Journal of Universal Computer Science, vol. 19, no. 6 (2013), 771-804
submitted: 2/3/11, accepted: 25/3/13, appeared: 28/3/13

Compositionally Writing Proof Scores of Invariants in the OTS/CafeOBJ Method

Kazuhiro Ogata
(School of Information Science
Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan
ogata@jaist.ac.jp)

Kokichi Futatsugi
(School of Information Science
Japan Advanced Institute of Science and Technology (JAIST), Nomi, Japan
futatsugi@jaist.ac.jp)

Abstract: Observational transition systems (OTSs) are state machines that can be described as behavioral specifications in CafeOBJ, an algebraic specification language and processor. The OTS/CafeOBJ method uses OTSs and CafeOBJ for systems specification and verification. Simultaneous induction is intensively used to prove that an OTS enjoys invariants in the method. To prove that two state predicates p and q are invariants with respect to an OTS $\mathcal{S}$, simultaneous induction generates the proof obligations: (1) $p(\upsilon_0)$ and $p(\upsilon) \wedge q(\upsilon) \Rightarrow p(\upsilon')$, and (2) $q(\upsilon_0)$ and $p(\upsilon) \wedge q(\upsilon) \Rightarrow q(\upsilon')$ for each initial state υ_0, each state υ and each successor state υ' of υ. Instead, we may also use the proof obligations: (1) $q(\upsilon) \Rightarrow p(\upsilon)$, and (2) $q(\upsilon_0)$ and $p(\upsilon) \wedge q(\upsilon) \Rightarrow q(\upsilon')$. The proof technique generating proof obligations like this is called semi-simultaneous induction. The proof obligation is equivalent to (1) $q(\upsilon) \Rightarrow p(\upsilon)$, and (2) $q(\upsilon_0)$ and $q(\upsilon) \Rightarrow q(\upsilon')$. But, the former may need less cases, making proofs shorter, than the latter. More importantly, the former makes it possible to record the process in which way lemmas have been conjectured. This article demonstrates some benefits of (semi-)simultaneous induction, describes semi-simultaneous induction and justifies it.

Key Words: algebraic specification, CafeOBJ, invariant, observational transition system (OTS), simultaneous induction

Category: D.2.4, F.3.1

1 Introduction

Formal methods use mathematics and logics to formalize requirements, designs and/or programs of systems, and verify that there are no inconsistencies in requirements, designs enjoy requirements, and/or programs conforms to designs [Woodcock et al, 2009]. Algebraic specification techniques have been developed in formal methods and several algebraic specification languages and processors have been proposed. *CafeOBJ* [Diaconescu and Futatsugi, 1998] is one such language and processor. Behavioral specifications [Goguen and Malcolm, 2000, Diaconescu and Futatsugi, 2000] are algebraic specifications of systems behavior and can be described in CafeOBJ.

Observational transition systems (*OTSs*) are state machines that can be described as behavioral specifications. The *OTS/CafeOBJ method* [Ogata and Futatsugi, 2003, Ogata and Futatsugi, 2006, Diaconescu et al, 2003] uses OTSs and CafeOBJ for systems specification and verification. In the OTS/CafeOBJ method, (1) a system design is formalized as an OTS, (2) the OTS is described as a behavioral specification in CafeOBJ and system requirements or properties are written in CafeOBJ, and (3) it is proved that the OTS enjoys the properties by writing what are called *proof scores* in CafeOBJ and executing them with the CafeOBJ processor. The OTS/CafeOBJ method has been applied for specifications and verification of a wide range of applications. Among such applications are security protocols [Ogata and Futatsugi, 2010, Ouranos et al, 2010], e-government systems [Kong et al, 2010], mobile systems [Ouranos, 2007] and license choice algorithms [Triantafyllou et al, 2010].

Proof scores are compositionally written in the OTS/CafeOBJ method. Let us consider the proof that a state predicate p is an invariant with respect to (wrt) an OTS $\mathcal{S}$, namely the proof of $(\forall \upsilon : \mathcal{R}_{\mathcal{S}})p(\upsilon)$, where $\mathcal{R}_{\mathcal{S}}$ is the type denoting the set of all reachable states wrt $\mathcal{S}$. While writing the proof score wrt p, we often need other state predicates. But, we do not need to throw away the proof score wrt p constructed so far. All we have to do is to write the proof scores wrt the other state predicates. While doing so, we may also need yet other state predicates. Let us suppose that the proof score wrt p needs another state predicate q. All we need to do is to write the proof score wrt q. The proof score wrt p corresponds to the proofs of $p(\upsilon_0)$ and $p(\upsilon) \wedge q(\upsilon) \Rightarrow p(\upsilon')$, and the proof score wrt q to the ones of $q(\upsilon_0)$ and $p(\upsilon) \wedge q(\upsilon) \Rightarrow q(\upsilon')$ for each initial state υ_0, each state υ and each successor state υ' of υ. This way of writing proofs (or proof scores) is called *simultaneous induction* [Ogata and Futatsugi, 2003, Ogata and Futatsugi, 2006, Diaconescu et al, 2003]. The relations between p and q in the proof can be depicted as Fig. 1 (1).

The proofs of $p(\upsilon_0)$ and $p(\upsilon) \wedge q(\upsilon) \Rightarrow p(\upsilon')$ may be replaced with the one of $q(\upsilon) \Rightarrow p(\upsilon)$. The relations between p and q in the proof modified can be depicted as Fig. 1 (2). The proof technique that constructs proofs corresponding to Fig. 1 (2) is called *semi-simultaneous induction*.

The proofs of $q(\upsilon_0)$ and $p(\upsilon) \wedge q(\upsilon) \Rightarrow q(\upsilon')$ can also be replaced with the ones of $q(\upsilon_0)$ and $q(\upsilon) \Rightarrow q(\upsilon')$ because $q(\upsilon) \Rightarrow p(\upsilon)$. The relations between p and q in the proof modified further can be depicted as Fig. 1 (3), which corresponds to the INV rule [Manna and Pnueli, 1995].

The proof corresponding to Fig. 1 (2) is equivalent to the one to Fig. 1 (3), and essentially so is the one to Fig. 1 (1). But, the proof corresponding to Fig. 1 (2) may need less cases, making proof scores shorter, than the one to Fig. 1 (3). More importantly, the proof corresponding to Fig. 1 (2) makes it possible to record the

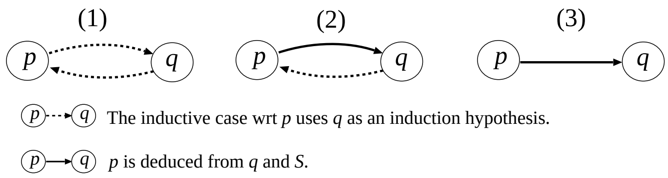

Figure 1: Some relations between two state predicates in invariant proofs

process in which way lemmas have been conjectured. This improves readability of proof scores for human users. High readability of proof scores is very important because we think that proof scores should be part of specifications, which are supposed to be read by human users. Note that we cannot conclude that p (and q) is an invariant wrt $\mathcal{S}$ from the proofs of $q(v) \Rightarrow p(v)$ and $p(v) \Rightarrow q(v)$.

This article generalizes the proof corresponding to Fig. 1 (2) and justifies it, and demonstrates some benefits of (semi-)simultaneous induction. The rest of the article is organized as follows. Section 2 describes CafeOBJ with some examples. Section 3 describes OTSs, and how to specify OTSs in CafeOBJ with a mutual exclusion protocol as an example. Section 4 describes proof scores with verification that the protocol enjoys the mutual exclusion property as an example. Section 5 describes simultaneous induction. Section 6 describes semi-simultaneous induction that can be used to conduct generalized proofs corresponding to Fig. 1 (2) and justifies it. Section 7 reports on another case study. Section 8 mentions related work. Section 9 concludes the article.

2 CafeOBJ

This section briefly introduces specification and verification in CafeOBJ with some concrete examples.

2.1 Specification

Basic units of CafeOBJ are modules. There are two kinds of modules in CafeOBJ: tight modules and loose modules. A tight module only accepts the smallest implementation that satisfies what are specified in the module, while a loose module can accept any implementations that satisfy what are specified in the module. A tight module is declared with the keyword `mod!`, and a loose module with the keyword `mod*`. What are declared in modules are module imports, sorts, operators, variables and equations. Some examples are used to describe them.

Process IDs are specified as the following module:

```
mod* PID {
  [Error Pid < Error&Pid]
  op _=_ : Error&Pid Error&Pid -> Bool {comm}
  vars EI EJ : Error&Pid
  var I : Pid
  var E : Error
  eq (I = E) = false .
  eq (EI = EI) = true .
  ceq [o2m-Error&Pid]: EI = EJ if EI = EJ .
}
```

PID is the name given to this module. Error, Pid and Error&Pid are sorts. Sorts can be partially ordered. Error and Pid are sub-sorts of Error&Pid. Any term of either Error or Pid is also a term of Error&Pid. Pid corresponds to the set of process IDs, Error to the set of error IDs, and Error&Pid to the set of both IDs. The module does not explicitly import any modules but implicitly does the built-in module BOOL, which is protected. Any user modules implicitly import BOOL and BOOL is protected in those modules unless otherwise stated. What are declared in BOOL include the sort Bool for truth values, the two constants true and false of Bool denoting the truth values, and operators denoting logical connectives. Operators without any arguments are called constants. Among operators denoting logical connectives are not_ for negation, _and_ for conjunction, _or_ for disjunction, _implies_ for implication, _iff_ for equivalence and _xor_ for exclusive disjunction, where underscores indicate places where arguments are put. One operator _=_ is declared in PID, which is a user-defined equality predicate[1]. Operators can be given attributes such as comm that specifies that the binary operator is commutative. Four variables, together with their sorts, are declared in PID. We have three equations in PID. Each variable in an equation is universally quantified and its scope is in the equation. The first equation says that any process ID is different from any error ID in terms of _=_. The second equation says that any ID equals itself in terms of _=_. The third equation has the label o2m-Error&Pid and a condition. Equations with conditions are called conditional equations. The third equation says that

[1] The same symbol = is used as an operation name (the user-defined equality predicate) and a language construct of which equations are made. In the last equation in PID, the first occurrence of = is the language construct, while the second occurrence of = is the user-defined equality predicate. _=_ is used to refer to the user-defined equality predicate.

