Formal Analysis of the Bakery Protocol with Consideration of Nonatomic Reads and Writes

Kazuhiro Ogata and Kokichi Futatsugi

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

Abstract. The bakery protocol is the first real solution of the mutual exclusion problem. It does not assume any lower mutual exclusion protocols. The bakery protocol has been often used as a benchmark to demonstrate that proposed verification methods and/or tools are powerful enough. But, the true bakery protocol has been rarely used. We have formally proved that the protocol satisfies the mutual exclusion property. The proof is mechanized with CafeOBJ, an algebraic specification language, in which state machines as well as data types can be specified. Nonatomic reads and writes to shared variables are formalized by representing an assignment to a shared variable with multiple atomic transitions. Our formal model of the protocol has states in which a shared variable is being modified. A read to the variable in such states obtains an arbitrary value, which is represented as a CafeOBJ term.

Keywords: CafeOBJ, invariant property, mutual exclusion, observational transition system (OTS), verification.

1 Introduction

The mutual exclusion problem is one of the classic but still important problems in computer science. The problem was originally raised and solved by Edsger Dijkstra in 1965 [1]. Many solutions have been proposed since then. But, it took nine years to solve the problem in the true sense of the word. The first real solution is the bakery protocol proposed by Leslie Lamport in 1974 [2]. All mutual exclusion protocols before the bakery protocol assume lower mutual exclusion protocols. On the other hand, the bakery protocol does not.

The bakery protocol has been often used as a benchmark to demonstrate that proposed verification methods and/or tools, among which are [3,4,5], are powerful enough. But, the true bakery protocol has been rarely used. A simplified version of the bakery protocol has been often used such that the simplified version assumes that a read and a write to a shared variable are performed exclusively.

Lamport gives an informal proof [2] that the protocol satisfies some properties and more rigorous proofs [6,7,8] that the protocol satisfies the mutual exclusion property. The proofs do not assume any atomicity. But, they do not seem to have been mechanized.

S. Liu, T. Maibaum, and K. Araki (Eds.): ICFEM 2008, LNCS 5256, pp. 187–207, 2008.

We describe a fully formal and mechanized proof that the protocol satisfies the mutual exclusion protocol. We have faithfully made an abstract model of the protocol as much as the formal method used can. Our abstract model respects nonatomic reads and writes to shared variables, namely that those reads and writes can overlap. Nonatomic reads and writes to shared variables are formalized by representing an assignment to a shared variable with multiple atomic transitions. Our abstract model has states in which a shared variable is being modified. A read to the variable in such states obtains an arbitrary value. Our abstract model uses natural numbers, while the bakery protocol uses integers. But, we do not think that this difference is major because sequences of bits can be naturally interpreted as unsigned integers, or natural numbers. We have formally proved that our abstract model satisfies the mutual exclusion property.

The formal method used is the OTS/CafeOBJ method [9,10]. Observational transition systems (OTSs) are used as abstract models, CafeOBJ [11], an algebraic specification language, is used to specify OTSs and properties to verify, and the CafeOBJ system is used as an interactive proof assistant. We also describe some specification and verification techniques used in the case study, which can be used for not only the OTS/CafeOBJ method but also other formal methods based on algebraic specifications.

The rest of the paper is organized as follows. Section 2 describes the bakery protocol. Sections 3 and 4 introduce CafeOBJ and OTSs, respectively. Section 5 describes how to model the bakery protocol as an OTS, which is specified in CafeOBJ. Section 5 describes the verification. Section 7 discusses some issues on specification and verification. Section 8 mentions some related work. Section 9 concludes the paper.

2 The Bakery Protocol

Many existing mutual exclusion protocols assume atomic instructions, which can be considered lower mutual exclusion protocols. Some assume that read (or load) and write (or store) instructions are atomic [1,12], which implies that a read and a write to a shared variable are performed exclusively. Some assume more complex atomic instructions such as test_and_set and fetch_and_store instructions [13,14]. Unlike those protocols, the bakery protocol does not assume any lower mutual exclusion protocols.

The bakery protocol is an N-process mutual exclusion protocol. The N natural numbers $1, \ldots, N$ are used as the identifications of the N processes, respectively. Fig. 1 shows the protocol in the ALGOL style for each process i. The type of each variable used is integer. $choosing[i]$ and $number[i]$ are shared variables among the N processes. While all processes read the variables, only the process i writes them. j is a local variable to the process i.

The function *maximum* takes N integers and returns one of them, which is not less than the others. The N arguments can be read in any order. $(a, b) < (c, d)$ equals $a < c \vee (a = b \wedge b < d)$.

```
begin integer j;
  L1: choosing[i] := 1;
      number[i] := 1 + maximum(number[1], ..., number[N]);
      choosing[i] := 0;
      for j = 1 step 1 until N do
        begin
          L2: if choosing[j] ≠ 0 then goto L2;
          L3: if number[j] ≠ 0 and (number[j], j) < (number[i], i)
                then goto L3;
        end;
      critical section;
      number[i] := 0;
      noncritical section;
      goto L1;
end
```

Fig. 1. The bakery protocol in the ALGOL style

Initially, the process i is in the noncritical section, both *choosing*$[i]$ and *number*$[i]$ are set to 0, and j is an arbitrary integer. We suppose that if the process i enters the critical section, it eventually leaves there, and the process i does not write both *choosing*$[i]$ and *number*$[i]$ in both the critical and noncritical sections.

One of the desired properties the bakery protocol should satisfy is the mutual exclusion property, which means that there exists at most one process in the critical section at any given moment.

3 CafeOBJ

CafeOBJ is an algebraic specification language mainly based on *order-sorted algebras* and *hidden algebras*. Data are specified in terms of the former and state machines are specified in terms of the latter. CafeOBJ has two kinds of sorts: *visible* and *hidden sorts*. Visible and hidden sorts denote carrier sets of order-sorted algebras and hidden algebras, respectively. Elements of visible and hidden sorts are data values and states of state machines, respectively.

There are two kinds of operators: *conventional* and *behavioral operators*. The former are used as data constructors and functions on data, and the latter are used for state machines. The former are also used to represent states of state machines. The latter are classified into *observations* and *actions*. Observations obtain data values that characterize states, and actions change states. A conventional operator f, an observation o and an action a are declared as "`op` f `:` VL `->` S", "`bop` o `:` H VL `->` V" and "`bop` a `:` H VL `->` H", respectively, where VL is a list of visible sorts, S is a visible or hidden sort, V is a visible sort and H is a hidden sort. Conventional operators with no arguments are called constants. An operator can be given attributes such as `assoc`, `comm` and `id:` t, which specify that the operator is associative and commutative, and a term t is an identity of the operator.

