Available online at www.sciencedirect.com

ScienceDirect

Electronic Notes in Theoretical Computer Science 201 (2008) 127–154

Electronic Notes in Theoretical Computer Science

www.elsevier.com/locate/entcs

# Simulation-based Verification for Invariant Properties in the OTS/CafeOBJ Method

Kazuhiro Ogata[1] Kokichi Futatsugi[2]

*School of Information Science*
*Japan Advanced Institute of Science and Technolog*
*1-1 Asahidai, Nomi, Ishikawa 923-1290*

**Abstract**

The OTS/CafeOBJ method is a formal method to model systems, specify models and verify that models satisfy properties. We propose a way to verify that a state machine $\mathcal{S}$ satisfies invariant properties based on a simulation from $\mathcal{S}$ to another state machine, which is more abstract than $\mathcal{S}$, in the OTS/CafeOBJ method. Three communication protocols are used as examples to demonstrate the proposed method.

*Keywords:* algebraic specifications, equations, rewriting, proof scores

## 1 Introduction

We have been developing the OTS/CafeOBJ method [17,15,6], in which we mainly use induction on the structure of the reachable state spaces of state machines to verify that state machines satisfy invariant properties. The verification method is called the induction-based (invariant) verification method in this paper. In the OTS/CafeOBJ method, observational transition systems, or OTSs are used as state machines, and CafeOBJ, an algebraic specification language and system, is used to specify OTSs and verify that OTSs satisfy properties. A number of cases studies have been conducted, among which are [14,16,11].

[1] Email: ogata@jaist.ac.jp
[2] Email: kokichi@jaist.ac.jp

doi:10.1016/j.entcs.2008.02.018

This paper proposes another way to verify that state machines satisfy invariant properties. The proposed method is based on simulations from OTSs to OTSs. A simulation from an OTS $\mathcal{S}$ to an OTS $\mathcal{S}_{\mathrm{A}}$ is a relation between the reachable sates of $\mathcal{S}$ and those of $\mathcal{S}_{\mathrm{A}}$ that satisfies some conditions. The proposed method is called the simulation-based (invariant) verification method.

We first define simulations from OTSs to OTSs and then prove a theorem saying that if there exists a simulation $r$ from an OTS $\mathcal{S}$ to another OTS $\mathcal{S}_{\mathrm{A}}$, which is more abstract than $\mathcal{S}$, then a state predicate $p$ is invariant with respect to (wrt) $\mathcal{S}$ when a state predicate $p_{\mathrm{A}}$ is invariant wrt $\mathcal{S}_{\mathrm{A}}$ and $p$ is deduced from $p_{\mathrm{A}}$ assuming $r$. The proposed simulation-based verification method is based on the theorem. We then prove another theorem saying that the composition $s \circ r$ of a simulation $r$ from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$ and a simulation $s$ from $\mathcal{S}_{\mathrm{A}}$ to $\mathcal{S}_{\mathrm{B}}$ is a simulation $\mathcal{S}$ to $\mathcal{S}_{\mathrm{B}}$.

In this paper, we use three communication protocols (an abstract protocol called BCP, an intermediate protocol called SCP and a concrete protocol called ABP) as examples and report on two case studies on the protocols. In the first case study, we prove that there exists a simulation $r1$ from SCP to BCP, and in the second case study, we prove that there exists a simulation $r2$ from ABP to SCP, which implies that the composition $r2 \circ r1$ is a simulation from ABP to BCP.

The rest of the paper is organized as follows. Section 2 describes OTSs. Section 3 introduces CafeOBJ. Section 4 defines simulations from OTSs to OTSs. Section 5 proposes the simulation-based invariant verification method. Section 6 defines compositions of simulations from OTSs to OTSs. Section 7 mentions some related work. Section 8 concludes the paper.

## 2 Observational Transition Systems (OTSs)

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

**Definition 2.1** An OTS $\mathcal{S}$ is $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that

- $\mathcal{O}$: A finite set of observers. Each *observer* $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \rightarrow D_o$ is an indexed function that has $m$ indexes $x_1, \ldots, x_m$ whose types are $D_{o1}, \ldots, D_{om}$. For brevity, we suppose that the name $o$ of each observer $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \rightarrow D_o$ is distinct from each other. Therefore, the name $o$ may be used to refer to the observer. The equivalence relation $(\upsilon_1 =_{\mathcal{S}} \upsilon_2)$ between two states $\upsilon_1, \upsilon_2 \in \Upsilon$ is defined as $\forall o : \mathcal{O}. \forall x_1 : D_{o1} \ldots \forall x_m : D_{om}. (o_{x_1,\ldots,x_m}(\upsilon_1) = o_{x_1,\ldots,x_m}(\upsilon_2))$.

- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A finite set of transitions. Each *transition* $t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \to \Upsilon$ is an indexed function that has $n$ indexes $y_1, \ldots, y_n$ whose types are $D_{t1}, \ldots, D_{tn}$ provided that $t_{y_1,\ldots,y_n}(\upsilon_1) =_{\mathcal{S}} t_{y_1,\ldots,y_n}(\upsilon_2)$ for each equivalence class $\mathbf{c} \in \Upsilon/=_{\mathcal{S}}$ of the quotient set of $\Upsilon$ by $=_{\mathcal{S}}$, each $\upsilon_1, \upsilon_2 \in \mathbf{c}$ and each $y_k : D_{tk}$ for $k = 1, \ldots, n$. Each transition $t_{y_1,\ldots,y_n}$ has the condition $c\text{-}t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \to \text{Bool}$, which is called *the effective condition* of the transition. If $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ does not hold, then $t_{y_1,\ldots,y_n}(\upsilon) =_{\mathcal{S}} \upsilon$. For brevity, we suppose that the name $t$ of each transition $t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \to \Upsilon$ is distinct from each other. Therefore, the name $t$ may be used to refer to the transition and the name $c\text{-}t$ may be used to refer to the effective condition.□

The definition of each transition looks like:

$t_{y_1,\ldots,y_n}(\upsilon) \triangleq \upsilon'$ **if** $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ **s.t.**
  $\ldots$
  $o_{x_1,\ldots,x_m}(\upsilon') = \ldots$
  $\ldots$
**where** $c\text{-}t_{y_1,\ldots,y_n}(\upsilon) \triangleq \ldots$

The definition says that if $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ holds for a given state $\upsilon$, then $t_{y_1,\ldots,y_n}$ moves $\upsilon$ to $\upsilon'$ that satisfies all equations between **s.t.** and **where**, and if $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ does not, then $t_{y_1,\ldots,y_n}$ does not change $\upsilon$. If $o_{x_1,\ldots,x_m}(\upsilon')$ equals $o_{x_1,\ldots,x_m}(\upsilon)$, the corresponding equation "$o_{x_1,\ldots,x_m}(\upsilon') = o_{x_1,\ldots,x_m}(\upsilon)$" can be omitted. The definition of $c\text{-}t_{y_1,\ldots,y_n}$ is written after **where**. If $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ holds for an arbitrary state $\upsilon$, then "**if** $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$" and "**where** $c\text{-}t_{y_1,\ldots,y_n}(\upsilon) \triangleq \ldots$" may be omitted.

Given an OTS $\mathcal{S}$ and two states $\upsilon, \upsilon' \in \Upsilon$, if there exist $t \in \mathcal{T}$ and $y_k : D_{tk}$ for $k = 1.\ldots, n$ such that $t_{y_1,\ldots,y_n}(\upsilon) =_{\mathcal{S}} \upsilon'$, we write $\upsilon \leadsto_{\mathcal{S}} \upsilon'$ and call $\upsilon'$ a *successor state* of $\upsilon$ with respect to (wrt) $\mathcal{S}$. $\leadsto^*_{\mathcal{S}}$ is a reflexive transitive closure of $\leadsto_{\mathcal{S}}$. When $k$ time applications of transitions move $\upsilon$ to $\upsilon'$, we write $\upsilon \leadsto^k_{\mathcal{S}} \upsilon'$. That is, $\upsilon \leadsto^0_{\mathcal{S}} \upsilon'$ such that $\upsilon =_{\mathcal{S}} \upsilon'$, and $\upsilon \leadsto^{k+1}_{\mathcal{S}} \upsilon'$ if there exists $\upsilon'' \in \Upsilon$ such that $\upsilon \leadsto^k_{\mathcal{S}} \upsilon''$ and $\upsilon'' \leadsto_{\mathcal{S}} \upsilon'$.