CafeOBJ provides the built-in equality predicate _==_ for each sort. The built-in predicate works well for ground terms (terms without variables) in a specification that is both terminating and confluent. Fresh constants are often added to specifications in writing proof scores in CafeOBJ, making specifications non-confluent. This is why we need to have user-defined equality predicates for some sorts. Systematic ways to define such user-defined equality predicates have been proposed[Nakamura and Futatsugi, 2007, Gutiérrez et al, 2012].

if two IDs are equal in terms of `_=_`, then they are really equal. Since `PID` is a loose module, it accepts multiple implementations. One implementation can use natural numbers, where zero is used for `Error` and non-zero numbers for `Pid`. Another implementation can use strings, where the null string is used for `Error` and non-null strings for `Pid`.

Queues of process IDs are specified as the following module:

```
mod! QUEUE {
  pr(PID)
  [Empty NeQueue < Queue]
  op empty : -> Empty {constr}
  op _|_ : Pid Queue -> NeQueue {constr}
  op _=_ : Queue Queue -> Bool {comm}
  op top : Queue -> Error&Pid
  op top : Empty -> Error
  op top : NeQueue -> Pid
  op enq : Queue Pid -> NeQueue
  op deq : Queue -> Queue
  op _\in_ : Pid Queue -> Bool
  op _-_ : Queue Pid -> Queue
  vars I J : Pid
  vars Q Q1 Q2 : Queue
  eq (Q = Q) = true .
  eq (empty = J | Q2) = false .
  eq (I | Q1 = J | Q2) = (I = J) and (Q1 = Q2) .
  ceq [o2m-Queue]: Q1 = Q2 if Q1 = Q2 .
  eq top(I | Q) = I .
  eq enq(empty,I) = I | empty .
  eq enq(J | Q,I) = J | enq(Q,I) .
  eq deq(empty) = empty .
  eq deq(I | Q) = Q .
  eq I \in empty = false .
  eq I \in (J | Q) = (I = J) or (I \in Q) .
  eq empty - I = empty .
  eq (J | Q) - I = (if J = I then Q else J | (Q - I) fi) .
}
```

The module `QUEUE` explicitly imports the module `PID`. `Empty` corresponds to the set of the empty queue, and `NeQueue` to the set of non-empty queues. The constant `empty` and the operator `_|_` are the constructors of queues as the attribute `constr` indicates. Given three process IDs i, j, k, the term "i `|` j `|` k `| empty`" denotes the queue that consists of i, j, k such that i is the first element and k is the last element. The operator `_=_` is a user-defined equality predicate for queues. The three declarations for `top` say that `top` returns an error ID for the empty queue, a process ID for a non-empty queue, and either a process ID or an error ID for a queue. The three operators `top`, `enq` and `deq` denote the functions for returning the top element in a given queue if any, enqueuing and dequeuing, respectively. The operator `_\in_` denotes the membership predicate of queues. The operator `_-_` denotes the function that takes a queue and an element, and deletes one occurrence of the element nearest to the top from the queue if any. The non-constructor operators, or some properties of the operators are defined in equations.

The CafeOBJ processor uses equations as left-to-right rewrite rules to compute (or reduce) a given term. Let us take a look at the following CafeOBJ code:

```
open QUEUE
  ops i j k : -> Pid .
  eq (i = j) = false .
  eq (i = k) = false .
  eq (j = k) = false .
  red (i | j | empty) = (i | k | empty) .
  red top(i | j | k | empty) .
  red enq(i | j | k | empty, j) .
  red deq(i | j | k | empty) .
  red j \in (i | j | k | empty) .
  red (i | j | k | j | empty) - j .
close
```

The command `open` makes a given module available, and the command `close` declares the end of the use of the module. The command `red` reduces a given term with the equations available. The six `red` commands return `false`, `i`, `i | j | k | j | empty`, `j | k | empty`, `true`, and `i | k | j | empty`.

2.2 Verification

Lists of process IDs are specified as the following module:

```
mod! LIST {
  pr(PID)
  [List]
  op nil : -> List {constr}
  op __ : Pid List -> List {constr}
  op _=_ : List List -> Bool {comm}
  op _@_ : List List -> List {assoc}
  op rev : List -> List
```

```
  vars L L1 L2 : List
  vars P P1 P2 : Pid
  eq (L = L) = true .
  eq (P1 L1 = P2 L2) = (P1 = P2) and (L1 = L2) .
  eq L @ nil = L .
  eq nil @ L = L .
  eq (P1 L1) @ L = P1 (L1 @ L) .
  eq rev(nil) = nil .
  eq rev(P L) = rev(L) @ (P nil) .
}
```

The constant `nil` and the juxtaposition operator `__` are the constructors of lists. The operators `_@_` and `rev` denote the concatenation function and the reverse function of lists, respectively. The attribute `assoc` is given to `_@_`, specifying that the operator is associative.

Let us prove a property on `rev` that for all lists l, `rev(rev(`l`))` equals l. The property is described in the following module:

```
mod! REVREV-LIST {
  pr(LIST)
  op th1 : List -> Bool
  var L : List
  eq th1(L) = (rev(rev(L)) = L) .
}
```

The proof is conducted by case analysis on `L`: (1) `nil` and (2) `p l`, where `p` is an arbitrary process ID and `l` is an arbitrary list. The proof (called a *proof score*) of the property in CafeOBJ is as follows:

```
open REVREV-LIST
-- check
  red th1(nil) .
close
open REVREV-LIST
-- fresh constants
  op p : -> Pid .
  op l : -> List .
-- check
  red lem1(l,p) implies th1(p l) .
close
```

A comment starts with double hyphens “`--`” and continues by the end of the line. The proof uses the lemma `lem1` declared and defined in the module `REVREV-LIST` as follows:

```
op lem1 : List Pid -> Bool
eq lem1(L,P) = (rev(rev(L) @ (P nil)) = P L) .
```

Since `th1(p l)` reduces to `rev(rev(l) @ (p nil)) = p l` in the `open-close` fragment (called a *proof passage*), the lemma is straightforwardly conjectured.

If we try to prove `lem1` by induction on `L`, we need to use a series of lemmas that are similar to `lem1`. This is why we need to generalize `lem1`. A generalized version of `lem1` is as follows:

```
op lem2 : List List -> Bool
eq lem2(L,L1) = (rev(rev(L) @ L1) = rev(L1) @ L) .
```

`lem1` can be deduced from `lem2` as follows:

```
open REVREV-LIST
-- fresh constants
  op p : -> Pid .
  op l : -> List .
-- check
  red lem2(l,p nil) implies lem1(l,p) .
close
```

The proof of `lem2` is conducted by induction on `L` as follows:

```
open REVREV-LIST
-- fresh constants
  op l1 : -> List .
-- check
  red lem2(nil,l1) .
close
open REVREV-LIST
-- fresh constants
  op p : -> Pid .
  ops l l1 : -> List .
-- check
  red lem2(l,p l1) implies lem2(p l,l1) .
close
```

where `lem2(l,p l1)` is an instance of the induction hypothesis ($\forall$`L1 : List`) $\texttt{lem2}(\texttt{l}, \texttt{L1})$.

Although `lem2` could be directly used in the proof of the property, the use of `lem1` has some benefit in that we can record the process in which way lemmas have been conjectured. Semi-simultaneous induction makes it possible to use this benefit in a more general setting.

3 Observational Transition Systems (OTSs)

We describe OTSs and how to specify an OTS as a behavioral specification in CafeOBJ by using a mutual exclusion protocol called Qlock in this section.

3.1 Some Definitions

We suppose that there exists a universal state space denoted by Υ and that each data type used in OTSs is provided. Υ is the set of all states. In this article, a set may be regarded as the type corresponding to the set. The data types include Bool for truth values. A data type is expressed as D with a subscript such as D_{o1} and D_o.

Definition 1 OTSs. An observational transition system (OTS) consists of

- $\mathcal{O}$: A set of *observers*. Each observer is a function $o : \Upsilon\ D_{o1} \ldots D_{om} \rightarrow D_o$. The equivalence between two states υ_1, υ_2 (denoted as $\upsilon_1 =_{\mathcal{S}} \upsilon_2$) is defined wrt values returned by the observers: $\upsilon_1 =_{\mathcal{S}} \upsilon_2$ if and only if (iff) for each $o : \mathcal{O}$, $o(\upsilon_1, x_1, \ldots, x_m) = o(\upsilon_2, x_1, \ldots, x_m)$ for all $x_1 : D_{o1}, \ldots, x_m : D_{om}$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A set of *transitions*. Each transition is a function $t : \Upsilon\ D_{t1} \ldots D_{tn} \rightarrow \Upsilon$. Each transition t, together with any other parameters $y_1, \ldots, y_n$, preserves the equivalence between two states: if $\upsilon_1 =_{\mathcal{S}} \upsilon_2$, then for each $t : \mathcal{T}$, $t(\upsilon_1, y_1, \ldots, y_n) =_{\mathcal{S}} t(\upsilon_2, y_1, \ldots, y_n)$ for all $y_1 : D_{t1}, \ldots, y_n : D_{tn}$. Each t has the *effective condition* c-$t : \Upsilon\ D_{t1} \ldots D_{tn} \rightarrow$ Bool. If $\neg c$-$t(\upsilon, y_1, \ldots, y_n)$, then $t(\upsilon_1, y_1, \ldots, y_n) =_{\mathcal{S}} \upsilon$. $t(\upsilon, y_1, \ldots, y_n)$ is called a *successor state* of a state υ. We write $\upsilon \rightsquigarrow_{\mathcal{S}} \upsilon'$ iff a state $\upsilon' \in \Upsilon$ is a successor state of a state $\upsilon \in \Upsilon$.

Definition 2 Reachable states. *Reachable states* wrt an OTS $\mathcal{S}$ are inductively defined as follows:

- Each initial state $\upsilon_0 \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.
- If a state υ is reachable wrt $\mathcal{S}$, so is each successor state of υ.

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

Definition 3 Invariants. A state predicate $p : \Upsilon \rightarrow$ Bool is called an *invariant* wrt an OTS $\mathcal{S}$ if p holds in all reachable states wrt $\mathcal{S}$, namely $(\forall \upsilon : \mathcal{R}_{\mathcal{S}}) p(\upsilon)$.

Note that a state predicates is mainly constructed of observers because states in OTSs are characterized by the values returned by observers in those states.

Definition 4 Inductive invariants. A state predicate $p : \Upsilon \rightarrow$ Bool is called an *inductive invariant* wrt an OTS $\mathcal{S}$ if $p(\upsilon_0)$ holds for each initial state $\upsilon_0 \in \mathcal{I}$ and $p(\upsilon) \Rightarrow p(\upsilon')$ holds for each state $\upsilon \in \Upsilon$ and each successor state of υ.

Note that inductive invariants wrt an OTS $\mathcal{S}$ are invariants wrt $\mathcal{S}$.