There are two kinds of equations: *conventional* and *behavioral equations*. Both can have conditions. A conventional equation says that two data values are equal, and a behavioral equation says that two states are equal in that any observation returns a same data value in the two states and any sequence of actions preserves it. A conventional equation and a behavioral equation are declared as "[c]`eq` l_v `=` r_v [`if` c] ." and "[c]`beq` l_h `=` r_h [`if` c] .", respectively, where l_v and r_v are terms whose sorts are visible, l_h and r_h are terms whose sorts are hidden and c is a term whose sort is `Bool` (see below).

Basic units of CafeOBJ specifications are modules. CafeOBJ provides built-in modules. One of the most important built-in modules is `BOOL` in which propositional logic is specified. `BOOL` is automatically imported by almost every module unless otherwise stated. In `BOOL` and its parent modules, declared are the visible sort `Bool` denoting the set of Boolean values, the constants `true` and `false` of `Bool`, and operators denoting some basic logical connectives. Among the operators are `not_`, `_and_`, `_or_`, `_xor_`, `_implies_` and `_iff_` denoting negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), exclusive disjunction (xor), implication ($\Rightarrow$) and logical equivalence ($\Leftrightarrow$), respectively. An underscore `_` indicates the place where an argument is put such as "b_1 `and` b_2". The operators `_and_`, `_or_` and `_xor_` are given `assoc` and `comm`. The operator `if_then_else_fi` corresponding to the **if** construct in programming languages is also declared. CafeOBJ uses the Hsiang term rewriting system [15] as the decision procedure for propositional logic, which is implemented in `BOOL`. CafeOBJ reduces any term denoting a proposition that is always true (false) to `true` (`false`). More generally, a term denoting a proposition reduces to an exclusively disjunctive normal form of the proposition.

4 Observational Transition Systems (OTSs)

We suppose that there exists a universal state space and each data type used in OTSs is provided. The state space is represented by a hidden sort H and data types are represented by visible sorts such as V_{o1}.

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

- $\mathcal{O}$: A finite set of observers. Each *observer* is represented as an observation declared as "`bop` o `:` H V_{o1} $\ldots V_{om}$ `->` V_o". The equivalence relation ($s_1 =_{\mathcal{S}} s_2$) between two states $s_1, s_2 : H$ is defined as $o(s_1, x_1, \ldots, x_m) = o(s_2, x_1, \ldots, x_m)$ for all $o \in \mathcal{O}$ and all $x_i : V_{oi}$ for $i = 1, \ldots, m$. Observers correspond to variables in the conventional definitions of transition systems.
- $\mathcal{I}$: The set of initial states. An arbitrary initial state is represented by a constant *init* declared as "`op` *init* `: ->` H". Let X_i be a CafeOBJ variable of V_{oi}. The constant *init* is defined with a set of equations. For each observer o, the definition has the equation "`eq` $o(init, X_1, \ldots, X_m)$ `=` *initVal* .", where *initVal* is a term denoting a data value returned by the observer in an arbitrary initial state.
- $\mathcal{T}$: A finite set of transitions. Each *transition* is represented as an action declared as "`bop` t `:` H V_{t1} $\ldots V_{tn}$ `->` H". Each transition preserves

the equivalence relation $=_{\mathcal{S}}$. Each transition t has the condition $c\text{-}t$. If $c\text{-}t(s, y_1, \ldots, y_n)$ does not hold, then $t(s, y_1, \ldots, y_n) =_{\mathcal{S}} s$ for all $s : H$ and all y_j for $j = 1, \ldots, n$. Each transition t is defined with a set of equations. Let S be a CafeOBJ variable of H and Y_j be a CafeOBJ variable of V_{tj}. For each observer o, the definition has the equation "`ceq` $o(t(S, Y_1, \ldots, Y_n), X_1 \ldots, X_m)$ `=` *newVal* `if` $c\text{-}t(S, Y_1, \ldots, Y_n)$ `.`", where *newVal* is a term denoting the data value returned by the observer in the successor state $t(S, Y_1, \ldots, Y_n)$ if the effective condition holds. If t does not change the value returned by o, we may omit the condition and use a nonconditional equation. The definition has one more equation "`cbeq` $t(S, Y_1, \ldots, Y_n)$ `=` S `if not` $c\text{-}t(S, Y_1, \ldots, Y_n)$ `.`", which says that the transition does not change states essentially if the effective condition does not hold.

Given an OTS $\mathcal{S}$, a transition $t \in \mathcal{T}$ and two states $s, s' : H$, if $t(s, b_1, \ldots, b_n) =_{\mathcal{S}} s'$ for some values $b_1 : V_{t1}, \ldots, b_n : V_{tn}$, then we write $s \rightsquigarrow^t_{\mathcal{S}} s'$ and call s' a *t-successor state* of s with respect to (wrt) $\mathcal{S}$. t may be omitted from $s \rightsquigarrow^t_{\mathcal{S}} s'$ and s' may be called a successor state of s wrt $\mathcal{S}$.

Given an OTS $\mathcal{S}$, *reachable states* wrt $\mathcal{S}$ are inductively defined:

- Each $s_0 \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.
- For each $s, s' : H$ such that $s \rightsquigarrow_{\mathcal{S}} s'$, if s is reachable wrt $\mathcal{S}$, so is s'.

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

Operators whose ranks (types) are H `->` `Bool` are called *state predicates*. Any state predicate p `:` H `->` `Bool` is called *invariant* wrt $\mathcal{S}$ if p holds in all reachable states wrt $\mathcal{S}$, i.e. $\forall s : \mathcal{R}_{\mathcal{S}}. p(s)$.

5 Model and Specification of the Bakery Protocol

We describe the OTS $\mathcal{S}_{\text{Bakery}}$ modeling the bakery protocol, which is specified in CafeOBJ.

5.1 Data Used

Five kinds of data are used in $\mathcal{S}_{\text{Bakery}}$: (1) Boolean values, (2) natural numbers, (3) pairs of natural numbers, (4) sets of natural numbers, and (5) labels.

The built-in module `BOOL` is used for Boolean values. As described in Sect. 3, a Boolean term reduces to an exclusively disjunctive normal form. This is useful because this can check if a given Boolean term is always true (or false). But, it may take too much time for Boolean terms to reduce to their normal forms if the terms have many disjunctions. In addition to `_or_`, `BOOL` has the declaration of another operator `_or-else_` for disjunction. The use of `_or-else_`, instead of `_or_`, can prevent Boolean terms from fully reducing to their exclusive-or normal forms and can save much time. The attributes `assoc` and `comm` are not given to `_or-else_` in `BOOL`. For convenience, those attributes are given to the operator.