**Definition 2.2** Given an OTS $\mathcal{S}$, *reachable states* wrt $\mathcal{S}$ are inductively defined:

- Each $\upsilon \in \mathcal{I}$ is reachable wrt $\mathcal{S}$.
- For each $\upsilon, \upsilon' \in \Upsilon$ such that $\upsilon \leadsto_{\mathcal{S}} \upsilon'$, if $\upsilon$ is reachable wrt $\mathcal{S}$, so is $\upsilon'$.

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

**Proposition 2.3** *Given an arbitrary OTS $\mathcal{S}$ and arbitrary two states $\upsilon, \upsilon' \in \Upsilon$ such that $\upsilon \leadsto^*_{\mathcal{S}} \upsilon'$, if $\upsilon$ is reachable wrt $\mathcal{S}$, then so is $\upsilon'$.*

**Proof.** By induction on $k$ in $\upsilon \rightsquigarrow_{\mathcal{S}}^{k} \upsilon'$. If $k$ is 0, it is trivial. If $\upsilon \rightsquigarrow_{\mathcal{S}}^{k+1} \upsilon'$, there exists $\upsilon''$ such that $\upsilon \rightsquigarrow_{\mathcal{S}}^{k} \upsilon''$ and $\upsilon'' \rightsquigarrow_{\mathcal{S}} \upsilon'$. Since $\upsilon''$ is reachable wrt $\mathcal{S}$ from the induction hypothesis, so is $\upsilon'$ from Definition 2.2. □

Predicates whose types are $\Upsilon \rightarrow$ Bool are called *state predicates*. All properties considered in this paper are *invariant properties*.

**Definition 2.4** Any state predicate $p : \Upsilon \rightarrow \text{Bool}$ is called *invariant* wrt $\mathcal{S}$ if $p$ holds in all reachable states wrt $\mathcal{S}$, i.e. $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.\, p(\upsilon)$. □

**Example 2.5** Let us consider a communication protocol, which directly delivers natural numbers from the sender to the receiver. The communication protocol is called a *bare communication protocol* (*BCP*). Figure 1 shows a snapshot of BCP. The sender has a natural number variable (*index*) and the receiver has a list (*list*) of natural numbers. Initially, *index* is 0 and *list* is nil. What BCP does is as follows:

- *send*: *index* is added to *list* and incremented.

The OTS $\mathcal{S}_{\text{BCP}}$ modeling BCP is as follows:

$$\mathcal{O}_{\text{BCP}} \triangleq \{\text{index} : \Upsilon \rightarrow \text{Nat}, \text{list} : \Upsilon \rightarrow \text{List}\}$$
$$\mathcal{I}_{\text{BCP}} \triangleq \{\upsilon \mid \text{index}(\upsilon) = 0 \wedge \text{list}(\upsilon) = \text{nil}\}$$
$$\mathcal{T}_{\text{BCP}} \triangleq \{\text{send} : \Upsilon \rightarrow \Upsilon\}$$

where Nat and List are the data types for natural numbers and lists of natural numbers, respectively. The transition *send* is defined as follows:

$$\begin{aligned}&\text{send}(\upsilon) \triangleq \upsilon' \ \textbf{s.t.}\\ &\quad \text{index}(\upsilon') = \text{index}(\upsilon) + 1\\ &\quad \text{list}(\upsilon') = (\text{index}(\upsilon)\ \text{list}(\upsilon))\end{aligned}$$

The juxtaposition operator is used as the constructor of non-nil lists. That is, $\text{index}(\upsilon)\ \text{list}(\upsilon)$ is the list obtained by adding $\text{index}(\upsilon)$ to $\text{list}(\upsilon)$ at the top.

One desired property that BCP should satisfy is as follows: when the receiver receives $N$ natural numbers, they are the first $N$ natural numbers that the sender has sent and the order in which the $N$ natural numbers has been sent is preserved. The property is called the *reliable communication property* in this paper. The property can be formalized as an invariant property wrt $\mathcal{S}_{\text{BCP}}$ with the state predicate that is defined as follows:

$$\text{rc0}(\upsilon) \triangleq \text{mk}(\text{index}(\upsilon)) = (\text{index}(\upsilon)\ \text{list}(\upsilon))$$

where $\text{mk}(0)$ equals nil and $\text{mk}(k+1)$ equals $(k+1\ \text{mk}(k))$. The verification that BCP satisfies the reliable communication property is equivalent to the proof of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\text{BCP}}}.\, \text{rc0}(\upsilon)$. □

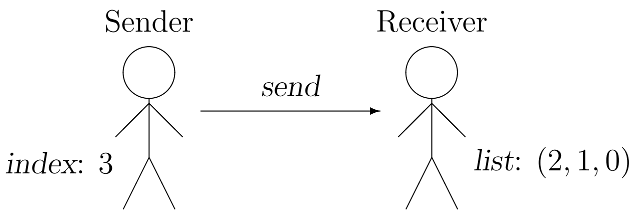

Fig. 1. A snapshot of BCP

## 3 CafeOBJ

CafeOBJ[3] [4] is an algebraic specification language mainly based on order-sorted algebras and hidden algebras [8,5]. A specification written in CafeOBJ is basically a set of equations. One functionality provided by the CafeOBJ system (which is an implementation of CafeOBJ and simply referred as CafeOBJ) is a rewrite engine, which reduces a given term by regarding equations as left-to-right rewrite rules. The executability makes it possible to use CafeOBJ as an interactive proof assistant.

Data types are specified in terms of order-sorted algebras, and state machines such as OTSs are specified in terms of hidden algebras. Algebraic specifications of state machines are called *behavioral specifications*. There are two kinds of sorts in CafeOBJ: *visible sorts* and *hidden sorts*. A visible sort denotes a data type, while a hidden sort denotes the state space of a state machine. There are three kinds of operators (or operations) wrt hidden sorts: *hidden constants*, *action operators* and *observation operators*. Hidden constants denote initial states of state machines, action operators denote state transitions of state machines, and observation operators let us know the situation where state machines are located. Both an action operator and an observation operator take a state of a state machine and zero or more data. The action operator returns the successor state of the state wrt the state transition denoted by the action operator plus the data. The observation operator returns a value that characterizes the situation where the state machine is located.

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`, 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 ($\neg$),

[3] See `http://www.ldl.jaist.ac.jp/cafeobj/`.

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 `B1 and B2`. The operator `if_then_else_fi` corresponding to the **if** construct in programming languages is also declared. CafeOBJ uses the Hsiang term rewriting system [9] 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.

$\mathcal{S}_{\mathrm{BCP}}$ is used as an example to describe specification in CafeOBJ and verification with CafeOBJ.

## *3.1 Specification in CafeOBJ*

The data types used are first specified. Natural numbers are specified in the module `PNAT`:

```
mod! PNAT { [Nat]
  op 0 : -> Nat  op s : Nat -> Nat
  op _=_ : Nat Nat -> Bool {comm}
  vars X Y : Nat
  eq (X = X) = true .  eq (0 = s(X)) = false .
  eq (s(X) = s(Y)) = (X = Y) .
}
```

The keyword `mod!` indicates that the module is a tight semantics declaration, meaning the smallest model (implementation) that respects all requirements written in the module. Visible sorts are declared by enclosing them with `[` and `]`. `Nat` is the visible sort of natural numbers. The keyword `op` is used to declare (non-observation and action) operators, and `ops` to declare more than one such operator simultaneously. The operator `0` denotes zero and the operator `s` is the successor function of natural numbers. Operators with no arguments such as `0` are called constants. The operator `_=_` checks if two natural numbers are equal. The keyword `comm` specifies that the operator `_=_` is commutative. The keyword `var` is used to declare variables, and `vars` to declare more than one variable simultaneously. `X` and `Y` are variables of `Nat`. The keyword `eq` is used to declare equations, and `ceq` to declare conditional equations. Equations and conditional equations are used to define operators and specify properties of operators.