3.2 Specification of OTSs in CafeOBJ

CafeOBJ is used to specify OTSs. Υ is denoted by a sort, say `Sys`. Each $o \in \mathcal{O}$ is denoted by an operator (called an *observation operator*) declared as follows:

$$\texttt{bop o : Sys } \texttt{D}_{o1} \ldots \texttt{D}_{om} \texttt{ -> } \texttt{D}_{o}$$

where each $\texttt{D}_*$ is a sort corresponding to D_*.

An arbitrary initial state in $\mathcal{I}$ is denoted by an operator declared as follows:

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

For each $o \in \mathcal{O}$, the following equation is declared:

$$\texttt{eq o(init,X}_1\texttt{,...,X}_m\texttt{) = } T^{\circ}_{\texttt{X}_1,\ldots,\texttt{X}_m} \texttt{ .}$$

where each $\texttt{X}_*$ is a CafeOBJ variable of the sort $\texttt{D}_*$ and $T^{\circ}_{\texttt{X}_1,\ldots,\texttt{X}_m}$ is a term denoting the value returned by o, together with any other parameters, in an arbitrary initial state[2]

Each $t \in \mathcal{T}$ is denoted by an operator (called a *transition operator*) declared as follows:

$$\texttt{bop t : Sys } \texttt{D}_{t1} \ldots \texttt{D}_{tn} \texttt{ -> Sys \{constr\}}$$

For each `o` and `t`, a conditional equation is declared:

$$\texttt{ceq o(t(S,Y}_1\texttt{,...,Y}_n\texttt{),X}_1\texttt{,...,X}_m\texttt{) = o-t}_{\texttt{S},\texttt{Y}_1,\ldots,\texttt{Y}_n,\texttt{X}_1,\ldots,\texttt{X}_m}$$
$$\texttt{if c-t(S,Y}_1\texttt{,...,Y}_n\texttt{) .}$$

where `c-t(S,...)` corresponds to $c\text{-}t(v,\ldots)$, and $\texttt{o-t}_{\texttt{S},\ldots}$ is a term whose sort is the same as the sort of `o` and does not use any transition operators[3]. The equation says how t changes the value observed by o if the effective condition holds. If $\texttt{o-t}_{\texttt{S},\ldots}$ is always equal to $\texttt{o(S,X}_1\texttt{,...,X}_m\texttt{)}$, the condition may be omitted.

For each `t`, one more conditional equation is declared:

$$\texttt{ceq t(S,Y}_1\texttt{,...,Y}_n\texttt{) = S if not c-t(S,Y}_1\texttt{,...,Y}_n\texttt{) .}$$

[2] Although an arbitrary initial state is typically denoted by a constant, it may be denoted by a non-constant operator. The value returned by o in an arbitrary initial state may be defined by case analysis. For example, when $m = 1$ and $\texttt{D}_{o1}$ is `Queue`, the value returned by o is defined with the equations that look like `eq o(init,empty) =` $\cdots$ `.` and `eq o(init,P | Q) =` $\cdots$ `.`, where `P` and `Q` are variables of `Pid` and `Queue`. If you use an auxiliary operator declared as `op aux-o : Queue ->` $\texttt{D}_o$ and defined as `eq aux-o(empty) =` $\cdots$ `.` and `eq aux-o(P | Q) =` $\cdots$ `.`, however, the same effect can be obtained.

[3] The operator `c-t` should be well-defined, namely that for all ground terms $s, y_1, \ldots, y_n$, `c-t(`s`,`y_1`,...,`y_n`)` must reduce to either `true` or `false`.

which says that t changes nothing if the effective condition does not hold.

As indicated by `constr`, `init` and each `t` are constructors of `Sys`[4], which corresponds to $\mathcal{R}_{\mathcal{S}}$.

An example is used to describe how to specify an OTS in CafeOBJ. The example used is a mutual exclusion protocol called Qlock.

Example 1 Qlock. The pseudo-code executed by each process i can be written as follows:

Loop
Remainder Section
rs: $enq(queue, i)$
ws: **repeat until** $top(queue) = i$
Critical Section
cs: $deq(queue)$

where $queue$ is the queue of process IDs shared by all processes. enq, top and deq are the functions of queues for enqueuing, returning the top element in a given queue if any, and dequeuing, respectively. The body (between **repeat** and `until`) of the loop at label ws is empty. Initially, each process i is at the label rs and $queue$ is empty. Let Label, Pid and Queue be the types of labels (rs, ws and cs), process IDs and queues of process IDs, respectively. We suppose that $queue$ is never used in Remainder Section and Critical Section.

On the assumption that each of the two statements at the labels rs and cs is atomically executed and each iteration of the loop at the label ws is atomically executed, Qlock is formalized as an OTS $\mathcal{S}_{\text{Qlock}}$. $\mathcal{S}_{\text{Qlock}}$ uses two observers. The corresponding observation operators are declared as follows:

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

Given a state s : `Sys` and a process ID i : `Pid`, `pc(`s`,`i`)` denotes the label at which the process i is in the state s and `queue(`s`)` denotes the shared queue in the state s.

In the rest of this section, let `I`, `J` and `S` be CafeOBJ variables of `Pid`, `Pid` and `Sys`. We have the following two equations for `init`, the constant denoting an arbitrary initial state:

```
eq pc(init,I)  = rs .
eq queue(init) = empty .
```

[4] `Sys` denotes $\mathcal{R}_{\mathcal{S}}$ but not Υ if the constructor-based logic[Gâinâ et al, 2009] is adopted, which is the current logic underlying the OTS/CafeOBJ method. In this setting, properties that we are interested in are only safety ones, especially invariants.

$\mathcal{S}_{\mathrm{Qlock}}$ uses three transitions. The corresponding transition operators are declared as follows:

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

Given a state s : `Sys` and a process ID i : `Pid`, `want(`s`,`i`)` denotes the successor state of s when i executes the statement at the label rs in s, `try(`s`,`i`)` denotes the successor state of s when i executes one iteration of the loop at the label ws in s, and `exit(`s`,`i`)` denotes the successor state of s when i executes the statement at the label cs in s.

The set of equations for `want` is as follows:

```
ceq pc(want(S,I),J)
    = (if I = J then ws else pc(S,J) fi) if c-want(S,I) .
ceq queue(want(S,I)) = enq(queue(S),I) if c-want(S,I) .
ceq want(S,I)        = S if not c-want(S,I) .
```

where the operator `c-want` is declared and defined as follows:

```
op c-want : Sys Pid -> Bool
eq c-want(S,I) = (pc(S,I) = rs) .
```

The set of equations for `try` is as follows:

```
ceq pc(try(S,I),J)
    = (if I = J then cs else pc(S,J) fi) if c-try(S,I) .
 eq queue(try(S,I)) = queue(S) .
ceq try(S,I)        = S if not c-try(S,I) .
```

where the operator `c-try` is declared and defined as follows:

```
op c-try : Sys Pid -> Bool
eq c-try(S,I) = (pc(S,I) = ws and top(queue(S)) = I) .
```

The set of equations for `exit` is as follows:

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

where the operator `c-exit` is declared and defined as follows:

```
op c-exit : Sys Pid -> Bool
eq c-exit(S,I) = (pc(S,I) = cs) .
```

One of the properties that Qlock should enjoy is the mutual exclusion property, which can be expressed as an invariant wrt $\mathcal{S}_{\mathrm{Qlock}}$. The state predicate concerned is denoted by the operator declared and defined as follows:

```
op inv1 : Sys Pid Pid -> Bool
eq inv1(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs implies I = J) .
```

To verify that Qlock enjoys the mutual exclusion property, all we have to do is to prove that $(\forall i, j : \texttt{Pid})\, \texttt{inv1(}s\texttt{,}i\texttt{,}j\texttt{)}$ is an invariant wrt $\mathcal{S}_{\mathrm{Qlock}}$.

We suppose that $\mathcal{S}_{\mathrm{Qlock}}$ is specified as a module `QLOCK`, and operators denoting state predicates such as `inv1` are declared in a module `PRED-QLOCK` that imports `QLOCK`.

4 Proof Scores of Invariants

In the OTS/CafeOBJ method, invariants are proved by writing proof scores in CafeOBJ and executing them with its processor. We use the proof of $(\forall s : \texttt{Sys})\,(\forall i, j : \texttt{Pid})\, \texttt{inv1(}s\texttt{,}i\texttt{,}j\texttt{)}$ as an example to describe proof scores in the OTS/CafeOBJ method.

4.1 Proof Scores by Simultaneous Induction

We suppose that we start to prove the formula by structural induction on s. In the induction case where `try` is taken into account, the case is divided into five sub-cases as follows:

1. `c-try(s,k) = true`, `i = k`, `j = k`
2. `c-try(s,k) = true`, `i = k`, `(j = k) = false`
3. `c-try(s,k) = true`, `(i = k) = false`, `j = k`
4. `c-try(s,k) = true`, `(i = k) = false`, `(j = k) = false`
5. `c-try(s,k) = false`

where `s`, `k`, `i` and `j` are constants of `Sys`, `Pid`, `Pid` and `Pid`, respectively. `s` is used to denote an arbitrary state, and `k`, `i` and `j` arbitrary process IDs. Sub-cases 2 and 3 need to use another state predicate as an induction hypothesis. Another state predicate is denoted by the operator declared and defined as follows:

```
op inv2 : Sys Pid -> Bool
eq inv2(S,I) = (pc(S,I) = cs implies top(queue(S)) = I) .
```

This state predicate is discovered while the proof passage corresponding to sub-case 2 is being written [Ogata and Futatsugi, 2006].

The proof passage is as follows:

```
open ISTEP-QLOCK
-- fresh constants
  op k : -> Pid .
  op q : -> Queue .
-- assumptions
  -- eq c-try(s,k) = true .
  eq pc(s,k) = ws .
  eq queue(s) = k | q .
  --
  eq i = k .
  eq (j = k) = false .
-- successor state
  eq s' = try(s,k) .
-- check
  red inv2(s,j) implies istep1 .
close
```

where `ISTEP-QLOCK` is a module in which another module `BASE-QLOCK` is imported and some constants such as `istep1` are declared and defined. In `BASE-QLOCK`, `PRED-QLOCK` is imported and the following constants are declared:

```
ops s s' : -> Sys
ops i j k : -> Pid
```

They are used to denote arbitrary states and process IDs.

The constant `q` is used to denote an arbitrary queue. Instead of `c-try(s,k) = true`, we use `pc(s,k) = ws` and `queue(s) = k | q`, which are equivalent to `c-try(s,k) = true`. The constant `istep1` of `Bool` is defined as follows:

```
eq istep1 = inv1(s,i,j) implies inv1(s',i,j) .
```

where `inv1(s',i,j)` denotes the formula to prove in the induction case, and `inv1(s,i,j)` and `inv2(s,j)` denote instances of the induction hypotheses used. In this proof passage, `s'` is `try(s,k)`. The CafeOBJ processor returns `true` for the proof passage, discharging the sub-case. It also returns `true` for the remaining four sub-cases, discharging the induction case where `try` is taken into account. We can write proof passages of the base case and the remaining induction cases where `want` and `exit` are taken into account, which do not need any other state predicates.