The sort `Nat` represents the set of all natural numbers. The constant `0` and the operator `s` denote zero and the successor function. We have the operators

`_=_`, `_<_` and `max`. The first two are the equivalence predicate and the less-than predicate. The third one takes two natural number and returns one that is not less than the other. The attribute `comm` is given to `_=_`.

We have the constant `numOfProcs` of `Nat`, which is the number of processes participating in the protocol. `numOfProcs` corresponds to N in Fig. 1. The predicate `isPid` checks if a given natural number p is used as a process identification, namely that p is greater than 0 and less than or equal to N represented by `numOfProcs`.

We also have the operator `next`. Given a natural number x, the term `next(`x`)` denotes an arbitrary natural number. The operator is used to model an assignment to a shared variable.

The sort `NatPair` represents the set of all pairs of natural numbers. The operator `<_,_>` is the constructor of pairs of natural numbers. We have the operators `_=_` and `_<_`. `<` a`,`b `> = <` c`,`d `>` equals a `=` b `and` b `=` d, and `<` a`,`b `> < <` c`,`d `>` equals a `<` c `or (`a `=` c `and` b `<` d`)`. The attribute `comm` is given to `_=_`.

The sort `NatSet` represents the set of all sets of natural numbers. `Nat` is declared as a sub-sort of `NatSet`, which specifies that a natural number is the singleton set that contains the number. The constant `empty` denotes the empty set and the juxtaposition operator `__` is the constructor of nonempty sets. The attributes `assoc`, `comm` and `id: empty` are given to the juxtaposition operator. We have the operators `_\in_`, `del` and `empty?`. The first checks if a given element is in a given set, the second deletes a given element from a given set if any, and the third checks if a given set is empty. We also have the operator `mkSet`, which takes a natural number n and returns `empty` if n is zero and the term denoting $\{1, \ldots, n\}$ if n is greater than zero.

Labels are used to indicate which parts of the protocol processes are going to execute next. The sort `Label` represents the set of all labels. There are 17 labels, which are represented by the 17 constants `l1`, `l2`, ..., `l15`, `cs`, and `ncs`. We have the operator `_=_`, which is the equivalence predicate on labels. The attribute `comm` is given to `_=_`.

5.2 Definition of Equivalence Predicate on Labels

The operator `_=_` is defined with a set of equations. One equation is "`eq (L = L) = true .`", where `L` is a CafeOBJ variable of `Label`. Given two different labels l_1 and l_2 such as `cs` and `ncs`, we want l_1 `=` l_2 to reduce to `false`. Given one label l and a term x whose final value has not been determined, however, we do not want l `=` x to reduce to either `true` or `false` because x may or may not equal l. One way to define the operator to fulfill the requirement is to declare the equation "`eq (`l_1 `=` l_2`) = false .`" for each pair l_1, l_2 of different labels. But, we need to declare many such equations.

Another solution [16], which does not require to declare many equations, is as follows. We use another sort `RealLabel`, which is declared as a sub-sort of `Label`. The 17 constants `l1`, `l2`, ..., `l15`, `cs`, and `ncs` are declared as those of `RealLabel`. Then, all we have to do is to declare one more equation "`eq (RL1`

`= RL2) = (RL1 == RL2) .`", in addition to the equation "`eq (L = L) = true .`", where `RL1` and `RL2` are CafeOBJ variables of `RealLabel`, and the operator `_==_` is a built-in predicate. The built-in predicate returns `true` if given two terms reduce to a same term and `false` otherwise. That is, it returns `false` even if two terms may represent a same data value. This is why we cannot use `_==_` naively as the equivalence predicate on data values for verification. `RealLabel` is only used to declare the 17 constants. In other places in the specification, `Label` is used.

5.3 Assignments to Shared Variables

Since the Bakery protocol does not assume atomic reads and writes to shared variables, we cannot model an assignment ($x := E;$) to a shared variable as one transition. The assignment is modeled as two or more transitions. The two or more transitions model the following things:

- Zero or more transitions model the calculation of the expression E. If it is not necessary to divide the calculation into multiple steps such that E is a data value such as 0, no transitions are used.
- One transition models the situation that the assignment has started but not finished.
- One transition models the situation that the assignment has finished.

Let `beginWtX` and `endWtX` be the last two transitions, and `x` be the observer with which the value of the variable x is obtained. One of the equations defining `beginWtX` looks like

`ceq x(beginWtX(S,I)) =` *anArbVal* `if c-beginWtX(S,I) .`

and one of the equations defining `endWtX` looks like

`ceq x(endWtX(S,I)) =` *theValOfX* `if c-endWtX(S,I) .`

where `S` is a CafeOBJ variable of a hidden sort denoting the state space, `I` is a CafeOBJ variable of a visible sort denoting process identifications, *anArbVal* is a term denoting an arbitrary value of the visible sort, and *theValOfX* is a term denoting the values obtained by calculating E. In $calS_{\text{Bakery}}$, *anArbVal* is an arbitrary natural number, which is represented as a term `next(`v`)`, where v is a natural number.

When another process `J` than `I` tries to read the variable x in the state `beginWtX(S,I)`, which corresponds to a situation where a `J`'s read to x overlaps an `I`'s write to x , the value obtained by J is arbitrary.

5.4 Choice of Arguments in an Arbitrary Order

The arguments of the *maximum* function can be read in any order. One way to respect the arbitrary choice of arguments and calculate $maximum(number[1], \ldots, number[N])$ is as follows:

1. Set temporary variables *tmp* and m to $\{1, \ldots, N\}$ and 0.
2. If *tmp* is empty, then the calculation is done and m contains the result; otherwise go to 3.
3. Choose and delete an arbitrary number k from *tmp*, set m to $\max(m, k)$, and go to 2.

Three transitions are used to model the calculation: `setTmp`, `checkLC` and `findMax`. Let `tmp` and `m` be the observers with which the values of *tmp* and m are obtained. Let `step1`, `step2` and `step3` denote the locations corresponding to the three steps described above, respectively. Let `step4` denotes the location to which the process moves after finishing the calculation. Let `pc` be the observer that returns the location where the process is.