Generic lists are specified in the module `List`:

```
mod! LIST(M :: EQTRIV) { [List]
  op nil : -> List       op __ : Elt.M List -> List
  op hd : List -> Elt.M  op tl : List -> List
  op _=_ : List List -> Bool {comm}
  vars L L1 L2 : List  vars X Y : Elt.M
  eq hd(X L) = X .     eq tl(X L) = L .
  eq (L = L) = true .  eq (nil = X L) = false .
  eq (X L1 = Y L2) = (X = Y and L1 = L2) .
}
```

The constant `nil` denotes the nil list and the juxtaposition operator `__` is the constructor of non-nil lists. The operators `hd` and `tl` are the usual functions of lists. The operator `_=_` checks if two lists are equal.

`LIST` has the (formal) parameter `M`. Given a module that respects all the requirements in the module `EQTRIV` as an actual parameter, `List` is instantiated. `EQTRIV` is as follows:

```
mod* EQTRIV { [Elt]
  op _=_ : Elt Elt -> Bool {comm}
}
```

The keyword `mod*` indicates that the module is a loose semantics declaration, meaning an arbitrary model (implementation) that respects all requirements written in the module.

Lists of natural numbers are specified in the module `PNAT-LIST`:

```
mod! PNAT-LIST { pr(LIST(PNAT))
  op mk : Nat -> List
  var X : Nat
  eq mk(0) = 0 nil .  eq mk(s(X)) = s(X) mk(X) .
}
```

`LIST(PNAT)` is the instance of `LIST` with `PNAT`. The keyword `pr` is used to import modules. The operator `mk` takes a natural number $k$ and makes the list $(k, k-1, \ldots, 0)$.

Now that we have all data types used in $\mathcal{S}_{\mathrm{BCP}}$, $\mathcal{S}_{\mathrm{BCP}}$ is next specified in the module `BCP`:

```
mod* BCP { pr(PNAT-LIST)  *[Sys0]*
  op init : -> Sys0
  bop index : Sys0 -> Nat  bop list  : Sys0 -> List
  bop send  : Sys0 -> Sys0
  var S : Sys0
  eq index(init) = 0 .  eq list(init)  = nil .
  eq index(send(S)) = s(index(S)) .
  eq list(send(S)) = index(S) list(S) .
}
```

Hidden sorts are declared by enclosing them with `*[` and `]*`. `Sys0` is the hidden sort denoting $\Upsilon$. The hidden constant `init` denotes an arbitrary initial state of $\mathcal{S}_{\mathrm{BCP}}$. The keyword `bop` is used to declare observation and action operators, and `bops` to declare more than one such operator simultaneously. The observation operators `index` and `list` correspond to the observers index and list, respectively. The action operator `send` corresponds to the transition send. The first two equations define `init` and the last two equations define the transition send.

### 3.2 Verification with CafeOBJ

We describe the verification that BCP satisfies the property.

The module `INV` is first declared:

```
mod INV { pr(BCP)
  op BCP-inv1 : Sys0 -> Bool
  var S0 : Sys0
  eq BCP-inv1(S0) = (index(S0) list(S0) = mk(index(S0))) .
}
```

The keyword `mod` is used to declare modules, which are not parts of (system) specifications. The operator `BCP-inv1` corresponds to the state predicate rc0. The proof of $\forall v : \mathcal{R}_{\mathcal{S}_{\mathrm{BCP}}}.\,\mathrm{rc0}(v)$ is done by induction on $v$. Therefore, the module `ISTEP` is declared:

```
mod ISTEP { pr(INV)
  ops s0 s0' : -> Sys0
  op BCP-istep1 : -> Bool
  eq BCP-istep1 = BCP-inv1(s0) implies BCP-inv1(s0') .
}
```

The constant `s0` denotes an arbitrary state and the constant `s0'` is used to denote a successor state of `s0` in proofs written in CafeOBJ. The term `BCP-inv1(s0')` is the formula to prove in each induction case, and the term `BCP-inv1(s0)` is the induction hypothesis.

The proof written in CafeOBJ is as follows:

```
open INV
  red BCP-inv1(init) .
close

open ISTEP
  eq s0' = send(s0) .
```

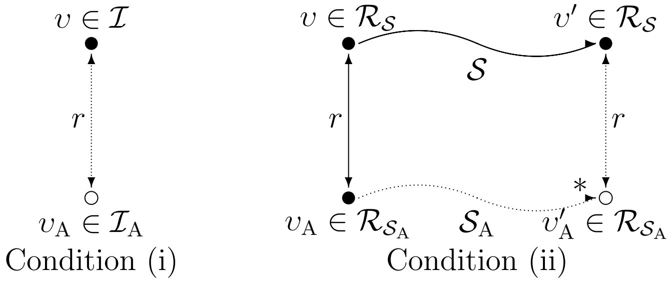
$v \in \mathcal{I}$
$r$
$v_{\mathrm{A}} \in \mathcal{I}_{\mathrm{A}}$
Condition (i)
$v \in \mathcal{R}_{\mathcal{S}}$
$v' \in \mathcal{R}_{\mathcal{S}}$
$\mathcal{S}$
$r$
$r$
$*$
$v_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$
$\mathcal{S}_{\mathrm{A}}$
$v'_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$
Condition (ii)

Fig. 2. A simulation $r$ from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$

```
  red BCP-istep1 .
close
```

Such proofs are called *proof scores*. The keyword `open` makes a temporary module that imports a given module and the keyword `close` destroys such a temporary module. Each fragment enclosed with `open` and `close` in proof scores is called a *proof passage*. The proof score consists of two proof passages. CafeOBJ returns `true` for each of the two proof passages, which means that $\forall v : \mathcal{R}_{\mathcal{S}_{\mathrm{BCP}}} . \mathrm{rc0}(v)$ has been proved.

A survey of proof scores in CafeOBJ is described in [7].

## 4 Simulations from OTSs to OTSs

Simulations from OTSs to OTSs are defined as follows:

**Definition 4.1** Given OTSs $\mathcal{S}$ and $\mathcal{S}_{\mathrm{A}}$, $r : \mathcal{R}_{\mathcal{S}}\ \mathcal{R}_{\mathcal{S}_{\mathrm{A}}} \to \mathrm{Bool}$ is called a *simulation* from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$ if it satisfies the following conditions:

(i) For each $v \in \mathcal{I}$, there exists $v_{\mathrm{A}} \in \mathcal{I}_{\mathrm{A}}$ such that $r(v, v_{\mathrm{A}})$.

(ii) For each $v, v' \in \mathcal{R}_{\mathcal{S}}$ and $v_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ such that $r(v, v_{\mathrm{A}})$ and $v \rightsquigarrow_{\mathcal{S}} v'$, there exists $v'_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ such that $r(v', v'_{\mathrm{A}})$ and $v_{\mathrm{A}} \rightsquigarrow^{*}_{\mathcal{S}_{\mathrm{A}}} v'_{\mathrm{A}}$.

Note that if $v_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ and $v_{\mathrm{A}} \rightsquigarrow^{*}_{\mathcal{S}_{\mathrm{A}}} v'_{\mathrm{A}}$, then $v'_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ from Proposition 2.3.□

Figure 2 shows the diagrams corresponding to the two conditions in Definition 4.1. When there exists a simulation $r$ from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$, we may say that $\mathcal{S}$ is simulated by $\mathcal{S}_{\mathrm{A}}$ (in terms of $r$), or $\mathcal{S}_{\mathrm{A}}$ simulates $\mathcal{S}$ (in terms of $r$).