We need to prove $(\forall s : \texttt{Sys})\,(\forall i : \texttt{Pid})\,\texttt{inv2}(s,i)$ to complete the proof. $(\forall s : \texttt{Sys})\,(\forall i : \texttt{Pid})\,\texttt{inv2}(s,i)$ is also proved by structural induction on s. In the induction case where `exit` is taken into account, the case is divided into three sub-cases as follows:

1. `c-exit(s,k) = true`, `i = k`
2. `c-exit(s,k) = true`, `(i = k) = false`
3. `c-exit(s,k) = false`

Sub-case 2 needs to use `inv1` as an induction hypothesis.

The proof passage corresponding to sub-case 2 is as follows:

```
open ISTEP-QLOCK
-- fresh constants
  op k : -> Pid .
-- assumptions
  -- eq c-exit(s,k) = true .
  eq pc(s,k) = cs .
  --
  eq (i = k) = false .
-- successor state
  eq s' = exit(s,k) .
-- check
  red inv1(s,i,k) implies istep2 .
close
```

The constant `istep2` of `Bool` is defined as follows:

```
eq istep2 = inv2(s,i) implies inv2(s',i) .
```

The CafeOBJ processor returns `true` for the proof passage, discharging the sub-case. It also returns `true` for the remaining two sub-cases, discharging the induction case where `exit` is taken into account. We can write proof passages of the base case and the remaining induction cases where `want` and `try` are taken into account.

When we notice that the proof score wrt `inv1` needs `inv2`, we do not need to start to write the proof score wrt `inv2` before the proof score wrt `inv1`, nor to throw away the proof score wrt `inv1` constructed so far. We can complete the proof score wrt `inv1` and then start to write the proof score wrt `inv2`. Simultaneous induction makes it possible to compositionally (or incrementally) construct proof scores in this way. This is one benefit of simultaneous induction.

The proof scores described correspond to Fig. 1 (1).

4.2 Proof Scores by Semi-Simultaneous Induction

We notice that $(\forall s : \texttt{Sys})\,(\forall i, j : \texttt{Pid})\,\texttt{inv1}(s,i,j)$ can be deduced from $(\forall s : \texttt{Sys})\,(\forall i : \texttt{Pid})\,\texttt{inv2}(s,i)$ under the specification of $\mathcal{S}_{\mathrm{Qlock}}$. Hence, we do not need to use structural induction on s to write the proof score of $(\forall s : \texttt{Sys})\,(\forall i, j :$

`Pid`) `inv1(`s`,`i`,`j`)`. It suffices to use case analysis to prove the formula. In the proof, all we have to do is to take into account three cases:

1. `queue(s) = empty`
2. `queue(s) = k | q`, `k = i`
3. `queue(s) = k | q`, `(k = i) = false`

The proof passage of case 2 is as follows:

```
open BASE-QLOCK
-- fresh constants
  op k : -> Pid .
  op q : -> Queue .
-- assumptions
  eq queue(s) = k | q .
  eq k = i .
-- check
  red inv2(s,i) and inv2(s,j) implies inv1(s,i,j) .
close
```

The CafeOBJ processor returns `true` for this proof passage, and does so for the remaining two cases as well.

The proof score wrt `inv2` described in the previous sub-section can be used here. The proof scores correspond to Fig. 1 (2).

4.3 Proof Scores by the INV Rule

`inv2` can be proved without use of `inv1` as an induction hypothesis because `inv1` can be deduced from `inv2` under the specification of $\mathcal{S}_{\mathrm{Qlock}}$. If that is the case, in the induction case where `exit` is taken into account, the case is divided into five sub-cases as follows:

1. `c-exit(s,k) = true`, `i = k`
2. `c-exit(s,k) = true`, `(i = k) = false`, `queue(s) = empty`
3. `c-exit(s,k) = true`, `(i = k) = false`, `queue(s) = l | q`, `l = k`
4. `c-exit(s,k) = true`, `(i = k) = false`, `queue(s) = l | q`, `(l = k) = false`
5. `c-exit(s,k) = false`

The proof passage of sub-case 4 is as follows:

```
open ISTEP-QLOCK
-- fresh constants
  ops k l : -> Pid .
  op q : -> Queue .
-- assumptions
  -- eq c-exit(s,k) = true .
  eq pc(s,k) = cs .
  --
  eq (i = k) = false .
  eq queue(s) = l | q .
  eq (l = k) = false .
-- successor state
  eq s' = exit(s,k) .
-- check
  red inv2(s,k) implies istep2 .
close
```

In addition to `inv2(s,i)`, `inv2(s,k)` is used as an instance of the induction hypothesis $(\forall i : \texttt{Pid})\, \texttt{inv2(s,}i\texttt{)}$.

The proof score wrt `inv1` described in the previous sub-section can be used here. The proof scores correspond to Fig. 1 (3).

4.4 Comparison

The three proof techniques are compared in terms of how long the corresponding proofs (or proof scores) are based on the example. Let PS1 be the proof score of

$(\forall s : \texttt{Sys})\, (\forall i : \texttt{Pid})\, \texttt{inv2(}s\texttt{,}i\texttt{)} \Rightarrow (\forall s : \texttt{Sys})\, (\forall i, j : \texttt{Pid})\, \texttt{inv1(}s\texttt{,}i\texttt{,}j\texttt{)}$,

PS2 be that of

$(\forall i, j : \texttt{Pid})\, \texttt{inv1(init,}i\texttt{,}j\texttt{)}$ and
$(\forall i, j : \texttt{Pid})\, \texttt{inv1(s,}i\texttt{,}j\texttt{)} \wedge (\forall i : \texttt{Pid})\, \texttt{inv2(s,}i\texttt{)} \Rightarrow (\forall i, j : \texttt{Pid})\, \texttt{inv1(s',}i\texttt{,}j\texttt{)}$

for an arbitrary initial state `init`, an arbitrary state `s` and an arbitrary successor state `s'` of `s`, PS3 be that of

$(\forall i : \texttt{Pid})\, \texttt{inv2(init,}i\texttt{)}$ and
$(\forall i, j : \texttt{Pid})\, \texttt{inv1(s,}i\texttt{,}j\texttt{)} \wedge (\forall i : \texttt{Pid})\, \texttt{inv2(s,}i\texttt{)} \Rightarrow (\forall i : \texttt{Pid})\, \texttt{inv2(s',}i\texttt{)}$

for an arbitrary initial state `init`, an arbitrary state `s` and an arbitrary successor state `s'` of `s`, and PS4 be that of

$(\forall i : \texttt{Pid})\, \texttt{inv2(init,}i\texttt{)}$ and
$(\forall i : \texttt{Pid})\, \texttt{inv2(s,}i\texttt{)} \Rightarrow (\forall i : \texttt{Pid})\, \texttt{inv2(s',}i\texttt{)}$

for an arbitrary initial state `init`, an arbitrary state `s` and an arbitrary successor state `s'` of `s`.

The proof scores by semi-simultaneous induction are shorter than those by the INV rule, which are shorter than those by simultaneous induction. The reason of the latter is that PS1 is shorter than PS2. The reason of the former is that PS3 is shorter than PS4 because the induction case where `exit` is taken into account is split into five sub-cases in PS4, while the case into three sub-cases in PS3. This example demonstrates that the proof corresponding to Fig. 1 (2) is equivalent to the one to Fig. 1 (3) but may need less cases, making proofs (or proof scores) shorter, than the one to Fig. 1 (3).

Why does semi-simultaneous induction need less cases than the INV rule in the induction case where `exit` is taken into account in the proof wrt `inv2`? Although $(\forall s : \texttt{Sys})\,(\forall i : \texttt{Pid})\,\texttt{inv2}(s,i)$ is equivalent to $(\forall s : \texttt{Sys})\,(\forall i,j : \texttt{Pid})\,\texttt{inv1}(s,i,j) \wedge (\forall s : \texttt{Sys})\,(\forall i : \texttt{Pid})\,\texttt{inv2}(s,i)$ under the specification of $\mathcal{S}_{\mathrm{Qlock}}$, its proof needs case splitting. Hence, the two formulas reduce to different irreducible forms in some cases. `inv1(s,i,k)` reduces to `pc(s,i) = cs xor true` in Sub-case 2 in Subsection 4.1 and then `inv1(s,i,k) implies inv2(exit(s,k),i)` reduces to `true`, discharging the case. On the other hand, `inv2(s,j)` reduces to `top(queue(s)) = k` if `j` equals `k` and to `(pc(s,j) = cs and top(queue(s)) = j) xor pc(s,j) = cs xor true` otherwise in the case. Any instance of $(\forall i : \texttt{Pid})\,\texttt{inv2}(\texttt{s},i)$ does not lead to the discharge of the case by reduction only.

5 Simultaneous Induction

The proof that a state predicate p_0 is an invariant wrt an OTS $\mathcal{S}$ can be done by finding an inductive invariant wrt $\mathcal{S}$ that implies p_0 under $\mathcal{S}$. This is the INV rule [Manna and Pnueli, 1995]. Such an inductive invariant can be found by connecting other state predicates $p_1, \ldots, p_n$ to p_0 with conjunctions, namely, making $p_0 \wedge p_1 \wedge \ldots \wedge p_n$. Let q be $p_0 \wedge p_1 \wedge \ldots \wedge p_n$. To prove that q is an inductive invariant wrt $\mathcal{S}$, all we have to do is to prove (1) $q(\upsilon_0)$ holds for each initial state $\upsilon_0 \in \mathcal{I}$ and (2) $q(\upsilon) \Rightarrow q(\upsilon')$ holds for each state $\upsilon, \upsilon' \in \Upsilon$ such that $\upsilon \leadsto_{\mathcal{S}} \upsilon'$.

