between states in TICKET and ANDERSON

```
-- 3 after execution of ‘leave’
  ceq p(I, leave(I, S)) = t1 if p(I, S) == e1 .
  ceq p(J, leave(I, S)) = p(J, S)  if J =/= I or p(J, S) =/= e1 .
  ceq place(I, leave(I, S)) = N          if p(I, S) == e1 .
  ceq place(J, leave(I, S)) = place(J, S) if J =/= I or p(J, S) =/= e1 .
  eq  next(leave(I, S)) = next(S) .
  ceq array(Idx, leave(I, S)) = false         if Idx == place(I, S) .
  ceq array(Idx, leave(I, S)) = true          if Idx == place(I, S) + 1 mod N .
  ceq array(Idx, leave(I, S)) = array(Idx, S) if Idx =/= place(I, S) and Idx =/= place(I, S) + 1 mod N .
```

Verification We verify Anderson algorithm specified in CafeOBJ w.r.t. the two properties as the same way in the verification of PETERSON2P by showing there exists a simulation relation from ANDERSON to TICKET.

Proof sketch 1: First of all, a simulation function mapping from each state of ANDERSON to some state of TICKET is defined. The mapping from each process's state in ANDERSON to some process's state in TICKET is defined as shown in Fig. 2. The following module SIMFUNC defines the simulation function `sim`.

```
mod SIMFUNC {
  pr( TICKET * {op N -> N0} + ANDERSON * {op N -> N1} )
  op sim : SState1 -> SState0
  op N : -> Nat  var S : SState1  var I : Nat
  ceq p(I, sim(S)) = t0 if p(I, S) == t1 .
  ceq p(I, sim(S)) = h0 if p(I, S) == h1 .
  ceq p(I, sim(S)) = e0 if p(I, S) == e1 .
  eq  ticket(I, sim(S)) = place(I, S) .
  eq  next(sim(S)) = next(S) .
  ceq (I == serve(sim(S))) = true  if array(I, S) .
  ceq (I == serve(sim(S))) = false if not array(I, S) .
  eq  N0 = N .  eq  N1 = N .
}
```

In the definition, the fact "*invariant* $array[i] \wedge array[j] \Rightarrow i = j$" is used. Next a candidate for a simulation relation from ANDERSON to TICKET is defined. The following module SIMREL defines the candidate `R`.

```
mod SIMREL {
  pr( SIMFUNC )
  op _R[_]_ : SState0 Nat SState1 -> Bool
  var S0 : SState0  var S1 : SState1
  var I : Nat
  eq S0 R[I] S1 = p(I, S0) == p(I, sim(S1)) and ticket(I, S0) == ticket(I, sim(S1)) and
                  next(S0) == next(sim(S1)) and serve(S0) == serve(sim(S1)) .
}
```

We prove `R` a simulation relation from ANDERSON to TICKET by a case analysis. It is divided into three cases: (1) Every process is in `t1`; (2) Some processes are in `h1`, and the rest are in `t0`;

(3) A process is in `e0`. In this paper, only the case (2) is handled. In the following proof score, let the process i be in `h1` s.t. $\mathit{array}[\mathit{place}[i]]$, the process j be in `h1` s.t. $\neg \mathit{array}[\mathit{place}[j]]$, and the process k be an arbitrary process in `h1`. Besides the processes l and m are arbitrary processes in `t1`. The execution of the following proof score can show, for each action of ANDERSON in the system state of the case (2), there exists some action sequence of actions in `TICKET`.

```
mod SIMREL-PROOF {
  pr( SIMREL )
  op s0 : -> SState0  op s1 : -> SState1
  var I : Nat
  eq p(I, s0) = p(I, sim(s1)) .
  eq ticket(I, s0) = place(I, s1) .
  eq next(s0) = next(s1) .
  eq serve(s0) = serve(sim(s1)) .
  eq array(serve(sim(s1)),s1) = true .
}
open SIMREL-PROOF
  ops i j k l m : -> Nat .
  eq p(i, s1) = h1 .  eq p(j, s1) = h1 .  eq p(k, s1) = h1 .
  eq p(l, s1) = t1 .  eq p(m, s1) = t1 .
  eq place(l, s1) = N .  eq place(m, s1) = N .
  eq place(i, s1) = serve(sim(s1)) .
  eq array(serve(sim(s1)), s1) = true .
  eq array(place(j,s1), s1) = false .
  red try(l, s0) R[l] try(l, s1) and try(l, s0) R[m] try(l, s1) and try(l, s0) R[k] try(l, s1) .
  red enter(i, s0) R[i] enter(i, s1) and enter(i, s0) R[j] enter(i, s1)  and enter(i, s0) R[l] enter(i, s1) .
  red enter(j, s0) R[j] enter(j, s1) and enter(j, s0) R[k] enter(j, s1) enter(j, s0) R[l] enter(j, s1) .
close
```