Some of the equations defining `setTmp` look like

```
ceq pc(setTmp(S)) = step2 if c-setTmp(S) .
ceq tmp(setTmp(S)) = mkSet(numOfProcs) if c-setTmp(S) .
ceq m(setTmp(S)) = 0 if c-setTmp(S) .
```

some of the equations defining `checkLC` look like

```
ceq pc(checkLC(S)) = (if empty?(tmp(S)) then step4 else step3 fi)
    if c-checkLC(S) .
eq tmp(checkLC(S)) = tmp(S) .
eq m(checkLC(S)) = m(S) .
```

and some of the equations defining `findMax` look like

```
ceq pc(findMax(S,K)) = step2 if c-findMax(S,K) .
ceq tmp(findMax(S,K)) = del(tmp(S),K) if c-findMax(S,K) .
ceq m(findMax(S,K)) = max(m(S),K) if c-findMax(S,K) .
```

where `S` is a CafeOBJ variable of a hidden sort denoting the state space, `K` is a CafeOBJ variable of `Nat`, `c-setTmp(S)` is `pc(S) = step1`, `c-checkLC(S)` is `pc(S) = step2`, and `c-findMax(S,K)` is `pc(S) = step3 and K \in tmp(S)`. The transition `findMax` arbitrarily chooses a natural number `K` that is in `tmp(S)`.

5.5 Observers and Transitions

Seven observers are used, which are declared as follows:

```
bop pc     : Sys Nat -> Label   bop choosing : Sys Nat -> Nat
bop number : Sys Nat -> Nat     bop j        : Sys Nat -> Nat
bop tmp    : Sys Nat -> NatSet  bop m        : Sys Nat -> Nat
bop rand   : Sys -> Nat
```

`pc` returns the location where a given process is in a given state. `choosing`, `number` and `j` correspond to the variables found in the bakery protocol. `tmp` and `m` are used to model the calculation of $maximum(number[1], \ldots, number[N])$. `rand` returns an arbitrary natural number, which is used to model assignments to shared variables.

```
l1: beginWtCh1   ⟶  L1: choosing[i] := 1;
l2: endWtCh1

l3: setTmp       ⟶      number[i] := 1 + maximum(number[1], ..., number[N]);
l4: checkLC1
l5: findMax
l6: beginWtNum1
l7: endWtNum1

l8: beginWtCh2   ⟶      choosing[i] := 0;
l9: endWtCh2

l10: setJ        ⟶      for j = 1 step 1 until N do
l11: checkLC2              begin
l12: checkCh                 L2: if choosing[j] ≠ 0 then goto L2;
l13: checkNum                L3: if number[j] ≠ 0 and (number[j], j) < (number[i], i)
                                   then goto L3;
                           end;

cs: execCS       ⟶      critical section;

l14: beginWtNum2 ⟶      number[i] := 0;
l15: endWtNum2

ncs: execNCS     ⟶      nonciritcal section;

ncs: tryCS       ⟶      goto L1;
```

Fig. 2. Correspondence between transitions and the protocol

18 transitions are used, which are declared as follows:

```
bop beginWtCh1  : Sys Nat -> Sys  bop endWtCh1  : Sys Nat -> Sys
bop setTmp      : Sys Nat -> Sys  bop checkLC1  : Sys Nat -> Sys
bop findMax     : Sys Nat Nat -> Sys
bop beginWtNum1 : Sys Nat -> Sys  bop endWtNum1 : Sys Nat -> Sys
bop beginWtCh2  : Sys Nat -> Sys  bop endWtCh2  : Sys Nat -> Sys
bop setJ        : Sys Nat -> Sys  bop checkLC2  : Sys Nat -> Sys
bop checkCh     : Sys Nat -> Sys  bop checkNum  : Sys Nat -> Sys
bop execCS      : Sys Nat -> Sys
bop beginWtNum2 : Sys Nat -> Sys  bop endWtNum2 : Sys Nat -> Sys
bop execNCS     : Sys Nat -> Sys  bop tryCS     : Sys Nat -> Sys
```

Figure 2 shows the correspondence between the 18 transitions and the bakery protocol in the ALGOL style. The first 16 labels represented by the 16 constants `l1`, `l2`, ..., `l15` and `cs` correspond to the first 16 transitions, respectively. The label `ncs` correspond to both `execNCS` and `tryCS`.

```
-- setTmp
eq c-setTmp(S,I) = (pc(S,I) = l3) .
ceq pc(setTmp(S,I),J)
    = (if I = J then l4 else pc(S,J) fi) if c-setTmp(S,I) .
eq choosing(setTmp(S,I),J) = choosing(S,J) .
eq number(setTmp(S,I),J)   = number(S,J) .
eq j(setTmp(S,I),J)        = j(S,J) .
ceq tmp(setTmp(S,I),J)
    = (if I = J then mkSet(numOfProcs) else tmp(S,J) fi)
    if c-setTmp(S,I) .
ceq m(setTmp(S,I),J)
    = (if I = J then 0 else m(S,J) fi) if c-setTmp(S,I) .
eq rand(setTmp(S,I))       = rand(S) .
bceq setTmp(S,I)           = S  if not c-setTmp(S,I) .

-- checkLC1
eq c-checkLC1(S,I) = (pc(S,I) = l4) .
ceq pc(checkLC1(S,I),J)
    = (if I = J then (if empty?(tmp(S,I)) then l6 else l5 fi)
       else pc(S,J) fi) if c-checkLC1(S,I) .
eq choosing(checkLC1(S,I),J) = choosing(S,J) .
eq number(checkLC1(S,I),J)   = number(S,J) .
eq j(checkLC1(S,I),J)        = j(S,J) .
eq tmp(checkLC1(S,I),J)      = tmp(S,J) .
eq m(checkLC1(S,I),J)        = m(S,J) .
eq rand(checkLC1(S,I))       = rand(S) .
bceq checkLC1(S,I)           = S if not c-checkLC1(S,I) .

-- findMax
eq c-findMax(S,I,K) = (pc(S,I) = l5 and K \in tmp(S,I)) .
ceq pc(findMax(S,I,K),J)
    = (if I = J then l4 else pc(S,J) fi) if c-findMax(S,I,K) .
eq choosing(findMax(S,I,K),J) = choosing(S,J) .
eq number(findMax(S,I,K),J)   = number(S,J) .
eq j(findMax(S,I,K),J)        = j(S,J) .
ceq tmp(findMax(S,I,K),J)
    = (if I = J then del(tmp(S,I),K) else tmp(S,J) fi)
    if c-findMax(S,I,K) .
ceq m(findMax(S,I,K),J)
    = (if I = J then max(m(S,I),number(S,K)) else m(S,J) fi)
    if c-findMax(S,I,K) .
eq rand(findMax(S,I,K))       = rand(S) .
bceq findMax(S,I,K)           = S if not c-findMax(S,I,K) .
```