The proof of (1) and (2) can be compositionally written, divided into $n+1$ fragments wrt $p_0, p_1, \ldots, p_n$, respectively. That is, all we have to do is to prove that

1. $p_i(\upsilon_0)$ holds for $\mathcal{S}$ for each $\upsilon_0 \in \mathcal{I}$, and
2. $(\bigwedge_{h \in \mathcal{H}_i} h(\upsilon)) \Rightarrow p_i(\upsilon')$ holds for $\mathcal{S}$ for each $\upsilon, \upsilon' \in \Upsilon$ such that $\upsilon \leadsto_{\mathcal{S}} \upsilon'$, where $\mathcal{H}_i \subseteq \{p_0, \ldots, p_n\}$,

for each $i \in \{0, \ldots, n\}$. This way of writing proofs is called simultaneous induction [Ogata and Futatsugi, 2003, Ogata and Futatsugi, 2006, Diaconescu et al, 2003]. The proof of 1 and 2 is called the *SI proof* of p_i wrt $\mathcal{S}$ based on $\{p_0, \ldots, p_n\}$. It may be called the SI proof of p_i if $\mathcal{S}$ and $\{p_0, \ldots, p_n\}$ are clear from the context. 1 is called the base case of the SI proof, and 2 is called the (simultaneous) induction case of the SI proof. Each $h(\upsilon)$ such that $h \in \mathcal{H}_i$ is called a *simultaneous induction hypothesis* for the SI proof of p_i. Note that only the SI proof of p_i does not necessarily imply that p_i is an invariant wrt $\mathcal{S}$, although all SI proofs of $p_0, \ldots, p_n$ imply that each p_i is an invariant wrt $\mathcal{S}$.

6 Semi-Simultaneous Induction

Let $\mathcal{P}$ and $\mathcal{Q}$ be the sets of state predicates such that $\mathcal{P} \cup \mathcal{Q} = \{p_0, \ldots, p_n\}$ and $\mathcal{P} \cap \mathcal{Q} = \emptyset$. We assume that

1. there is the SI proof of each state predicate in $\mathcal{P}$ wrt $\mathcal{S}$ based on $\{p_0, \ldots, p_n\}$, and
2. for each state predicate $q \in \mathcal{Q}$ and some $\mathcal{R}$ such that $\mathcal{R} \subseteq (\mathcal{P} \cup \mathcal{Q}) \setminus \{q\}$, $q(\upsilon)$ can be deduced from $(\bigwedge_{r \in \mathcal{R}} r(\upsilon))$ under $\mathcal{S}$ for each $\upsilon \in \Upsilon$.

The proof of 2 is called the *IMP(lication) proof* of q wrt $\mathcal{S}$ based on $\{p_0, \ldots, p_n\}$. It may be called the IMP(lication) proof of q when $\mathcal{S}$ and $\{p_0, \ldots, p_n\}$ are clear from the context. The state predicates in $\mathcal{R}$ are called *premises* of the IMP proof.

From the assumptions 1 and 2, are we able to conclude that $p_0 \wedge \ldots \wedge p_n$ is an inductive invariant wrt $\mathcal{S}$? To do so, we need a condition. Before describing the condition, we define a graph that represents 1 and 2.

Definition 5 PII graphs. A *pseudo inductive invariant (PII) graph* of $p_0 \wedge \ldots \wedge p_n$ is as follows. The nodes of the PII graph are $p_0, \ldots, p_n$. There are two kinds of edges in the graph: *SIH edges* and *IMP edges*. For each $p_i \in \{p_0, \ldots, p_n\}$, p_i has one kind (either SIH or IMP) of outgoing edges but can have both kinds of incoming edges.

That p_i has SIH (IMP) outgoing edges means that there exists the SI (IMP) proof of p_i, which uses the destination state predicates of the SIH (IMP) edges as the simultaneous induction hypotheses (premises). If p_i has an SIH (IMP) incoming edge, p_i is used as a simultaneous induction hypothesis (a premise) of the SI (IMP) proof of the source state predicate. Nodes that have SIH (IMP) outgoing edges are called *SIH (IMP) nodes*. Nodes in $\mathcal{P}$ are SIH nodes and nodes in $\mathcal{Q}$ are IMP nodes.

The condition concerned is as follows.

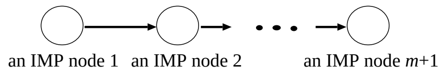

Figure 2: A path where IMP nodes occurs $m + 1$ times

Definition 6 II condition. The *inductive invariant (II) condition* of a PII graph is that the graph has no cycles such that every edge in a cycle is an IMP edge.

Before describing the main theorem, we prove the following lemma.

Lemma 7 A property of PII graphs. *If a PII graph G of $p_0 \wedge \ldots \wedge p_n$ satisfies the II condition and the number m of the IMP nodes in G is greater than zero, then there exists an IMP node such that the destinations of all the IMP outgoing edges are SIH nodes.*

Proof. By contradiction. We suppose that G does not have any IMP nodes such that the destinations of all the IMP outgoing edges are SIH nodes. Then, each IMP node has an IMP node as one of the destinations of the IMP outgoing edges. Hence, we can find a path in G such that it consists of IMP nodes only and IMP nodes occurs in it $m + 1$ times (see Fig. 2). Since the number of the IMP nodes in G is m, there exists an IMP node in the path that occurs at least twice. Therefore, G has a cycle that consists of IMP nodes only. $\square$

The following theorem guarantees that if a PII graph of $p_0 \wedge \ldots \wedge p_n$ has no such cycles, $p_0 \wedge \ldots \wedge p_n$ is an inductive invariant wrt $\mathcal{S}$.

Theorem 8 II condition of PII graphs. *If a PII graph G of $p_0 \wedge \ldots \wedge p_n$ satisfies the II condition, then $p_0 \wedge \ldots \wedge p_n$ is an inductive invariant wrt $\mathcal{S}$.*

Proof. By induction on the number of the IMP nodes in the PII graph G.

(I) For the base case, since the SI proof of every p_i wrt $\mathcal{S}$ based on $\{p_0, \ldots, p_n\}$ for $i = 0, \ldots, n$ can be conducted, $p_0 \wedge \ldots \wedge p_n$ is an inductive invariant wrt $\mathcal{S}$.

(II) For the induction case, since G satisfies the II condition, from Lemma 7, there exists an IMP node q such that the destinations $r_0, \ldots, r_m$ of all the IMP outgoing edges are SIH nodes. Since the SI proof of each r_j wrt $\mathcal{S}$ based on $\{p_0, \ldots, p_n\}$ for $j = 0, \ldots, m$ can be conducted, we can show that the following formulas hold for $\mathcal{S}$:

$$r_0(v_0), \ldots, r_m(v_0) \tag{1}$$

for each $v_0 \in \mathcal{I}$, and

$$(\bigwedge_{a \in A_0} a(v)) \Rightarrow r_0(v'), \ldots, (\bigwedge_{a \in A_m} a(v)) \Rightarrow r_m(v') \tag{2}$$

for each $\upsilon, \upsilon' \in \Upsilon$ such that $\upsilon \leadsto_{\mathcal{S}} \upsilon'$ and for some $\mathcal{A}_j \subseteq \{p_0, \ldots, p_n\}$ for $j = 0, \ldots, m$. Let $\mathcal{A}$ be $\mathcal{A}_0 \cup \ldots \cup \mathcal{A}_m$.

Since the IMP proof of q wrt $\mathcal{S}$ based on $\{p_0, \ldots, p_n\}$ can be conducted, we can also show that the following formula hods for $\mathcal{S}$:

$$r_0(\upsilon) \wedge \ldots \wedge r_m(\upsilon) \Rightarrow q(\upsilon) \tag{3}$$

for each $\upsilon \in \Upsilon$.

From (1) and (3), the following formula holds for $\mathcal{S}$:

$$q(\upsilon_0) \tag{4}$$

for each $\upsilon_0 \in \mathcal{I}$.

From (2) and (3), the following formula holds for $\mathcal{S}$:

$$(\bigwedge_{a \in \mathcal{A}} a(\upsilon)) \Rightarrow q(\upsilon') \tag{5}$$

for each $\upsilon, \upsilon' \in \Upsilon$ such that $\upsilon \leadsto_{\mathcal{S}} \upsilon'$.

From (4) and (5), it follows that the SI proof of q wrt $\mathcal{S}$ based on $\{p_0, \ldots, p_n\}$ can be conducted.

Another PII graph G' of $p_0 \wedge \ldots \wedge p_n$ can be constructed by deleting all the IMP edges from q to each r_j for $j = 0, \ldots, m$ from G and adding an SIH edge from q to each $a \in \mathcal{A}$ to G. The deletion of the IMP edges and the addition of the SIH edges are shown in Fig. 3. Since G' is constructed as described, G' also satisfies the II condition. The number of the IMP nodes in G' is one less than the number of the IMP nodes in G. From the induction hypothesis, therefore, we conclude that $p_0 \wedge \ldots \wedge p_n$ is an inductive invariant wrt $\mathcal{S}$. $\Box$

Let us consider the proof of $(\forall s : \texttt{Sys})\,(\forall i : \texttt{Pid})\,\texttt{inv3(}s\texttt{,}i\texttt{)}$, where `inv3` is as follows:

```
op inv3 : Sys Pid -> Bool
eq inv3(S,I)
   = (I \in queue(S) implies pc(S,I) = ws or pc(S,I) = cs) .
```

We need $(\forall s : \texttt{Sys})\,(\forall i : \texttt{Pid})\,\texttt{inv3(}s\texttt{,}i\texttt{)}$ to verify that Qlock enjoys the lockout (or starvation) freedom property, which is a liveness property [Ogata and Futatsugi, 2008].

While writing proof scores, we notice that the proof by simultaneous induction needs five more state predicates including $(\forall i, j : \texttt{Pid})\,\texttt{inv1(}s\texttt{,}i\texttt{,}j\texttt{)}$ and $(\forall i : \texttt{Pid})\,\texttt{inv2(}s\texttt{,}i\texttt{)}$. The remaining three state predicates are $(\forall i : \texttt{Pid})\,\texttt{inv4(}s\texttt{,}i\texttt{)}$, $(\forall i : \texttt{Pid})\,\texttt{inv5(}s\texttt{,}i\texttt{)}$ and $(\forall i : \texttt{Pid})\,\texttt{inv6(}s\texttt{,}i\texttt{)}$, where `inv4`, `inv5` and `inv6` are as follows:

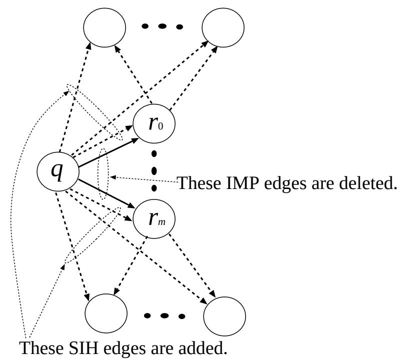

Figure 3: The induction case in the proof of the theorem

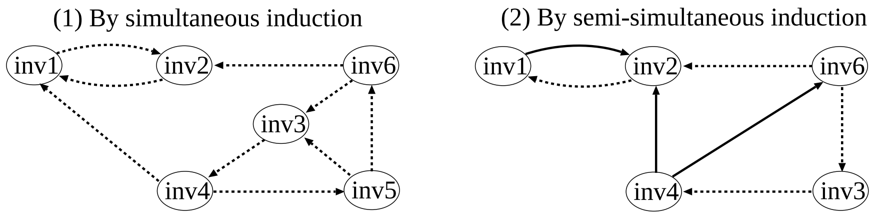

Figure 4: Relations of the state predicates in invariant proofs wrt $\mathcal{S}_{\mathrm{Qlock}}$

```
op inv4 : Sys Pid -> Bool
op inv5 : Sys Pid -> Bool
op inv6 : Sys Pid -> Bool
eq inv4(S,I) = (pc(S,I) = cs implies not(I \in deq(queue(S)))) .
eq inv5(S,I)
   = (top(queue(S)) = I implies not(I \in deq(queue(S)))) .
eq inv6(S,I) = not(I \in queue(S) - I) .
```

Fig. 4 (1) shows the relations of the six state predicates in the proof by simultaneous induction.