The other two cases can be treated the same way as the case (2). Since we have shown there exists a simulation relation from ANDERSON to TICKET, it has been shown that ANDERSON has the safety properties such as **ME1** that TICKET satisfies. □

Proof sketch 2: Let a process i be in `h1` in ANDERSON. If $\mathit{array}[\mathit{place}[i]]$, it is easy to show "$p_i = \mathrm{h1} \mapsto p_i = \mathrm{e1}$." Thus suppose that $\neg \mathit{array}[\mathit{place}[i]]$. If so, there must be a process i in `h0` s.t. $\mathit{ticket}[i] \neq \mathit{serve}$ in TICKET simulatiing ANDERSON. Moreover there must be a process j s.t. $\mathit{ticket}[j] = \mathit{serve}$ in TICKET, and then there must be a process j s.t. $\mathit{array}[\mathit{place}[j]]$ in ANDERSON. The process j in ANDERSON eventually reaches the state `t1`, when $\mathit{array}[\mathit{place}[j]]$ and $\mathit{array}[\mathit{place}[j] + 1 \bmod N]$ become false and true, respectively. Thta is, the distance between $\mathit{place}[i]$ and pos (s.t. $\mathit{array}[\mathit{pos}]$) decrements. By applying Induciton for *leads-to* to this, we can derive the desired liveness property "$p_i = \mathrm{h1} \mapsto p_i = \mathrm{e1}$." □

6 Conclusion

We have adopted UNITY computational model to specify mutual exclusion algorithms in CafeOBJ, and used UNITY logic and simulation relations to verify a safety and a liveness properties of mutual exclusion algorithms with CafeOBJ.

Throughout the experience that we have specified and verified some mutual exclusion algorithms with CafeOBJ, we have had the following impressions to CafeOBJ:

- CafeOBJ's notational machinery is suitable for specifying parallel computational models such as UNITY computational model.
- CafeOBJ reasonably supports verification of safety properties and *ensures* liveness properties of parallel systems; something is needed so that CafeOBJ can support verification of general liveness properties; in this paper, general liveness properties have been proven by hand.
- It might not be so easy to specify liveness properties in pure equations; in this paper, liveness properties have been described in UNITY logic; some logical system, e.g. temporal logic, might strengthen CafeOBJ, allowing CafeOBJ to specify and verify parallel (or distributed) algorithms and/or systems more suitably.

References

1. Anderson, T. E.: The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Trans. Parall. Dist. Syst.* **1** (1). (1990) 6–16

2. Chandy, K. M. and Misra, J.: *Parallel Program Design: A Foundation.* Addison-Wesley. 1988
3. Diaconescu, R. and Futatsugi, K.: *CafeOBJ Report. AMAST Series in Computing* **6**. World Scientific. 1998
4. Floyd, R.: Assigning Meanings to Programs. Proc. of Symposia Applied Mathematics **19**. (1967) 19–32
5. Futatsugi, K. and Nakagawa, A.: An Overview of CAFE Specification Environment – an algebraic approach for creating, verifying, and maintaining formal specification over networks –. Proc. of First IEEE Int'l. Conf. on Formal Engineering Methods. (1997) 170–181
6. Goguen, J. and Malcolm, G.: A Hidden Agenda. Technical Report CS97-538. Univ. of California at San Diego. 1997
7. Hoare, C. A. R.: An Axiomatic Basis for Computer Programming. *Communication of the ACM* **12** (10). (1969) 576–580
8. Lynch, N. A.: *Distributed Algorithm.* Morgan-Kaufmann. 1996
9. Nakagawa, A. T., Sawada, T. and Futatsugi, K: CafeOBJ User's Manual – ver.1.3 –. 1997. Available at http://caraway.jaist.ac.jp/cafeobj/
10. Peterson, G. L.: Myths about the Mutual Exclusion Problem. *Information Processing Letters* **12** (3). (1981) 115–116
11. Pnueli, A.: The Temporal Semantics of Concurrent Programs. *Theoretical Computer Science* **13**. North-Holland. (1981) 45–60
12. SPARC Int'l.: *The SPARC Architecture Manual.* Prentice Hall. 1992.

A Some Theorems of UNITY Logic

Implication Theorem:

$$\frac{p \Rightarrow q}{p \mapsto q}.$$

Finite Disjunction:

$$\frac{p \mapsto q,\ p' \mapsto q'}{p \vee p' \mapsto q \vee q'}.$$

Cancellation Theorem:

$$\frac{p \mapsto q \vee b,\ b \mapsto r}{p \mapsto q \vee r}.$$

Induction for Leads-to: Let W be a set well-founded under the raltion $\prec$, and let M be a function, also called a metric, from program states to W.

$$\frac{\langle \forall m : m \in W :: p \wedge M = m \mapsto (p \wedge M \prec m) \vee q \rangle}{p \mapsto q}.$$

Corollary of Induction for Leads-to:

$$\frac{\langle \forall m : m \in W :: p \wedge M = m \mapsto M \prec m \rangle}{\text{true} \mapsto \neg p}.$$