Fig. 3. Definitions of transitions (1)

`setTmp`, `checkLC1` and `findMax` correspond to $maximum(number[1], \ldots, number[N])$. `beginWtNum1` and `endWtNum1` correspond to the assignment of the value obtained by incrementing the result of the calculation to $number[i]$.

```
-- beginWtNum1
eq c-beginWtNum1(S,I) = (pc(S,I) = l6) .
ceq pc(beginWtNum1(S,I),J)
    = (if I = J then l7 else pc(S,J) fi) if c-beginWtNum1(S,I) .
eq choosing(beginWtNum1(S,I),J) = choosing(S,J) .
ceq number(beginWtNum1(S,I),J)
    = (if I = J then rand(S) else number(S,J) fi) if c-beginWtNum1(S,I) .
eq j(beginWtNum1(S,I),J)        = j(S,J) .
eq tmp(beginWtNum1(S,I),J)      = tmp(S,J) .
eq m(beginWtNum1(S,I),J)        = m(S,J) .
ceq rand(beginWtNum1(S,I))      = next(rand(S)) if c-beginWtNum1(S,I) .
bceq beginWtNum1(S,I)           = S if not c-beginWtNum1(S,I) .

-- endWtNum1
eq c-endWtNum1(S,I) = (pc(S,I) = l7) .
ceq pc(endWtNum1(S,I),J)
    = (if I = J then l8 else pc(S,J) fi) if c-endWtNum1(S,I) .
eq choosing(endWtNum1(S,I),J) = choosing(S,J) .
ceq number(endWtNum1(S,I),J)
    = (if I = J then s(m(S,I)) else number(S,J) fi) if c-endWtNum1(S,I) .
eq j(endWtNum1(S,I),J)        = j(S,J) .
eq tmp(endWtNum1(S,I),J)      = tmp(S,J) .
eq m(endWtNum1(S,I),J)        = m(S,J) .
eq rand(endWtNum1(S,I))       = rand(S) .
bceq endWtNum1(S,I)           = S if not c-endWtNum1(S,I) .
```

Fig. 4. Definitions of transitions (2)

`setJ` corresponds to the assignment of 1 to the process i's local variable j. `checkLC2` corresponds to the loop termination check. `checkCh` and `checkNum` correspond to the first and second conditional statements in the inner loop of the protocol, respectively.

5.6 Definitions of Transitions

We cannot show all equations defining the 18 transitions due to the space limitation. We show the equations defining `setTmp`, `checkLC1`, `findMax`, `beginWtNum1` and `endWtNum1` in Fig. 3 and Fig. 4. Lines starting with "`--`" are comments.

6 Verification Based on the Specification

We describe the mechanized proof that $\mathcal{S}_{\text{Bakery}}$ satisfies the mutual exclusion property based on the specification of $\mathcal{S}_{\text{Bakery}}$. The proof is conducted by writing proof socres in CafeOBJ and executing them with the CafeOBJ system. Proof scores are proofs or proof plans written in an algebraic specification language such as CafeOBJ.

6.1 Formalization of the Mutual Exclusion Property

The mutual exclusion property can be stated as there is at most one process in the critical section at any given moment. This can be rephrased as if there are processes in the critical section, then they are identical. The property is formalized as follows:

```
eq inv1(S,P,Q) = (isPid(P) and isPid(Q) and
     pc(S,P) = cs and pc(S,Q) = cs implies P = Q) .
```

Since $\mathcal{S}_{\text{Bakery}}$ does not explicitly disallow processes whose identifications are not in $\{1, \ldots, N\}$ to participate in the protocol, we need to have `isPid(P) and isPid(Q)` as part of the premises.

What to do is to prove `inv1(S,P,Q)` for all reachable states `S` and all natural numbers `P` and `Q`. The proof is done by induction on the structure of the reachable state space. Then, we declare the constant `istep1` of `Bool`, which is defined as "`eq istep1 = inv1(s,p,q) implies inv1(s',p,q) .`", where `s` is a constant of `Sys` denoting an arbitrary state, `s'` is a constant of `Sys` denoting an arbitrary successor state of `s`, and `p` and `q` are constants of `Nat` denoting arbitrary natural numbers. We suppose that the importation of a module, say `ISTEP`, makes those operators, equations and constants available.

6.2 Lemmas of the Verification

The verification needs to prove that 12 more state predicates are invariant wrt $\mathcal{S}_{\text{Bakery}}$. The 12 state predicates are shown in Table 1. The predicates `inWS2`, `inWS&CS`, `inCM` and `inZS` are defined as follows:

Predicate	Definition
`inWS1(L)`	`L = l8 or-else L = l9 or-else L = l10`
`inWS2(L)`	`L = l11 or-else L = l12 or-else L = l13`
`inWS(L)`	`inWS1(L) or-else inWS2(L)`
`inWS&CS(L)`	`inWS(L) or-else L = cs or-else L = l14`
`inCM(L)`	`L = l4 or-else L = l5 or-else L = l6 or-else L = l7`
`inZS(L)`	`L = ncs or-else L = l1 or-else L = l2 or-else L = l3 or-else L = l4 or-else L = l5 or-else L = l6`

Instead of `_or_`, `_or-else_` is used to prevent Boolean terms from fully reducing to their exclusive-or normal forms.

6.3 Proof Score of `inv1`

Let us take a close look at the protocol to check which transitions preserve `inv1` and which do not seem. In the induction case for a transition that does not seem to preserve `inv1`, we may need lemmas.