We notice that the SI proof of $(\forall i : \texttt{Pid})\,\texttt{inv5(}s\texttt{,}i\texttt{)}$ can be replaced with the IMP proof of the state predicate because the state predicate can be deduced from $(\forall i : \texttt{Pid})\,\texttt{inv6(}s\texttt{,}i\texttt{)}$ under the specification of $\mathcal{S}_{\mathrm{Qlock}}$. We also notice that the SI proof of $(\forall i : \texttt{Pid})\,\texttt{inv4(}s\texttt{,}i\texttt{)}$ can be replaced with the IMP proof of the state predicate because the state predicate can be deduced from $(\forall i : \texttt{Pid})\,\texttt{inv2(}s\texttt{,}i\texttt{)}$ and $(\forall i : \texttt{Pid})\,\texttt{inv5(}s\texttt{,}i\texttt{)}$. Furthermore, `inv6`, instead of `inv5`, can be used in the IMP proof of $(\forall i : \texttt{Pid})\,\texttt{inv4(}s\texttt{,}i\texttt{)}$. Hence, we do not need `inv5` anymore. Since we know that the SI proof of $(\forall i, j : \texttt{Pid})$ $\texttt{inv1(}s\texttt{,}i\texttt{,}j\texttt{)}$ can be replaced with the IMP proof of the state predicate, the relations of the five state predicates in the proof by semi-simultaneous induction can be depicted in Fig. 4 (2).

7 A Case Study

We report on a case study in which we have verified that Alternating Bit Protocol (ABP) enjoys a property. The case study demonstrates that semi-simultaneous induction makes it possible to record the process in which way state predicates used (or lemmas) have been conjectured.

7.1 Alternating Bit Protocol (ABP)

ABP is a communication protocol that makes it possible to reliably deliver packets (expressed as natural numbers in this paper) to a destination (called Receiver) from a source (called Sender) even under unreliable channels whose contents may be lost and duplicated. ABP uses two channels. One channel (called $fifo1$) is used for Sender to basically send packets to Receiver, and the other (called $fifo2$) is used for Receiver to send Sender acknowledgements showing that some packets have reached Receiver.

Sender has one bit (called $bit1$) and one natural number (called $next$), and Receiver has one bit (called $bit2$) and one list of natural numbers (called $list$). The truth values (true and false) are used as bits. Initially, $bit1$ is false, $next$ is 0, $bit2$ is false, and $list$ is nil. Sender repeatedly puts the pair $\langle bit1, next\rangle$ into $fifo1$ at the bottom, which corresponds to send1 in Fig. 5. If $fifo1$ is not empty, Receiver gets the top $\langle b, x\rangle$ from $fifo1$ and then if b is the same as $bit2$, $bit2$ is complemented and x is put into $list$ at the top, which corresponds to rec2 in Fig. 5. Receiver repeatedly puts $bit2$ into $fifo2$ at the bottom, which corresponds to send2 in Fig. 5. If $fifo2$ is not empty, Sender gets the top b from $fifo2$ and then if b is different from $bit1$, $bit1$ is complemented and $next$ is incremented, which corresponds to rec1 in Fig. 5.

Fig. 5 shows a snapshot of ABP, where t and f stand for true and false. In the snapshot, three natural numbers (0, 1 and 2) have been delivered to Receiver. The bit true has not reached Sender although 2 was delivered to Receiver, and then $bit1$ and $next$ remain false and 2.

Each content in each channel may be lost and duplicated. But, we suppose that such unreliable phenomena can been seen only if each content becomes top in each channel. So, if $fifo1$ (or $fifo2$) is not empty, only the top content may be lost and duplicated.

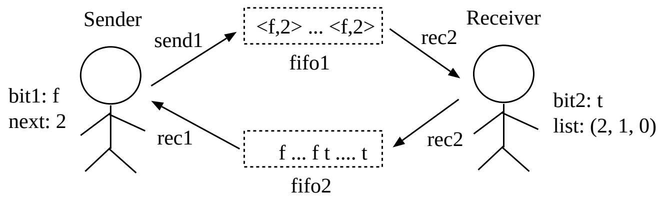

Figure 5: A snapshot of ABP

7.2 Specification of ABP

ABP is formalized as an OTS $\mathcal{S}_{\mathrm{ABP}}$ in which channels are formalized as queues. $\mathcal{S}_{\mathrm{ABP}}$ uses six observers that are used to observe $fifo1$. $fifo2$, $bit1$, $bit2$, $next$ and $list$, respectively. The corresponding observation operators are declared as follows:

```
bop fifo1 : Sys -> PFifo
bop fifo2 : Sys -> BFifo
bop bit1  : Sys -> Bool
bop bit2  : Sys -> Bool
bop next  : Sys -> Nat
bop list  : Sys -> List
```

where `PFifo`, `BFifo`, `Nat` and `List` are sorts for queues of pairs of bits and natural numbers, queues of bits, natural numbers and lists of natural numbers, respectively.

We have the following six equations for `init`, a constant denoting an arbitrary initial state:

```
eq fifo1(init) = empty .
eq fifo2(init) = empty .
eq bit1(init)  = false .
eq bit2(init)  = false .
eq next(init)  = 0 .
eq list(init)  = nil .
```

$\mathcal{S}_{\mathrm{ABP}}$ uses eight transitions. The corresponding transition operators are declared as follows:

```
bop send1 : Sys -> Sys
bop rec1  : Sys -> Sys
bop send2 : Sys -> Sys
bop rec2  : Sys -> Sys
```

```
bop drop1 : Sys -> Sys
bop dup1  : Sys -> Sys
bop drop2 : Sys -> Sys
bop dup2  : Sys -> Sys
```

The first six transition operators correspond to send1, rec1, send2 and rec2 in Fig. 5, respectively. The lost of the first content in $fifo1$ (or $fifo2$) is formalized as `drop1` (or `drop2`), and the duplication of the first content in $fifo1$ (or $fifo2$) is formalized as `dup1` (or `dup2`).

The set of equations for `send1` is as follows:

```
eq fifo1(send1(S)) = enq(fifo1(S),< bit1(S),next(S) >) .
eq fifo2(send1(S)) = fifo2(S) .
eq bit1(send1(S))  = bit1(S) .
eq bit2(send1(S))  = bit2(S) .
eq next(send1(S))  = next(S) .
```

The set of equations for `rec2` is as follows:

```
ceq fifo1(rec2(S)) = deq(fifo1(S)) if c-rec2(S) .
eq  fifo2(rec2(S)) = fifo2(S) .
eq  bit1(rec2(S))  = bit1(S) .
ceq bit2(rec2(S))  = (if bit2(S) = fst(top(fifo1(S)))
                       then not fst(top(fifo1(S)))
                       else bit2(S) fi)
                     if c-rec2(S) .
eq  next(rec2(S))  = next(S) .
ceq list(rec2(S))  = (if bit2(S) = fst(top(fifo1(S)))
                       then (snd(top(fifo1(S))) list(S))
                       else list(S) fi)
                     if c-rec2(S) .
ceq rec2(S)        = S if not c-rec2(S) .
```

where the operator `c-rec2` is declared and defined as follows:

```
op c-rec2 : Sys -> Bool
eq c-rec2(S) = not(fifo1(S) = empty) .
```

Given a pair, the operator `fst` returns the first element and the operator `snd` returns the second element.

The set of equations for `drop1` is as follows:

```
ceq fifo1(drop1(S)) = deq(fifo1(S)) if c-drop1(S) .
eq  fifo2(drop1(S)) = fifo2(S) .
eq  bit1(drop1(S))  = bit1(S) .
```

```
eq  bit2(drop1(S))  = bit2(S) .
eq  next(drop1(S))  = next(S) .
eq  list(drop1(S))  = list(S) .
ceq drop1(S)        = S if not c-drop1(S) .
```

where the operator `c-drop1` is declared and defined as follows:

```
op c-drop1 : Sys -> Bool
eq c-drop1(S) = not(fifo1(S) = empty) .
```

The set of equations for `dup1` is as follows:

```
ceq fifo1(dup1(S)) = top(fifo1(S)) | fifo1(S) if c-dup1(S) .
eq  fifo2(dup1(S)) = fifo2(S) .
eq  bit1(dup1(S))  = bit1(S) .
eq  bit2(dup1(S))  = bit2(S) .
eq  next(dup1(S))  = next(S) .
eq  list(dup1(S))  = list(S) .
ceq dup1(S)        = S if not c-dup1(S) .
```

where the operator `c-dup1` is declared and defined as follows:

```
op c-dup1 : Sys -> Bool
eq c-dup1(S) = not(fifo1(S) = empty) .
```

The set of equations for each of the remaining transition operators can be declared likewise.

7.3 Verification of ABP

One of the properties that ABP should enjoy is the reliable communication property: whenever Receiver receives n natural numbers, they are the first n natural numbers that Sender has sent and the order in which the n natural numbers have been sent is preserved. The property can be expressed as an invariant wrt $\mathcal{S}_{ABP}$. The state predicate concerned is denoted by the operator declared and defined as follows:

```
op inv1 : Sys -> Bool
eq inv1(S)
    = (bit1(S) = bit2(S)
       implies mk(next(S)) = (next(S) list(S))) and
      (not(bit1(S) = bit2(S))
       implies mk(next(S)) = list(S)) .
```

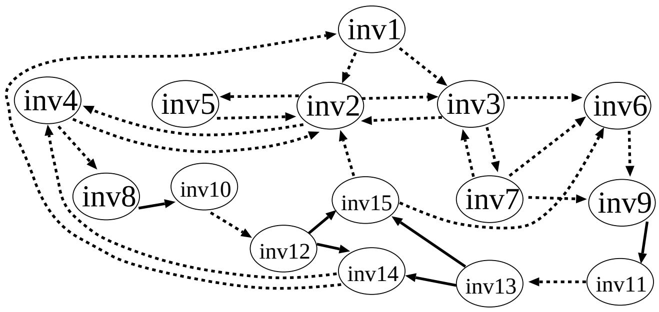

Figure 6: Relations of the state predicates in invariant proofs wrt $\mathcal{S}_{\mathrm{ABP}}$

Given a natural number n, the operator `mk` produces the list $n\ n-1\ \ldots\ 0$ `nil` of natural numbers.