Since Definition 4.1 does not say explicitly that for each $v \in \mathcal{R}_{\mathcal{S}}$, there exists $v_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ such that $r(v, v_{\mathrm{A}})$, we need to have the following proposition:

**Proposition 4.2** *Given arbitrary OTSs $\mathcal{S}, \mathcal{S}_{\mathrm{A}}$ such that there exists a simulation from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$ and an arbitrary such simulation $r$, for each $v \in \mathcal{R}_{\mathcal{S}}$, there exists $v_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ such that $r(v, v_{\mathrm{A}})$.*

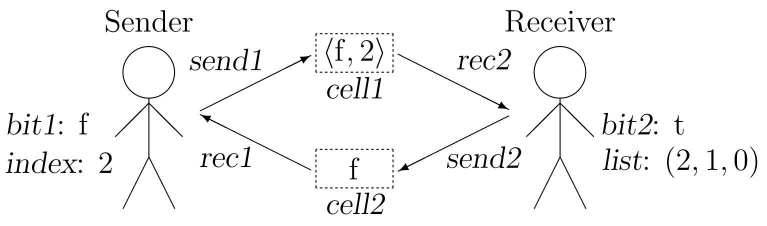

Fig. 3. A snapshot of SCP

**Proof.** By induction on $v \in \mathcal{R}_{\mathcal{S}}$. If $v \in \mathcal{I}$, then there exists $v_{\rm A} \in \mathcal{R}_{\mathcal{S}_{\rm A}}$ such that $r(v, v_{\rm A})$ from Definition 4.1 (i). If $v \rightsquigarrow_{\mathcal{S}} v'$, then there exists $v_{\rm A} \in \mathcal{R}_{\mathcal{S}_{\rm A}}$ such that $r(v, v_{\rm A})$ from the induction hypothesis. Then, from Definition 4.1 (ii), there exists $v'_{\rm A} \in \mathcal{R}_{\mathcal{S}_{\rm A}}$ such that $r(v', v'_{\rm A})$ and $v_{\rm A} \rightsquigarrow^{*}_{\mathcal{S}_{\rm A}} v'_{\rm A}$. □

Then, we have the theorem that guarantees the correctness of the proposed invariant verification method.

**Theorem 4.3** *Given arbitrary OTSs $\mathcal{S}$ and $\mathcal{S}_{\rm A}$ such that there exists a simulation from $\mathcal{S}$ to $\mathcal{S}_{\rm A}$, an arbitrary such simulation $r$ and arbitrary state predicates $p, p_{\rm A}$ such that $p_{\rm A}(v_{\rm A}) \Rightarrow p(v)$ for arbitrary states $v, v_{\rm A}$ with $r(v, v_{\rm A})$, if $\forall v_{\rm A} : \mathcal{R}_{\mathcal{S}_{\rm A}} .\, p_{\rm A}(v_{\rm A})$, then $\forall v : \mathcal{R}_{\mathcal{S}} .\, p(v)$.*

**Proof.** For an arbitrary $v \in \mathcal{R}_{\mathcal{S}}$, there exists $v_{\rm A} \in \mathcal{R}_{\mathcal{S}_{\rm A}}$ with $r(v, v_{\rm A})$ from Proposition 4.2. Therefore, $p(v)$ holds from the assumption. □

**Example 4.4** Let us consider another communication protocol, which is a simplified version of the alternating bit protocol (ABP). The communication protocol is called a *simple communication protocol* (*SCP*). SCP uses unreliable cells as communication channels, while ABP uses unreliable queues. Cells used are unreliable in that their contents can be lost. Figure 3 shows a snapshot of SCP. The sender intends to send the receiver natural numbers from 0 in increasing order. SCP uses two cells: one (*cell1*) is used for sending pairs of Boolean values and natural numbers from the sender to the receiver, and the other (*cell2*) for sending Boolean values from the receiver to the sender. The sender has a Boolean variable (*bit1*) and a natural number variable (*index*), and the receiver has a Boolean variable (*bit2*) and a list (*list*) of natural numbers. Initially, both *cell1* and *cell2* are empty, both *bit1* and *bit2* are false, *index* is 0, and *list* is nil.

What the sender does is as follows:

- *send1*: The sender repeatedly puts a pair of *bit1* and *index* into *cell1*.
- *rec1*: The sender gets a Boolean value $b$ from *cell2* when it is not empty. If $b$ is different from *bit1*, *bit1* is complemented and *index* is incremented.

What the receiver does is as follows:

- *send2*: The receiver repeatedly puts *bit2* into *cell2*.
- *rec2*: The receiver gets a pair of a Boolean value $b$ and a natural number $n$ from *cell1* when it is not empty. If $b$ is the same as *bit2*, *bit2* is complemented and $n$ is added to *list*.

The unreliability of the two cells is realized as follows:

- *drop1*: *cell1* becomes empty if it is not empty.
- *drop2*: *cell2* becomes empty if it is not empty.

The OTS $\mathcal{S}_{\mathrm{SCP}}$ modeling SCP is as follows:

$$\begin{aligned}
\mathcal{O}_{\mathrm{SCP}} &\triangleq \{\mathrm{cell1} : \Upsilon \to \mathrm{PCell}, \mathrm{cell2} : \Upsilon \to \mathrm{BCell}, \mathrm{bit1} : \Upsilon \to \mathrm{Bool},\\
&\qquad \mathrm{index} : \Upsilon \to \mathrm{Nat}, \mathrm{bit2} : \Upsilon \to \mathrm{Bool}, \mathrm{list} : \Upsilon \to \mathrm{List}\}\\
\mathcal{I}_{\mathrm{SCP}} &\triangleq \{\upsilon \mid \mathrm{cell1}(\upsilon) = \mathrm{empty} \wedge \mathrm{cell2}(\upsilon) = \mathrm{empty} \wedge \mathrm{bit1}(\upsilon) = \mathrm{false} \wedge\\
&\qquad \mathrm{index}(\upsilon) = 0 \wedge \mathrm{bit2}(\upsilon) = \mathrm{false} \wedge \mathrm{list}(\upsilon) = \mathrm{nil}\}\\
\mathcal{T}_{\mathrm{SCP}} &\triangleq \{\mathrm{send1} : \Upsilon \to \Upsilon, \mathrm{rec1} : \Upsilon \to \Upsilon, \mathrm{send2} : \Upsilon \to \Upsilon, \mathrm{rec2} : \Upsilon \to \Upsilon,\\
&\qquad \mathrm{drop1} : \Upsilon \to \Upsilon, \mathrm{drop2} : \Upsilon \to \Upsilon\}
\end{aligned}$$

where PCell and BCell are data types for cells of pairs of Boolean values and natural numbers and for cells of Boolean values, respectively.

The six transitions are defined as follows:

$$\begin{aligned}
&\mathrm{send1}(\upsilon) \triangleq \upsilon' \ \mathbf{s.t.}\\
&\quad \mathrm{cell1}(\upsilon') = \mathrm{c}(\langle \mathrm{bit1}(\upsilon), \mathrm{index}(\upsilon)\rangle)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{rec1}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-rec1}(\upsilon)\ \mathbf{s.t.}\\
&\quad \mathrm{cell2}(\upsilon') = \mathrm{empty}\\
&\quad \mathrm{bit1}(\upsilon') = \mathbf{if}\ \mathrm{bit1}(\upsilon) = \mathrm{get}(\mathrm{cell2}(\upsilon))\ \mathbf{then}\ \mathrm{bit1}(\upsilon)\ \mathbf{else}\ \neg\mathrm{bit1}(\upsilon)\\
&\quad \mathrm{index}(\upsilon') = \mathbf{if}\ \mathrm{bit1}(\upsilon) = \mathrm{get}(\mathrm{cell2}(\upsilon))\ \mathbf{then}\ \mathrm{index}(\upsilon)\ \mathbf{else}\ \mathrm{index}(\upsilon) + 1\\
&\mathbf{where}\ \text{c-rec1}(\upsilon) \triangleq \mathrm{cell2}(\upsilon) \neq \mathrm{empty}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{send2}(\upsilon) \triangleq \upsilon' \ \mathbf{s.t.}\\
&\quad \mathrm{cell2}(\upsilon') = \mathrm{c}(\mathrm{bit2}(\upsilon))
\end{aligned}$$