All transitions except for `checkLC2` preserve `inv1`. This is because they change `pc(`s`,`p`)` for some process identification p to a value that is different from `cs` or do not change the value returned by any observer. In the former, the change makes the premise of `inv1` false, namely making `inv1` true. In the latter, `inv1` is clearly preserved. We still need to prove the induction case for each transition. But, such

Table 1. The lemmas used in the verification

Lemma	Definition
inv2(S,P,J)	(isPid(P) and inWS2(pc(S,P)) and 0 < J and J < j(S,P) and inWS&CS(pc(S,J)) and not(J = P)) implies < number(S,P),P > < < number(S,J),J >
inv3(S,P,J)	(isPid(P) and pc(S,P) = cs and 0 < J and J < s(numOfProcs) and inWS&CS(pc(S,J)) and not(J = P)) implies < number(S,P),P > < < number(S,J),J >
inv4(S,P)	inWS&CS(pc(S,P)) implies 0 < number(S,P)
inv5(S,P,Q)	(isPid(P) and isPid(Q) and inCM(pc(S,Q)) and (inWS2(pc(S,P)) or-else pc(S,P) = cs) and not(P \in tmp(S,Q)) and Q < j(S,P) and not(P = Q)) implies (number(S,P) = m(S,Q) or number(S,P) < m(S,Q))
inv6(S,P)	(pc(S,P) = l6 or-else pc(S,P) = l7) implies empty?(tmp(S,P))
inv7(S,P)	inWS2(pc(S,P)) implies (j(S,P) = s(numOfProcs) or-else j(S,P) < s(numOfProcs))
inv8(S,P)	pc(S,P) = cs implies j(S,P) = s(numOfProcs)
inv9(S,P)	(isPid(P) and isPid(j(S,P)) and inCM(pc(S,j(S,P))) and pc(S,P) = l13 and not(P \in tmp(S,j(S,P))) and not(P = j(S,P)) and (number(S,j(S,P)) = 0 or-else not(< number(S,j(S,P)),j(S,P) > < < number(S,P),P >))) implies (number(S,P) = m(S,j(S,P)) or number(S,P) < m(S,j(S,P)))
inv10(S,P)	(inCM(pc(S,j(S,P))) and pc(S,P) = l12 and choosing(S,j(S,P)) = 0) implies (number(S,P) = m(S,j(S,P)) or number(S,P) < m(S,j(S,P)))
inv11(S,P)	(inCM(pc(S,P)) or-else pc(S,P) = l3) implies not(choosing(S,P) = 0)
inv12(S,P)	inZS(pc(S,P)) implies number(S,P) = 0
inv13(S,P)	(pc(S,P) = l12 or-else pc(S,P) = l13) implies j(S,P) < s(numOfProcs)

a thought experiment makes it clear that only case splitting can discharge the induction case for all transitions except for `checkLC2` but we may also need lemmas for `checkLC2`.

Let us consider the following *proof passage* (a fragment of a proof score):

```
open ISTEP
-- arbitrary values
  op i : -> Nat .
-- assumptions
  -- eq c-checkLC2(s,i) = true .
  eq pc(s,i) = l11 .
  --
  eq p = i .
  eq (q = i) = false .
  eq j(s,i) = s(numOfProcs) .
  eq pc(s,q) = cs .
-- successor state
```

```
  eq s' = checkLC2(s,i) .
-- check
  red istep1 .
close
```

The command `open` makes a temporary module in which a given module (`ISTEP` in this case) is imported, and the command `close` destroys such a module. The constant `i` is used to denote an arbitrary natural number. The constant `s` is used to denote an arbitrary state in which the first five equations hold. The last equation says that `s'` is an arbitrary `checkLC2`-successor state of `s`. The command `red` reduces a given term.

Since `pc(s',p)` equals `cs`, `pc(s',q)` equals `cs` and `p` does not equal `q`, if `p` and `q` are used as process identifications and `s` is reachable, then `inv1` is not invariant wrt $\mathcal{S}_{\text{Bakery}}$. We need to conjecture lemmas to discharge the proof passage.

A close inspection of the bakery protocol allows us to conjecture the following two statements: for all reachable states s, all process identifications p and all natural numbers q,

1. If `pc(`s`,`p`)` is `l1`, `l2` or `l3`, `pc(`s`,`q`)` is `l8`, ..., `cs` or `l14`, q is not p, and q is greater than 0 and less than `j(`s`,`p`)`, then `< number(`s`,`p`),`p `>` is less than `< number(`s`,`q`),`q `>`.
2. If `pc(`s`,`p`)` is `cs`, `pc(`s`,`q`)` is `l8`, ..., `cs` or `l14`, q is not p, and q is greater than 0 and less than `s(numOfProcs)`, then `< number(`s`,`p`),`p `>` is less than `< number(`s`,`q`),`q `>`.

Both statements roughly say that if it has been decided that p has high priority over q when p is in the inner loop of the protocol, the situation lasts while p is in the inner loop or in the critical section. They corresponds to `inv2` and `inv3` in Table 1, respectively.

Instead of `istep1`, we reduce `inv2(s,p,q) and inv3(s,q,p) implies istep1` in the proof passage, whose results is `true`. This is because if both `p` and `q` are process identifications, `inv2(s,p,q) and inv3(s,q,p)` is equivalent to `< number[p],p > < < number[q],q > and < number[q],q > < < number[q],q >`, which reduces to `false`.

In the proof score of `inv1`, there is one more proof passage, which uses `inv2` and `inv3`, to discharge the proof passage:

```
open ISTEP
  op i : -> Nat .
  eq pc(s,i) = l11 .
  eq (p = i) = false .
  eq q = i .
  eq j(s,i) = s(numOfProcs) .
  eq pc(s,p) = cs .
  eq s' = checkLC2(s,i) .
  red inv2(s,q,p) and inv3(s,p,q) implies istep1 .
close
```

Comments are omitted. Any other proof passages do not use any lemmas.

6.4 Proof Score of inv2

All transitions except for endWtNum1, setJ and checkNum preserve inv2. This is because although they may change pc(*s*,*p*) for some process identification, they do not change the truth value of inv2. setJ may change pc(*s*,*p*) for some process identification to l11 from l10, but if it does, it sets j(*s*,*p*) to s(0), which makes the premise of inv2 false and then makes inv2 true. We may need lemmas in the induction case for endWtNum1 and checkNum.

Let us consider the following proof passage:

```
open ISTEP
-- arbitrary values
  op i : -> Nat .
-- assumptions
  -- eq c-endWtNum1(s,i) = true .
  eq pc(s,i) = l7 .
  --
  eq (p = i) = false .
  eq j = i .
  eq number(s,p) < s(m(s,i)) = false .
  eq (number(s,p) = s(m(s,i)) and p < i) = false .
  eq 0 < p = true .
  eq p < s(numOfProcs) = true .
  eq (pc(s,p) = l11 or-else pc(s,p) = l12
                    or-else pc(s,p) = l13) = true .
  eq 0 < i = true .
  eq i < j(s,p) = true .
  eq (j(s,p) = s(numOfProcs) or-else j(s,p) = numOfProcs
                             or-else j(s,p) < numOfProcs) = true .
  eq i < s(numOfProcs) = true .
  eq tmp(s,i) = empty .
-- successor state
  eq s' = endWtNum1(s,i) .
-- check
  red istep2 .
close
```

CafeOBJ returns false for this proof passage. If inv2 is invariant wrt $\mathcal{S}_{\text{Bakery}}$, an arbitrary state s characterized by the first 13 equations must be unreachable. We need to find lemmas to show that s is unreachable.