We used 14 more state predicates to prove $(\forall s :$ `Sys`$)$ `inv1`(s) by semi-simultaneous induction. Fig. 6 shows the relations of the 15 state predicates in the proof by semi-simultaneous induction. The state predicates are found in the rest of this section and Appendix A.

The state predicates `inv8` is as follows:

```
op inv8 : Sys Bool Bool Bool BFifo -> Bool
eq inv8(S,BIT1,BIT2,BIT3,BFIFO)
   = ((fifo2(S) = BIT1 | BIT2 | BFIFO and not(BIT1 = BIT2))
      implies ((BIT3 \in BFIFO implies BIT2 = BIT3)
               and BIT2 = bit2(S))) .
```

In the SI proof of `inv8`, we need to take into account cases in which the second and third bits in $fifo2$ are different, the third and fourth bits in $fifo2$ are different, and so on. Therefore, we needed to generalize `inv8` to proceed on the proof. The generalized version of `inv8` is `inv10` that is as follows:

```
op inv10 : Sys Bool Bool Bool BFifo BFifo -> Bool
eq inv10(S,BIT1,BIT2,BIT3,BFIFO1,BFIFO2)
   = ((fifo2(S) = BFIFO1 @ (BIT1 | BIT2 | BFIFO2)
      and not(BIT1 = BIT2))
     implies ((BIT3 \in BFIFO2 implies BIT2 = BIT3)
              and BIT2 = bit2(S))) .
```

Proofs often need such generalization. The SI proof of `inv4` could use `inv10` instead of `inv8`, but lets us directly come up with `inv8` but not `inv10`. It is useful to record the process in which way state predicates used (or lemmas) have been conjectured, which can be done by use of both `inv8` and `inv10` with semi-simultaneous induction. The same argument can be applied to `inv9` and `inv11`.

The state predicates `inv12` is as follows:

```
op inv12 : Sys Bool -> Bool
eq inv12(S,BIT)
   = ((not(fifo1(S) = empty)
      and bit2(S) = fst(top(fifo1(S))))
     implies (BIT \in fifo2(S) implies bit2(S) = BIT)) .
```

In the SI proof of `inv12`, we need to take into account the second pair in $fifo1$, the third pair in $fifo1$, and so on. Therefore, we needed to conjecture some other state predicates (which are expected to hold in all reachable states wrt $\mathcal{S}_{\mathrm{ABP}}$) from which, together with $\mathcal{S}_{\mathrm{ABP}}$, `inv12` is deduced. Like `inv8`, `inv12` could be generalized to conjecture the following state predicate:

```
(PAIR \in fifo1(S) and bit2(S) = fst(PAIR))
implies (BIT \in fifo2(S) implies bit2(S) = BIT))
```

We can also use the following two state predicates:

```
op inv14 : Sys Bool -> Bool
eq inv14(S,BIT)
   = ((bit1(S) = bit2(S))
     implies (BIT \in fifo2(S) implies BIT = bit2(S))) .

op inv15 : Sys BNPair -> Bool
eq inv15(S,PAIR)
   = (not(bit1(S) = bit2(S))
     implies (PAIR \in fifo1(S)
             implies PAIR = < bit1(S),next(S) >)) .
```

If $bit1$ equals $bit2$, the conclusion of `inv12` is deduced from `inv14`, and otherwise the negation of the premise of `inv12` is deduced from `inv15`. Like the SI proof of `inv4`, the SI proof of `inv10` could use `inv14` and `inv15` instead of `inv12`, but lets us directly come up with `inv12` but neither `inv14` nor `inv15`. `inv14` and `inv15` can also be used to deduce `inv13`. This is why we prefer `inv14` and `inv15` to the generalized version of `inv12`.

8 Related Work

(Semi-)Simultaneous induction makes it possible to compositionally (or incrementally) write the proof that $p_0 \wedge \ldots \wedge p_n$ is an inductive invariant wrt an

OTS $\mathcal{S}$, or each p_i is an invariant wrt an OTS $\mathcal{S}$ such that the proof is divided into multiple fragments wrt $p_0, \ldots, p_n$, respectively. There has been proposed another proof technique that allows to compositionally write invariant proofs [Rushby, 2000].

In the proof technique, to prove that a state predicate G is an invariant wrt a state machine, all you have to do is to prove the following:

$\bigvee_j C_{i,j}(s)$ for each state s,
$G'_i(s) \wedge T(s,t) \wedge C_{i,j}(s) \Rightarrow G'_j(t)$
for each state s, t and for each $i, j \in \{1, \ldots, m\}$,
$I(s) \Rightarrow G'_1(s) \vee \ldots \vee G'_m(s)$ for each state s, and
$G'_1(s) \vee \ldots \vee G'_m(s) \Rightarrow G(s)$ for each state s,

where T is the transition relation of the state machine and I is the initiality predicate. $C_{i,j}$ is called a transition condition, and G'_i is called a configuration. A configuration can be regarded as an abstract state represented by a state predicate. Let $G^m_\vee$ be $G'_1 \vee \ldots \vee G'_m$. The first two formulas imply $G^m_\vee(s) \wedge T(s,t) \Rightarrow G^m_\vee(t)$ for each state s, t. From this and the third formula, we conclude that $G^m_\vee$ is an inductive invariant wrt the state machine. Therefore, from the fourth formula, G is an invariant wrt the state machine. The proofs of the first and second formulas for each G'_i can be independently written from the others.

In the proof technique, it is proved that a disjunction $G'_1 \vee \ldots \vee G'_m$ is an inductive invariant wrt a state machine, while it is proved that a conjunction $G_1 \wedge \ldots \wedge G_n$ is an inductive invariant wrt a state machine in a typical proof technique of invariants. Therefore, the proof technique may be called the disjunctive invariant proof technique. Let $G^n_\wedge$ be $G_1 \wedge \ldots \wedge G_n$. John Rushby writes the following [Rushby, 2000]:

> ...the inadequacy of $G^i_\wedge$ only becomes apparent through failure of the attempted proof of its inductiveness—and proof of the putative inductiveness of $G^{i+1}_\wedge$ must then start over.

It is proved that a conjunction $p_0 \wedge \ldots \wedge p_i$ is an inductive invariant in (semi-)simultaneous induction. When we need another state predicate p_{i+1}, however, the entire proof of the inductiveness of $p_0 \wedge \ldots \wedge p_{i+1}$ does not have to start over. All we have to do is to write the proof fragment wrt p_{i+1}.

Another difference between the disjunctive invariant proof technique and (semi-)simultaneous induction is that the targets of the former are synchronous systems, while those of the latter are asynchronous systems. It must be worth trying to apply the former to asynchronous systems, and the latter to synchronous systems.

Since OTSs are described as behavioral specifications in the OTS/CafeOBJ method such that each in $\mathcal{T}$ and $\mathcal{O}$ is regarded as a behavioral operator, it must be worth mentioning what are mainly concerned in behavioral specifications, namely behavioral equivalence. Several proof methods have been proposed for behavioral equivalence. Among them are context induction [Hennicker, 1990], coinduction [Goguen and Malcolm, 2000, Diaconescu and Futatsugi, 2000] and circular coinduction [Roşu and Lucanu, 2009]. Behavioral equivalence $=_{\mathcal{S}}$ in OTSs is given by the definition: $\upsilon_1 =_{\mathcal{S}} \upsilon_2$ iff for each $o : \mathcal{O}$, $o(\upsilon_1, x_1, \ldots, x_m) = o(\upsilon_2, x_1, \ldots, x_m)$ for all $x_1 : D_{o1}, \ldots, x_m : D_{om}$. Although we may have to use some proof method for behavioral equivalence to prove that a given specification conforms to the definition of OTSs, it is straightforward to carry out the proof if an OTS is specified according to what is described in Section 3.

Given an OTS $\mathcal{S}$ and a state predicate p, let us consider a behavioral specification such that each in $\mathcal{T} \cup \{p\}$ is regarded as a behavioral operator. Let $\sim$ be the behavioral equivalence in this behavioral specification. Let $[\upsilon_p]$ be an equivalence class of $\upsilon_p \in \Upsilon$ in the quotient set $\Upsilon/\!\sim$ such that $p(\upsilon_p)$ holds and for each $t \in \mathcal{T}$, $t(\upsilon_p, y_1, \ldots, y_n) \sim \upsilon_p$ for all $y_1 : D_{t1}, \ldots, y_n : D_{tn}$. If $\upsilon_0 \sim \upsilon_p$ for each $\upsilon_0 \in \mathcal{I}$, then p is an invariant wrt $\mathcal{S}$. Note that p is not necessarily an inductive invariant wrt $\mathcal{S}$ because there may be $\upsilon_1 \in \Upsilon \setminus [\upsilon_p]$ such that $p(\upsilon_1)$ holds but $p(t(\upsilon_1, y_1, \ldots, y_n))$ does not for some $t \in \mathcal{T}$ and some $y_1 : D_{t1}, \ldots, y_n : D_{tn}$. If we use this approach to prove that a given state predicate p is an invariant wrt a given OTS $\mathcal{S}$, then (1) the behavioral equivalence $\sim$ is constructed, (2) $\upsilon_p \in \Upsilon$ is found such that $p(\upsilon_p)$ holds and for each $t \in \mathcal{T}$, $t(\upsilon_p, y_1, \ldots, y_n) \sim \upsilon_p$ for all $y_1 : D_{t1}, \ldots, y_n : D_{tn}$ and (3) it is checked that $\upsilon_0 \sim \upsilon_p$ for each $\upsilon_0 \in \mathcal{I}$. It is not straightforward to carry out (1) and (2).

Let q be a state predicate such that for each $\upsilon \in \Upsilon$, $q(\upsilon)$ holds and $q(t(\upsilon, y_1, \ldots, y_n))$ holds for each $t \in \mathcal{T}$ and all $y_1 : D_{t1}, \ldots, y_n : D_{tn}$. Then, $\{\upsilon_q \in \Upsilon \mid p(\upsilon_q)\}$ is the equivalence class of $\upsilon_q \in \Upsilon$ in $\Upsilon/\!\sim$ such that $p(\upsilon_q)$ holds. If $\upsilon_0 \sim \upsilon_q$ for each $\upsilon_0 \in \mathcal{I}$, then q is an inductive invariant wrt $\mathcal{S}$. If $q \Rightarrow p$, then p is an invariant wrt $\mathcal{S}$. Although we do not need to carry (1) and (2) in this approach, this approach needs to find an inductive invariant q wrt $\mathcal{S}$ such that $q \Rightarrow p$, which is the standard approach to proving invariants. It may be worth investigating whether there are some cases such that the approach described in the previous paragraph is superior to the standard approach.