$$\begin{aligned}
&\mathrm{rec2}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-rec2}(\upsilon)\ \mathbf{s.t.}\\
&\quad \mathrm{cell1}(\upsilon') = \mathrm{empty}\\
&\quad \mathrm{bit2}(\upsilon') = \mathbf{if}\ \mathrm{bit2}(\upsilon) = \mathrm{fst}(\mathrm{get}(\mathrm{cell1}(\upsilon)))\ \mathbf{then}\ \neg\mathrm{bit2}(\upsilon)\ \mathbf{else}\ \mathrm{bit2}(\upsilon)\\
&\quad \mathrm{list}(\upsilon') = \mathbf{if}\ \mathrm{bit2}(\upsilon) = \mathrm{fst}(\mathrm{get}(\mathrm{cell1}(\upsilon)))\\
&\qquad\qquad \mathbf{then}\ (\mathrm{snd}(\mathrm{get}(\mathrm{cell1}(\upsilon)))\ \mathrm{list}(\upsilon))\ \mathbf{else}\ \mathrm{list}(\upsilon)\\
&\mathbf{where}\ \text{c-rec2}(\upsilon) \triangleq \mathrm{cell1}(\upsilon) \neq \mathrm{empty}
\end{aligned}$$

$$\begin{aligned}
&\mathrm{drop1}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-drop1}(\upsilon)\ \mathbf{s.t.}\\
&\quad \mathrm{cell1}(\upsilon') = \mathrm{empty}\\
&\mathbf{where}\ \text{c-drop1}(\upsilon) \triangleq \mathrm{cell1}(\upsilon) \neq \mathrm{empty}
\end{aligned}$$

$$\begin{array}{l}\mathrm{drop2}(\upsilon) \triangleq \upsilon' \ \textbf{if}\ \text{c-drop2}(\upsilon)\ \textbf{s.t.}\\ \quad \mathrm{cell2}(\upsilon') = \mathrm{empty}\\ \textbf{where}\ \text{c-drop2}(\upsilon) \triangleq \mathrm{cell2}(\upsilon) \neq \mathrm{empty}\end{array}$$

where c is the constructor of cells, get gets the content of a given cell if it is not empty, and fst and snd extract the first and second elements of a given pair, respectively.

The reliable communication property wrt SCP can be formalized as an invariant property wrt $\mathcal{S}_{\mathrm{SCP}}$ with the state predicate that is defined as follows:

$$\begin{array}{rl}\mathrm{rc1}(\upsilon) \triangleq & (\mathrm{bit1}(\upsilon) = \mathrm{bit2}(\upsilon) \Rightarrow \mathrm{mk}(\mathrm{index}(\upsilon)) = (\mathrm{index}(\upsilon)\ \mathrm{list}(\upsilon))) \wedge\\ & (\mathrm{bit1}(\upsilon) \neq \mathrm{bit2}(\upsilon) \Rightarrow \mathrm{mk}(\mathrm{index}(\upsilon)) = \mathrm{list}(\upsilon))\end{array}$$

The verification that SCP satisfies the reliable communication property is equivalent to the proof of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{SCP}}}.\, \mathrm{rc1}(\upsilon)$.

If we find a simulation $r1$ from $\mathcal{S}_{\mathrm{SCP}}$ to $\mathcal{S}_{\mathrm{BCP}}$ such that $\mathrm{rc0}(\upsilon_0) \Rightarrow \mathrm{rc1}(\upsilon_1)$ for arbitrary states $\upsilon_0, \upsilon_1$ with $\mathrm{r1}(\upsilon_1, \upsilon_0)$, then completed is the verification that SCP satisfies the property. One candidate for such simulations is as follows:

$$\begin{array}{rl}\mathrm{r1}(\upsilon_1, \upsilon_0) \triangleq & (\textbf{if}\ \mathrm{bit1}(\upsilon_1) = \mathrm{bit2}(\upsilon_1)\\ & \quad \textbf{then}\ \mathrm{index}(\upsilon_1) = \mathrm{index}(\upsilon_0)\ \textbf{else}\ \mathrm{index}(\upsilon_1) + 1 = \mathrm{index}(\upsilon_0))\\ & \quad \wedge\ (\mathrm{list}(\upsilon_1) = \mathrm{list}(\upsilon_0))\end{array}$$

We describe the proof of $\mathrm{rc0}(\upsilon_0) \Rightarrow \mathrm{rc1}(\upsilon_1)$ for arbitrary states $\upsilon_0, \upsilon_1$ with $\mathrm{r1}(\upsilon_1, \upsilon_0)$ and the proof that the candidate is really a simulation from $\mathcal{S}_{\mathrm{SCP}}$ to $\mathcal{S}_{\mathrm{BCP}}$ in the coming section. □

## 5 Simulation-based Invariant Verification

The simulation-based invariant verification method proposed in this paper is described. When we want to verify that an OTS $\mathcal{S}$ satisfies an invariant property $p$, namely to prove $\forall \upsilon \in \mathcal{R}_{\mathcal{S}}.\, p(\upsilon)$, all we have to do is as follows:

(i) To make another OTS $\mathcal{S}_{\mathrm{A}}$, which is more abstract than $\mathcal{S}$ and seems clearly to satisfy an invariant property $p_{\mathrm{A}}$ that corresponds to $p$.

(ii) To verify that $\mathcal{S}_{\mathrm{A}}$ satisfies $p_{\mathrm{A}}$, namely to prove $\forall \upsilon_{\mathrm{A}} : \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}.\, p_{\mathrm{A}}(\upsilon_{\mathrm{A}})$.

(iii) To conjecture a simulation candidate $r$ from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$ such that $p_{\mathrm{A}}(\upsilon_{\mathrm{A}})$ implies $p(\upsilon)$ for each $\upsilon, \upsilon_{\mathrm{A}}$ such that $r(\upsilon, \upsilon_{\mathrm{A}})$.

(iv) To prove that $r$ is a simulation from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$.

We describe the verification that SCP satisfies the reliable communication property by proving $\mathrm{rc0}(\upsilon_0) \Rightarrow \mathrm{rc1}(\upsilon_1)$ for arbitrary states $\upsilon_0, \upsilon_1$ with $\mathrm{r1}(\upsilon_1, \upsilon_0)$ and that r1 is a simulation from $\mathcal{S}_{\mathrm{SCP}}$ to $\mathcal{S}_{\mathrm{BCP}}$.

### 5.1 Specification of SCP

$\mathcal{S}_{\text{SCP}}$ is specified in the module SCP. In the module, declared are the following hidden sort, hidden constant, observation operators and action operators:

```
*[Sys1]*
op  init  : -> Sys1
bop cell1 : Sys1 -> PCell  bop cell2 : Sys1 -> BCell
bop bit1  : Sys1 -> Bool   bop bit2  : Sys1 -> Bool
bop index : Sys1 -> Nat    bop list  : Sys1 -> List
bop send1 : Sys1 -> Sys1   bop rec1  : Sys1 -> Sys1
bop send2 : Sys1 -> Sys1   bop rec2  : Sys1 -> Sys1
bop drop1 : Sys1 -> Sys1   bop drop2 : Sys1 -> Sys1
```

The set of equations defining init is as follows:

```
eq cell1(init) = empty .  eq cell2(init) = empty .
eq bit1(init)  = false .  eq bit2(init)  = false .
eq index(init) = 0 .      eq list(init)  = nil  .
```

In this paper, the set of equations defining rec2 is shown:

```
op c-rec2 : Sys1 -> Bool
eq c-rec2(S) = not(cell1(S) = empty) .
ceq cell1(rec2(S)) = empty if c-rec2(S) .
eq  cell2(rec2(S)) = cell2(S) .
eq  bit1(rec2(S))  = bit1(S) .
ceq bit2(rec2(S))  = (if bit2(S) = fst(get(cell1(S)))
        then not bit2(S) else bit2(S) fi) if c-rec2(S) .
eq  index(rec2(S)) = index(S) .
ceq list(rec2(S))  = (if bit2(S) = fst(get(cell1(S)))
        then (snd(get(cell1(S))) list(S)) else list(S) fi)
    if c-rec2(S) .
ceq rec2(S)        = S if not c-rec2(S) .
```

S is a CafeOBJ variable of Sys1.