A close inspection of the bakery protocol allows us to conjecture the following statement: for all reachable states s and all process identifications p, q,

1. If pc(*s*,*p*) is l1, l2, l3 or cs, pc(*s*,*q*) is l4, ..., l7, q is not p, q is less than j(*s*,*p*) and p is not in tmp(*s*,*p*), then number(*s*,*p*) is less than or equal to m(*s*,*q*).

The statement roughly says that if it has been decided that p has high priority over q when p is in the inner loop of the protocol and q is calculating

$maximum(number[1], \ldots, number[N])$, the situation lasts while p is in the inner loop or in the critical section and q is calculating the expression. The statement corresponds to `inv5` in Table 1.

`inv5(s,p,j)` reduces to `false` in the proof passage, which means that if `inv5` is invariant wrt $\mathcal{S}_{\text{Bakery}}$, an arbitrary state `s` characterized by the first 13 equations in the proof passage is unreachable. Therefore, `inv5` discharges the proof passage.

In addition to `inv5`, the induction case where `endWtNum1` is considered needs `inv6` and `inv7` shown in Table 1. The induction case for `checkNum` needs `inv4` shown in Table 1. The induction case where other transitions are considered does not need any lemmas.

6.5 Other Proof Scores

All the other lemmas except for `inv10` are proved by induction on the structure of the reachable state space. A simple logical calculation deduces `inv10` from `inv11`, which is also done by writing a proof score. The proofs of some lemmas also need lemmas, which are as follows: (1) `inv3`: `inv2`, `inv6` `inv5` and `inv8`; (2) `inv4`: no lemmas; (3) `inv5`: `inv8`; (4) `inv6`: no lemmas; (5) `inv7`: `inv13`; (6) `inv8`: no lemmas; (7) `inv9`:`inv10` and `inv12`; (8) `inv11`: no lemmas; (9) `inv12`: no lemmas; (10) `inv13`: `inv7`. The verification also needs several lemmas on natural numbers.

7 Discussion

7.1 Choice of Arguments in an Arbitrary Order

We have described a way to formalize choice of arguments in an arbitrary order in Subsect. 5.4. Another seemingly possible way to do so, which we first came up with, is to use the operator `choose` declared and defined as follows:

```
op choose : NatSet -> Nat
eq choose(X S) = X .
```

where `X` and `S` are CafeOBJ variables of `Nat` and `NatSet`, respectively.

We thought that the equation successfully formalized an arbitrary choice of a natural number `X` among the set of natural numbers in which the natural number was. But, the equation makes all natural numbers identical. This is because since the juxtaposition operator `__` is associative and commutative, `x y xs` equals `y x xs` where `x` and `y` are arbitrary natural numbers and `xs` is an arbitrary set of natural numbers and then `choose(x y xs)` equals `choose(y x xs)`, which leads to the equivalence of `x` and `y` due to the equation.

7.2 Lemmas on Data

The verification also needs lemmas on natural numbers. There are at least two ways to declare lemmas on data such as natural numbers in the OTS/CafeOBJ

method. They can be declared as (1) standard (conditional) equations and (2) Boolean terms. An example of the first solution is "`eq (X < Y and Y < X) = false .`" and an example of the second solution is "`eq natLem7(X,Y,Z) = (X < Y and Y < Z implies X < Z) .`". Both lemmas are used in the verification.

Both solutions have pros and cons. The first solution's good and bad points are as follows:

Good points. Basically lemmas as standard equations can be used automatically by reduction and users do not care about where lemmas should be used.

But, there are some lemmas, which cannot be used automatically by reduction. An example is "`eq X < Z = true if X < Y and Y < Z .`". This is because the variable `Y` in the condition does not occur on the left-hand side of the equation.

Bad points. Lemmas as standard equations may affect confluence and terminating of specifications. We suppose that we use the lemma "`eq X < Y and Y < s(X) = false .`". If we also declare the lemma "`eq Y < s(X) = (Y = X or-else Y < X) .`", the specification becomes nonconfluent. The first lemma should be modified as "`eq X < Y and (Y = X or-else Y < X) = false .`".

The second solution's good and bad points are as follows:

Good points. Lemmas as Boolean terms do not affect confluence and terminating of specifications.

Bad points. Lemmas as Boolean terms are not used automatically by reduction. Users need to care about where lemmas should be used.

Since lemmas should be used explicitly, however, the second solution allows users to understand the reason why lemmas should be used and makes proofs more traceable.

8 Related Work

The bakery protocol has been often used as a benchmark to demonstrate that proposed verification methods and/or tools are powerful enough. Among such methods and tools are [3,4,5].

Mori, et al. [3] proves with a resolution-based theorem prover implemented on top of the CafeOBJ system that a simplified version of the protocol satisfies the mutual exclusion property. They assume that N processes participate in the protocol as we do. Their way to model the protocol is similar to ours. The resolution-based theorem prover could be used to proves that $\mathcal{S}_{\mathrm{Bakery}}$ satisfies the mutual exclusion protocol.

Meseguer, et al. [4] proves with an abstraction method and the Maude LTL model checker [17,18] that a simplified version of the protocol satisfies the mutual exclusion property. They only consider that two processes participate in the protocol. Their verification technique is based on rewriting like ours. But, their

way to specify state machines is different from ours. Their method needs to fix a concrete number, say 2, of processes participating in the protocol. Although it is possible to represent nonatomic reads and writes in their method, it does not seem clear to come up with a good abstraction when nonatomic reads and writes are taken into account.

de Moura, et al. [5] proves with the SAL [19] implementation of k-induction that a simplified version of the protocol satisfies the mutual exclusion property. The implementation uses an SMT-based bounded model checker. They only consider that two processes participate in the protocol. Their way to specify state machines needs to fix a concrete number, say 2, of processes participating in the protocol like the method used in [4]. It does not seem clear to model nonatomic reads and writes because it does not seem clear to express arbitrary values.

All the simplified versions of the protocol assume that a read and a write to a shared variable are performed exclusively. The true bakery protocol has been rarely used.

Lamport gives an informal proof that the bakery protocol satisfies some properties including the mutual exclusion one [2]. He also gives a more rigorous but nonassertional proof that a variant of the bakery protocol satisfies the mutual exclusion property [6]. The nonassertional proof does not assume any atomicity, but uses as axioms some relations between reads and writes to shared variables.