9 Conclusion

Although simultaneous induction is folklore in mathematics [Jouannaud, 2005] and treated in an undergraduate text in mathematics [Lovász et al.(2003)], mathematicians do not seem to use it actively. This seems to be because mathematicians do not use long formulas [Lamport, 2005]. Although semi-simultaneous induction must be folklore in mathematics as well, we have not found any written documents on it. (Semi-)simultaneous induction is just another way of organizing a traditional deductive invariant proof as the disjunctive invariant proof technique, but has some advantages over the standard way of doing so:

1. (Semi-)simultaneous induction makes it possible to compositionally (or incrementally) write proofs of invariants. Proofs (or proof scores) in progress do not have to be thrown away and only proof fragments wrt new state predicates have to be written.

2. Semi-simultaneous induction makes proofs by only simultaneous induction more concise, and may need less cases, making proof shorter, than the traditional deductive invariant proof technique.

3. Semi-simultaneous induction makes it possible to record the process in which way lemmas have been conjectured.

Some lessons learnt on how to use (semi-)simultaneous induction are as follows:

1. Mostly conduct SI proofs to prove that an OTS enjoys invariants, and conduct IMP proofs when you need to generalize state predicates. This is because it is possible to record the process in which way lemmas have been conjectured, making proof scores readable for human users. This is very important because we think that proof scores should be part of specifications, which are supposed to be read by human users.

2. After the completion of the proof, try to find a state predicate such that its SI proof has been conducted and it can be deduced from some other state predicates together with an OTS concerned, and if such a state predicate is found, replace the SI proof with the IMP proof of the state predicate. This makes proof scores shorter.

The second lesson may erase records of how to conjecture lemmas. This is because information on case analysis in induction cases is one important source of how to conjecture lemmas and may be deleted by use of the second lesson. If it is crucial to record the process in which way lemmas have been conjectured, the second lesson should not be used.

We have summarized some tips on writing proof scores in the OTS/CafeOBJ method in [Ogata and Futatsugi, 2006]. It may be worth noting that those tips can be effectively used to write proof scores by semi-simultaneous induction.

Although compositionally writing proof scores with (semi-)simultaneous induction makes it possible to reduce the burden of interactive theorem proving, it is unrealistic to manage much larger proof scores manually. Therefore, a proof assistant has been being designed and developed for the OTS/CafeOBJ method.

References

[Diaconescu and Futatsugi, 1998] Diaconescu, R., Futatsugi, K.: CafeOBJ report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification, volume 6 of AMAST Series in Computing, World Scientific, 1998.

[Diaconescu and Futatsugi, 2000] Diaconescu, R., Futatsugi, K.: Behavioural coherence in object-oriented algebraic specification, Journal of Universal Computer Science, 6 (2000) 74–95.

[Diaconescu et al, 2003] Diaconescu, R., Futatsugi, K., Ogata, K.: CafeOBJ: Logical foundations and methodologies, Computing and Informatics 22 (2003), 257–283.

[Gâinâ et al, 2009] Gâinâ, D., Futatsugi, K., Ogata, K.: Constructor-based institutions, 3rd Conference on Algebra and Coalgebra in Computer Science (3rd CALCO), LNCS 5728, 398–412, Springer, 2009.

[Goguen and Malcolm, 2000] Goguen, J., Malcolm, G.: A hidden agenda, Theoretical Computer Science 245 (2000), 55–101.

[Gutiérrez et al, 2012] Gutiérrez, R., Meseguer, J., Rocha, C.: Order-sorted equality enrichments modulo axioms, 9th InternationalWorkshop on Rewriting Logic and its Applications (WRLA 2012), Pre-Proceedings, 69–88, 2012.

[Hennicker, 1990] Hennicker, R.: Context induction: A proof principle for behavioural abstractions, International Symposium on Design and Implementation of Symbolic Computation Systems (DISCO '90), LNCS 429, 101–110, Springer, 1990.

[Ouranos, 2007] Ouranos, I., Stefaneas, P.: An algebraic framework for modeling of mobile systems, IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences 90-A (2007), 9, 1986–1999.

[Jouannaud, 2005] Jouannaud, J.-P.: Simultaneous induction is folklore, Personal Communication (2005).

[Kong et al, 2010] Kong, W., Ogata, K., Futatsugi, K.: Towards reliable e-government systems with the OTS/CafeOBJ method, IEICE Transactions on Information and Systems 93-D (2010), 5, 974–984.

[Lamport, 2005] Lamport, L.: How to write a long formula, Formal Aspects of Computing 6 (2005), 5, 580–584.

[Lovász et al.(2003)] Lovász, L., Pelikán, J., Vesztergombi, K.: Discrete Mathematics: Elementary and Beyond, Undergraduate Texts in Mathematics, Springer, 2003.

[Manna and Pnueli, 1995] Manna, Z., Pnueli, A.: Temporal Verification of Reactive Systems: Safety, Springer, 1995.

[Nakamura and Futatsugi, 2007] Nakamura, M., Futatsugi, K.: On equality predicates in algebraic specification languages, 4th International Conference on Theoretical Aspects of Computing (ICTAC 2007), LNCS 4711, 381–395, Springer, 2007.

[Ogata and Futatsugi, 2003] Ogata, K., Futatsugi, K.: Proof scores in the OTS/CafeOBJ method, 6th International Conference on Formal Methods for Open Object-Based Distributed Systems (6th FMOODS), LNCS 2884, 170–184, Springer, 2003.

[Ogata and Futatsugi, 2006] Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method, Algebra, Meaning, and Computation: A Festschrift Symposium in Honor of Joseph Goguen, LNCS 4060, 596–615, Springer, 2006.

[Ogata and Futatsugi, 2008] Ogata, K., Futatsugi, K.: Proof score approach to verification of liveness properties, IEICE Transactions on Information and Systems

E91-D (2008), 12, 2804–2817.

[Ogata and Futatsugi, 2010] Ogata, K., Futatsugi, K.: Proof score approach to analysis of electronic commerce protocols, International Journal of Software Engineering and Knowledge Engineering 20 (2010), 2, 253–287.

[Ouranos et al, 2010] Ouranos, I., Stefaneas, P., Ogata, K.: Formal modeling and verification of sensor network encryption protocol in the OTS/CafeOBJ method, 4th International Symposium on Leveraging Applications of Formal Methods, Verification, and Validation (ISoLA 2010, Part 1), LNCS 6415, 75–89, Springer, 2010.

[Roşu and Lucanu, 2009] Roşu, G., Lucanu, D.: Circular coinduction: A proof theoretical foundation, 3rd International Conference on Algebra and Coalgebra in Computer Science (3rd CALCO), LNCS 5728, 127–144, Springer, 2009.

[Rushby, 2000] Rushby, J.: Verification diagrams revisited: Disjunctive invariants for easy verification, 12th International Conference on Computer Aided Verification (12th CAV), LNCS 1855, 508–520, Springer, 2000.

[Triantafyllou et al, 2010] Triantafyllou, N., Ouranos, I., Stefaneas, P., Frangos, P.: Formal specification and verification of the OMA license choice algorithm in the OTS/CafeOBJ method, International Conference on Wireless Information Networks and Systems (WINSYS 2010), 173–180, 2010.

[Woodcock et al, 2009] Woodcock, J., Larsen, P. G., Bicarregui, J., Fitzgerald, J.: Formal methods: Practice and experience, ACM Computing Surveys 4 (2009), 19:1–19:36.

A Other State Predicates Used for ABP Verification

```
op inv2 : Sys -> Bool
eq inv2(S)
   = (not(fifo2(S) = empty) and not(bit1(S) = top(fifo2(S))))
     implies (bit2(S) = top(fifo2(S))) .

op inv3 : Sys -> Bool
eq inv3(S)
   = (not(fifo1(S) = empty) and bit2(S) = fst(top(fifo1(S))))
     implies (bit1(S) = fst(top(fifo1(S)))
              and next(S) = snd(top(fifo1(S)))) .

op inv4 : Sys Bool -> Bool
eq inv4(S,BIT)
   = (not(fifo2(S) = empty) and not(bit1(S) = top(fifo2(S)))
                            and BIT \in fifo2(S))
     implies (top(fifo2(S)) = BIT) .

op inv5 : Sys Bool -> Bool
eq inv5(S,BIT)
   = (not(fifo2(S) = empty) and BIT \in fifo2(S)
                            and not(bit1(S) = BIT))
     implies (bit2(S) = BIT) .
```

```
op inv6 : Sys BNPair -> Bool
eq inv6(S,PAIR)
   = (not(fifo1(S) = empty) and bit2(S) = fst(top(fifo1(S)))
                            and PAIR \in fifo1(S))
     implies (top(fifo1(S)) = PAIR) .

op inv7 : Sys BNPair -> Bool
eq inv7(S,PAIR)
   = (not(fifo1(S) = empty) and PAIR \in fifo1(S)
                            and bit2(S) = fst(PAIR))
     implies (bit1(S) = fst(PAIR) and next(S) = snd(PAIR)) .

op inv9 : Sys BNPair BNPair BNPair PFifo -> Bool
eq inv9(S,PAIR1,PAIR2,PAIR3,PFIFO)
   = ((fifo1(S) = PAIR1 | PAIR2 | PFIFO and not(PAIR1 = PAIR2))
     implies ((PAIR3 \in PFIFO implies PAIR2 = PAIR3)
              and PAIR2 = < bit1(S),next(S) >)) .

op inv11 : Sys BNPair BNPair BNPair PFifo PFifo -> Bool
eq inv11(S,PAIR1,PAIR2,PAIR3,PFIFO1,PFIFO2)
   = ((fifo1(S) = PFIFO1 @ (PAIR1 | PAIR2 | PFIFO2)
      and not(PAIR1 = PAIR2))
     implies ((PAIR3 \in PFIFO2 implies PAIR2 = PAIR3)
              and PAIR2 = < bit1(S),next(S) >)) .

op inv13 : Sys BNPair -> Bool
eq inv13(S,PAIR)
   = ((not(fifo2(S) = empty) and not(bit1(S) = top(fifo2(S))))
     implies
     (PAIR \in fifo1(S) implies PAIR = < bit1(S),next(S) >)) .
```