### 5.2 Verification of SCP on the Reliable Communication Property

In the module INV shown in Subsection 3.2, the module SCP is imported and the following things are declared:

```
op SCP-inv1 : Sys1 -> Bool
eq SCP-inv1(S1)
   = (bit1(S1) = bit2(S1)
      implies (index(S1) list(S1)) = mk(index(S1))) and
```

```
       (not(bit1(S1) = bit2(S1))
        implies list(S1) = mk(index(S1))) .
```

where `S1` is a CafeOBJ variable of `Sys1`, which is declared in `INV`. The operator `SCP-inv1` corresponds to the state predicate rc1.

The module `SIM` is then declared:

```
mod SIM { pr(INV)
  op SCP2BCP-sim1 : Sys1 Sys0 -> Bool
  var S1 : Sys1  var S0 : Sys0
  eq SCP2BCP-sim1(S1,S0)
     = (if bit1(S1) = bit2(S1) then index(S1) = index(S0)
                                else s(index(S1)) = index(S0) fi)
       and (list(S1) = list(S0)) .
}
```

The operator `SCP2BCP-sim1` corresponds to the simulation candidate r1 from $\mathcal{S}_{\text{SCP}}$ to $\mathcal{S}_{\text{BCP}}$.

We first describe the proof of $\text{rc0}(v_0) \Rightarrow \text{rc1}(v_1)$ for arbitrary states $v_0, v_1$ with $\text{r1}(v_1, v_0)$. The proof is done by case splitting. The case is split into five sub-cases. Each sub-case is characterized by a set of equations in proof scores. The five sets of equations for the five sub-cases are as follows[4]:

(i) `(list(s1) = list(s0)) = false`

(ii) `list(s1) = list(s0)`, `bit1(s1) = bit2(s1)`,
`index(s1) = index(s0)`

(iii) `list(s1) = list(s0)`, `bit1(s1) = bit2(s1)`,
`(index(s1) = index(s0)) = false`

(iv) `list(s1) = list(s0)`, `(bit1(s1) = bit2(s1)) = false`,
`index(s0) = s(index(s1))`

(v) `list(s1) = list(s0)`, `(bit1(s1) = bit2(s1)) = false`,
`(index(s0) = s(index(s1))) = false`

where `s0` and `s1` are constants of `Sys0` and `Sys1`, respectively, denoting arbitrary states.

The proof passage of the second sub-case is shown:

```
open SIM
  op s1 : -> Sys1 .  op s0 : -> Sys0 .
  eq list(s1) = list(s0) .  eq bit1(s1) = bit2(s1) .
  eq index(s1) = index(s0) .
  red SCP2BCP-sim1(s1,s0)
```

[4] Note that `s` in `s(index(s1)))` is the successor function of natural numbers.

```
      implies (BCP-inv1(s0) implies SCP-inv1(s1)) .
close
```

CafeOBJ returns `true` for the proof passages, and also for the remaining four proof passages.

For the proof that r1 is a simulation from $\mathcal{S}_{\mathrm{SCP}}$ to $\mathcal{S}_{\mathrm{BCP}}$, one more module called `SIM-ISTEP` is declared:

```
mod SIM-ISTEP { pr(SIM)
  ops s1 s1' : -> Sys1  ops s0 s0' : -> Sys0
  op SCP2BCP-istep1 : Sys0 -> Bool
  var S0' : Sys0
  eq SCP2BCP-istep1(S0')
     = SCP2BCP-sim1(s1,s0) implies SCP2BCP-sim1(s1',S0') .
}
```

The module is used for checking the second condition in Definition 4.1. The constants `s1`, `s0`, `s1'` and `s0'` correspond to $v$, $v_{\mathrm{A}}$, $v'$ and $v'_{\mathrm{A}}$, respectively. The CafeOBJ variable `S0'` in the term `SCP2BCP-istep1(S0')` is interpreted as existentially quantified. `SCP2BCP-istep1(S0')` is the formula to prove for each transition of $\mathcal{S}_{\mathrm{SCP}}$ (or each action of SCP) for the second condition. `SCP2BCP-sim1(s1,s0)` corresponds to the assumption $r(v, v_{\mathrm{A}})$ in the second condition, and the assumption $v \rightsquigarrow_{\mathcal{S}} v'$ in the second condition is written in each proof passage for checking the second condition.

The proof passage of the first condition is shown:

```
open SIM
  red SCP2BCP-sim1(init,init).
close
```

CafeOBJ returns `true` for the proof passage.

The proof of the second condition is first split into the six cases that correspond to the six transitions of $\mathcal{S}_{\mathrm{SCP}}$ (or six actions of SCP), respectively. We describe the case for `rec2` in this paper.

The case for `rec2` is split into eight sub-cases. The eight sets of equations for the eight sub-cases are as follows:

(i) `cell1(s1) = c(< b,n >)`, `bit1(s1) = bit2(s1)`,
`(bit2(s1) = b) = false`

(ii) `cell1(s1) = c(< b,n >)`, `bit1(s1) = bit2(s1)`,
`bit2(s1) = b`, `index(s1) = n`, `((b xor true) = b) = false`

(iii) `cell1(s1) = c(< b,n >)`, `bit1(s1) = bit2(s1)`,
`bit2(s1) = b`, `index(s1) = n`, `(b xor true) = b`

(iv) `cell1(s1) = c(< b,n >)`, `bit1(s1) = bit2(s1)`,
`bit2(s1) = b`, `(index(s1) = n) = false`

(v) `cell1(s1) = c(< b,n >)`, `(bit1(s1) = bit2(s1)) = false`,
`b = bit1(s1)`

(vi) `cell1(s1) = c(< b,n >)`, `(bit1(s1) = bit2(s1)) = false`,
`(b = bit1(s1)) = false`, `b = bit2(s1)`

(vii) `cell1(s1) = c(< b,n >)`, `(bit1(s1) = bit2(s1)) = false`,
`(b = bit1(s1)) = false`, `(b = bit2(s1)) = false`

(viii) `c-rec2(s1) = false`

where the constants `b` and `n` denote an arbitrary Boolean value and an arbitrary natural number, respectively. Note that the equation `cell1(s1) = c(< b,n >)` is equivalent to the equation `c-rec2(s1) = true`.

The proof passage of the second sub-case is as follows:

```
open SIM-ISTEP
  op b : -> Bool .  op n : -> Nat .
  eq cell1(s1) = c(< b,n >) .  eq bit1(s1) = bit2(s1) .
  eq bit2(s1) = b .  eq index(s1) = n .
  eq ((b xor true) = b) = false .
  eq s1' = rec2(s1) .  eq s0' = send(s0) .
  red SCP2BCP-istep1(s0') .
close
```

The equation `s1' = rec2(s1)` corresponds to the assumption $v \rightsquigarrow_{\mathcal{S}} v'$ in the second condition. CafeOBJ returns `true` for the proof passage.

The proof passage of the third sub-case is as follows:

```
open SIM-ISTEP
  op b : -> Bool .  op n : -> Nat .
  eq cell1(s1) = c(< b,n >) .  eq bit1(s1) = bit2(s1) .
  eq bit2(s1) = b .  eq index(s1) = n .  eq (b xor true) = b .
  eq s1' = rec2(s1) .  eq s0' = s0 .
  red eqbool-lemma1(b) implies SCP2BCP-istep1(s0') .
close
```

The proof passage uses the lemma `eqbool-lemma1` on Boolean values. The lemma is as follows:

```
  eq eqbool-lemma1(B) = not((not B) = B) .
```

where `B` is a CafeOBJ variable of `Bool`. CafeOBJ returns `true` for the proof passage.