He gives an assertional proof that the bakery protocol satisfies the mutual exclusion protocol [7]. The proof does not assume any atomicity, either. In the proof, Lamport introduced the predicate transformers *win* (the weakest invariant operator) and *sin* (the strongest invariant operator), which generalize *wlp* (the weakest liberal precondition operator) and *sp* (the strongest postcondition operator). Statements such as assignments that constitute the protocol are represented by (nonatomic) operations, which basically consist of atomic operations. But, the proof does not assume what atomic operations constitute the operation denoting each statement of the protocol. The proof revealed the two hidden assumptions that the assignment $(number[i] := 1 + maximum(number[1], \ldots, number[N]))$ sets $number[i]$ (1) positive and (2) greater than $number[j]$, even if it is executed while the value of $number[k]$ is being changed, for $k \neq i, j$.

We assume some atomicity, namely that every transition is atomic. But, we do not make an assumption that reads and writes to shared variables are atomic. In our abstract model of the protocol, the assignment $(number[i] := 1 + maximum(number[1], \ldots, number[N]))$ is represented by five transitions, while it is represented by a nonatomic operation in the Lamport's abstract model. Our proof does not need the two hidden assumptions. This is because our abstract model is more concrete than the Lamport's abstract model. Neither the Lamport's assertional and nonassertional proofs do not seem to have been mechanized, although they could be.

Lamport also uses multiple atomic transitions (or atomic operations) to represent a nonatomic assignment to a shared variable in [8]. He defines a nonatomic assignment of a value v to a shared variable x by the two atomic assignments

$x :=?$ and $x := v$. When x equals ?, a read to x obtains an arbitrary value, which needs to change the semantics of reads to shared variables. He proves that the bakery protocol satisfies some safety and liveness properties with consideration of nonatomic reads and writes. The proof does not need the two hidden assumptions.

Our abstract model is similar to the Lamport's one described in [8]. Our abstract model is written in an algebraic specification language, however, while his is written as a flowchart. The expressiveness of an algebraic specification language allows us to represent an arbitrary natural number as a term. Therefore, we do not need to change the semantics of reads to shared variables.

Another way to define a nonatomic assignment to a shared variable by multiple atomic transitions (or atomic operations) is given in [20]. A nonatomic assignment of a value v to a shared variable x is represented as a nondeterministic program fragment in which x is incremented (and decremented) arbitrarily but finitely many times and finally x is set to v. This solution does not need to change the semantics of reads to shared variables.

9 Conclusion

We have described a fully formal proof that the bakery protocol satisfies the mutual exclusion protocol. The proof has been mechanized with CafeOBJ. The CafeOBJ system has been used as an interactive proof assistant. Nonatomic reads and writes to shared variables have been formalized by representing an assignment to a shared variable with multiple atomic transitions. Our formal model of the protocol has states in which a shared variable is being modified. A read to the variable in such states obtains an arbitrary value, which is represented as a CafeOBJ term.

One piece of our future work is to conduct the verification based on the specification where integers are used instead of natural numbers. Another one is to prove that the protocol satisfies other properties such as the lockout (or starvation) freedom property, which is a liveness property.

References

1. Dijkstra, E.W.: Solution of a problem in concurrent programming control. CACM 8, 569 (1965)
2. Lamport, L.: A new solution of Dijkstra's concurrent programming problem. CACM 17, 453–455 (1974)
3. Mori, A., Futatsugi, K.: Cafeobj as a tool for behavioral system verification. In: Okada, M., Pierce, B.C., Scedrov, A., Tokuda, H., Yonezawa, A. (eds.) ISSS 2002. LNCS, vol. 2609, pp. 2–16. Springer, Heidelberg (2003)
4. Meseguer, J., Palomino, M., Martí-Oliet, N.: Equational abstraction. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 2–16. Springer, Heidelberg (2003)
5. de Moura, L., Rueß, H., Sorea, M.: Bounded model checking and induction: From refutation to verification. In: Hunt Jr., W.A., Somenzi, F. (eds.) CAV 2003. LNCS, vol. 2725, pp. 14–26. Springer, Heidelberg (2003)

6. Lamport, L.: A new approach to proving the correctness of multiprocess programs. ACM TOPLAS 1, 84–97 (1979)
7. Lamport, L.: win and sin: Predicate transformers for concurrency. ACM TOPLAS 12, 396–428 (1990)
8. Lamport, L.: Proving the correctness of multiprocess programs. IEEE TSE SE-3, 125–143 (1977)
9. Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method. In: Najm, E., Nestmann, U., Stevens, P. (eds.) FMOODS 2003. LNCS, vol. 2884, pp. 170–184. Springer, Heidelberg (2003)
10. Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method. In: Futatsugi, K., Jouannaud, J.-P., Meseguer, J. (eds.) Algebra, Meaning, and Computation. LNCS, vol. 4060, pp. 596–615. Springer, Heidelberg (2006)
11. Diaconescu, R., Futatsugi, K.: CafeOBJ report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification. AMAST Series in Computing, vol. 6. World Scientific, Singapore (1998)
12. Peterson, G.L.: Myths about the mutual exclusion problem. IPL 12, 115–116 (1981)
13. Anderson, T.E.: The performance of spin lock alternatives for shared-memory multiprocessors. IEEE TPDS 1, 6–16 (1990)
14. Mellor-Crummery, J.M., Scott, L.: Algorithms for scalable synchronization on shared-memory multiprocessors. ACM TOCS 9, 21–65 (1991)
15. Hsiang, J., Dershowitz, N.: Rewrite methods for clausal and nonclausal theorem proving. In: Díaz, J. (ed.) ICALP 1983. LNCS, vol. 154, pp. 331–346. Springer, Heidelberg (1983)
16. Nakamura, M., Futatsugi, K.: On equality predicates in algebraic specification languages. In: Jones, C.B., Liu, Z., Woodcock, J. (eds.) ICTAC 2007. LNCS, vol. 4711, pp. 381–395. Springer, Heidelberg (2007)
17. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: 4th WRLA. ENTCS, vol. 71. Elsevier, Amsterdam (2004)
18. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: All About Maude – A High-Performance Logical Framework: How to Specify, Program and Verify Systems in Rewriting Logic. In: Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C. (eds.) All About Maude - A High-Performance Logical Framework. LNCS, vol. 4350. Springer, Heidelberg (2007)
19. de Moura, L., Owre, S., Rueß, H., Rushby, J., Shankar, N., Sorea, M., Tiwari, A.: SAL 2. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 496–500. Springer, Heidelberg (2004)
20. Anderson, J.H., Gouda, M.G.: Atomic semantics of nonatomic programs. IPL 28, 99–103 (1988)