The proof passage of the fourth sub-case is as follows:

```
open SIM-ISTEP
  op b : -> Bool .  op n : -> Nat .
  eq cell1(s1) = c(< b,n >) .  eq bit1(s1) = bit2(s1) .
  eq bit2(s1) = b .  eq (index(s1) = n) = false .
  eq s1' = rec2(s1) .  eq s0' = s0 .
  red SCP-inv3(s1) implies SCP2BCP-istep1(s0') .
close
```

The proof passage uses the invariant property `SCP-inv3` wrt $\mathcal{S}_{\mathrm{SCP}}$. The invariant property declared in the module `INV` is as follows:

```
  eq SCP-inv3(S1) = (not(cell1(S1) = empty) and
                      bit2(S1) = fst(get(cell1(S1)))
                     implies index(S1) = snd(get(cell1(S1)))) .
```

CafeOBJ returns `true` for the proof passage.

The proof passages of the remaining four sub-cases can be written likewise. The proof passage of the sixth sub-case uses the invariant property `SCP-inv4` wrt $\mathcal{S}_{\mathrm{SCP}}$. The invariant property declared in the module `INV` is as follows:

```
  eq SCP-inv4(S1) = (not(cell1(S1) = empty) and
                      bit2(S1) = fst(get(cell1(S1)))
                     implies bit1(S1) = fst(get(cell1(S1)))) .
```

The proof passages of the other sub-cases use neither lemmas nor invariant properties.

The proof passages of the remaining cases for `send1`, `rec1`, `send2`, `drop1` and `drop2` can be written likewise. The case for `send1` uses two lemmas on Boolean values and one invariant property wrt $\mathcal{S}_{\mathrm{SCP}}$. The two lemmas are as follows:

```
  eq eqbool-lemma2(B1,B2,B3) = (B1 = B2 or B1 = B3 or B2 = B3) .
  eq eqbool-lemma3(B1,B2)
     = (not (B1 = B2)) implies (B1 = (not B2)) .
```

The invariant property declared in the module `INV` is as follows:

```
  eq SCP-inv2(S1) = ((cell2(S1) = empty) or
      bit1(S1) = get(cell2(S1)) or bit2(S1) = get(cell2(S1))) .
```

The other cases use neither lemmas nor invariant properties.

## 6 Compositions of Simulations from OTSs to OTSs

The *composition* $s \circ r : A\ C \to \mathrm{Bool}$ of two relations $r : A\ B \to \mathrm{Bool}$ and $s : B\ C \to \mathrm{Bool}$ is defined as follows: for each $a \in A$ and $c \in C$, $s \circ r(a, c)$ if

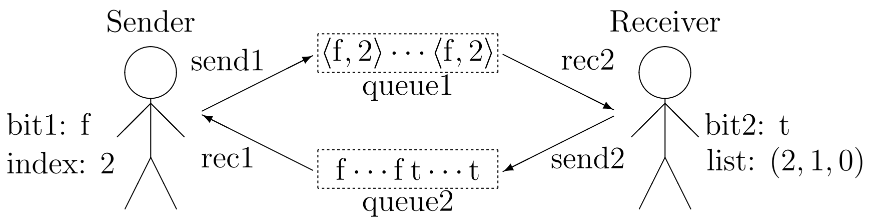

Fig. 4. A snapshot of ABP

and only if there exists $b \in B$ such that $r(a, b)$ and $s(b, c)$.

**Lemma 6.1** *Given arbitrary OTSs $\mathcal{S}$, $\mathcal{S}_{\mathrm{A}}$ and $\mathcal{S}_{\mathrm{B}}$ such that there exist a simulation $r$ from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$ and a simulation $s$ from $\mathcal{S}_{\mathrm{A}}$ to $\mathcal{S}_{\mathrm{B}}$, for each $\upsilon, \upsilon' \in \mathcal{R}_{\mathcal{S}}$ and $\upsilon_{\mathrm{B}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{B}}}$ such that $s \circ r(\upsilon, \upsilon_{\mathrm{B}})$ and $\upsilon \rightsquigarrow_{\mathcal{S}} \upsilon'$, there exists $\upsilon'_{\mathrm{B}}$ such that $s \circ r(\upsilon', \upsilon'_{\mathrm{B}})$ and $\upsilon_{\mathrm{B}} \rightsquigarrow^{*}_{\mathcal{S}_{\mathrm{B}}} \upsilon'_{\mathrm{B}}$.*

**Proof.** There exists $\upsilon_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ such that $r(\upsilon, \upsilon_{\mathrm{A}})$ and $s(\upsilon_{\mathrm{A}}, \upsilon_{\mathrm{B}})$ because of $s \circ r(\upsilon, \upsilon_{\mathrm{B}})$. There also exists $\upsilon'_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ such that $\upsilon_{\mathrm{A}} \rightsquigarrow^{*}_{\mathcal{S}_{\mathrm{A}}} \upsilon'_{\mathrm{A}}$ and $r(\upsilon', \upsilon'_{\mathrm{A}})$ because $r$ is a simulation from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$. All we have to do is to show that there exists $\upsilon'_{\mathrm{B}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{B}}}$ such that $s(\upsilon'_{\mathrm{A}}, \upsilon'_{\mathrm{B}})$ and $\upsilon_{\mathrm{B}} \rightsquigarrow^{*}_{\mathcal{S}_{\mathrm{B}}} \upsilon'_{\mathrm{B}}$. This is proved by induction on $k$ in $\upsilon_{\mathrm{A}} \rightsquigarrow^{k}_{\mathcal{S}_{\mathrm{A}}} \upsilon'_{\mathrm{A}}$.

If $k$ is 0, $\upsilon_{\mathrm{B}}$ is a witness. If $\upsilon_{\mathrm{A}} \rightsquigarrow^{k+1}_{\mathcal{S}_{\mathrm{A}}} \upsilon'_{\mathrm{A}}$, there exists $\upsilon''_{\mathrm{A}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{A}}}$ such that $\upsilon_{\mathrm{A}} \rightsquigarrow^{k}_{\mathcal{S}_{\mathrm{A}}} \upsilon''_{\mathrm{A}}$ and $\upsilon''_{\mathrm{A}} \rightsquigarrow_{\mathcal{S}_{\mathrm{A}}} \upsilon'_{\mathrm{A}}$. There also exists $\upsilon''_{\mathrm{B}}$ such that $s(\upsilon''_{\mathrm{A}}, \upsilon''_{\mathrm{B}})$ and $\upsilon_{\mathrm{B}} \rightsquigarrow^{*}_{\mathcal{S}_{\mathrm{B}}} \upsilon''_{\mathrm{B}}$ from the induction hypothesis. Therefore, we have $\upsilon'_{\mathrm{B}} \in \mathcal{R}_{\mathcal{S}_{\mathrm{B}}}$ such that $s(\upsilon'_{\mathrm{A}}, \upsilon'_{\mathrm{B}})$ and $\upsilon''_{\mathrm{B}} \rightsquigarrow^{*}_{\mathcal{S}_{\mathrm{B}}} \upsilon'_{\mathrm{B}}$ because $s$ is a simulation from $\mathcal{S}_{\mathrm{A}}$ to $\mathcal{S}_{\mathrm{B}}$. Then, $\upsilon'_{\mathrm{B}}$ is a witness. □

**Theorem 6.2** *Given arbitrary OTSs $\mathcal{S}$, $\mathcal{S}_{\mathrm{A}}$ and $\mathcal{S}_{\mathrm{B}}$ such that there exist a simulation $r$ from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$ and a simulation $s$ from $\mathcal{S}_{\mathrm{A}}$ to $\mathcal{S}_{\mathrm{B}}$, the composition $s \circ r$ is a simulation from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{B}}$.*

**Proof.** For each $\upsilon \in \mathcal{I}$, there exists $\upsilon_{\mathrm{B}} \in \mathcal{I}_{\mathrm{B}}$ such that $s \circ r(\upsilon, \upsilon_{\mathrm{B}})$ because for each $\upsilon \in \mathcal{I}$, there exists $\upsilon_{\mathrm{A}} \in \mathcal{I}_{\mathrm{A}}$ such that $r(\upsilon, \upsilon_{\mathrm{A}})$ and for each $\upsilon_{\mathrm{A}} \in \mathcal{I}_{\mathrm{A}}$, there exists $\upsilon_{\mathrm{B}} \in \mathcal{I}_{\mathrm{B}}$ such that $s(\upsilon_{\mathrm{A}}, \upsilon_{\mathrm{B}})$. Then, the first condition that $s \circ r$ is such a simulation is fulfilled. The second condition is also satisfied thanks to Lemma 6.1. □

We describe the verification that $\mathcal{S}_{\mathrm{ABP}}$ is simulated by $\mathcal{S}_{\mathrm{BCP}}$ by showing that there exists a simulation from $\mathcal{S}_{\mathrm{ABP}}$ to $\mathcal{S}_{\mathrm{SCP}}$.

## *6.1 Modeling ABP*

A snapshot of ABP is shown in Figure 4. Queues used are unreliable in that their contents can be lost and duplicated. Like BCP and SCP, the sender intends to send the receiver natural numbers from 0 in increasing order. The only difference between ABP and SCP is that ABP uses two unreliable queues, while SCP uses two unreliable cells. Initially, both *queue1* and *queue2* are empty, both *bit1* and *bit2* are false, *index* is 0, and *list* is nil.

What the sender does is as follows:

- *send1*: The sender repeatedly puts a pair of *bit1* and *index* into *queue1*.
- *rec1*: The sender gets a Boolean value $b$ from *queue2* when it is not empty. If $b$ is different from *bit1*, *bit1* is complemented and *index* is incremented.

What the receiver does is as follows:

- *send2*: The receiver repeatedly puts *bit2* into *queue2*.
- *rec2*: The receiver gets a pair of a Boolean value $b$ and a natural number $n$ from *queue1* when it is not empty. If $b$ is the same as *bit2*, *bit2* is complemented and $n$ is added to *list*.

The unreliability of the two queues is realized as follows:

- *drop1*: The top element of *queue1* is deleted if it is not empty.
- *dup1*: The top element of *queue1* is duplicated if it is not empty.
- *drop2*: The top element of *queue2* is deleted if it is not empty.
- *dup2*: The top element of *queue2* is duplicated if it is not empty.

The OTS $\mathcal{S}_{\mathrm{ABP}}$ modeling ABP is as follows:

$$\begin{aligned}
\mathcal{O}_{\mathrm{ABP}} \triangleq \{&\mathrm{queue1} : \Upsilon \to \mathrm{PQueue}, \mathrm{queue2} : \Upsilon \to \mathrm{BQueue}, \mathrm{bit1} : \Upsilon \to \mathrm{Bool},\\
&\mathrm{index} : \Upsilon \to \mathrm{Nat}, \mathrm{bit2} : \Upsilon \to \mathrm{Bool}, \mathrm{list} : \Upsilon \to \mathrm{List}\}\\
\mathcal{I}_{\mathrm{ABP}} \triangleq \{&\upsilon \mid \mathrm{queue1}(\upsilon) = \mathrm{empty} \wedge \mathrm{queue2}(\upsilon) = \mathrm{empty} \wedge \mathrm{bit1}(\upsilon) = \mathrm{false} \wedge\\
&\mathrm{index}(\upsilon) = 0 \wedge \mathrm{bit2}(\upsilon) = \mathrm{false} \wedge \mathrm{list}(\upsilon) = \mathrm{nil}\}\\
\mathcal{T}_{\mathrm{ABP}} \triangleq \{&\mathrm{send1} : \Upsilon \to \Upsilon, \mathrm{rec1} : \Upsilon \to \Upsilon, \mathrm{send2} : \Upsilon \to \Upsilon, \mathrm{rec2} : \Upsilon \to \Upsilon,\\
&\mathrm{drop1} : \Upsilon \to \Upsilon, \mathrm{dup1} : \Upsilon \to \Upsilon, \mathrm{drop2} : \Upsilon \to \Upsilon, \mathrm{dup2} : \Upsilon \to \Upsilon\}
\end{aligned}$$

where PQueue and BQueue are data types for queues of pairs of Boolean values and natural numbers and for queues of Boolean values, respectively.

The eight transitions are defined as follows:

$$\begin{aligned}
&\mathrm{send1}(\upsilon) \triangleq \upsilon' \ \mathbf{s.t.}\\
&\quad \mathrm{queue1}(\upsilon') = \mathrm{put}(\mathrm{queue1}(\upsilon), \langle \mathrm{bit1}(\upsilon), \mathrm{index}(\upsilon)\rangle)
\end{aligned}$$

$$\begin{aligned}
&\mathrm{rec1}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-rec1}(\upsilon)\ \mathbf{s.t.}\\
&\quad \mathrm{queue2}(\upsilon') = \mathrm{get}(\mathrm{queue2}(\upsilon))\\
&\quad \mathrm{bit1}(\upsilon') = \mathbf{if}\ \mathrm{bit1}(\upsilon) = \mathrm{top}(\mathrm{queue2}(\upsilon))\ \mathbf{then}\ \mathrm{bit1}(\upsilon)\ \mathbf{else}\ \neg\mathrm{bit1}(\upsilon)\\
&\quad \mathrm{index}(\upsilon') = \mathbf{if}\ \mathrm{bit1}(\upsilon) = \mathrm{top}(\mathrm{queue2}(\upsilon))\ \mathbf{then}\ \mathrm{index}(\upsilon)\ \mathbf{else}\ \mathrm{index}(\upsilon) + 1\\
&\mathbf{where}\ \text{c-rec1}(\upsilon) \triangleq \mathrm{queue2}(\upsilon) \neq \mathrm{empty}
\end{aligned}$$

$$\mathrm{send2}(\upsilon) \triangleq \upsilon' \ \mathbf{s.t.}$$

$$
\begin{array}{l}
\quad \mathrm{queue2}(\upsilon') = \mathrm{put}(\mathrm{queue2}(\upsilon), \mathrm{bit2}(\upsilon)) \\[1ex]
\mathrm{rec2}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-rec2}(\upsilon)\ \mathbf{s.t.} \\
\quad \mathrm{queue1}(\upsilon') = \mathrm{get}(\mathrm{queue1}(\upsilon)) \\
\quad \mathrm{bit2}(\upsilon') = \mathbf{if}\ \mathrm{bit2}(\upsilon) = \mathrm{fst}(\mathrm{top}(\mathrm{queue1}(\upsilon)))\ \mathbf{then}\ \neg\mathrm{bit2}(\upsilon)\ \mathbf{else}\ \mathrm{bit2}(\upsilon) \\
\quad \mathrm{list}(\upsilon') = \mathbf{if}\ \mathrm{bit2}(\upsilon) = \mathrm{fst}(\mathrm{top}(\mathrm{queue1}(\upsilon))) \\
\qquad\qquad\quad \mathbf{then}\ (\mathrm{snd}(\mathrm{top}(\mathrm{queue1}(\upsilon)))\ \mathrm{list}(\upsilon))\ \mathbf{else}\ \mathrm{list}(\upsilon) \\
\mathbf{where}\ \text{c-rec2}(\upsilon) \triangleq \mathrm{queue1}(\upsilon) \neq \mathrm{empty}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{drop1}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-drop1}(\upsilon)\ \mathbf{s.t.} \\
\quad \mathrm{queue1}(\upsilon') = \mathrm{get}(\mathrm{queue1}(\upsilon)) \\
\mathbf{where}\ \text{c-drop1}(\upsilon) \triangleq \mathrm{queue1}(\upsilon) \neq \mathrm{empty}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{dup1}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-dup1}(\upsilon)\ \mathbf{s.t.} \\
\quad \mathrm{queue1}(\upsilon') = \mathrm{top}(\mathrm{queue1}(\upsilon))\,, \mathrm{queue1}(\upsilon) \\
\mathbf{where}\ \text{c-dup1}(\upsilon) \triangleq \mathrm{queue1}(\upsilon) \neq \mathrm{empty}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{drop2}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-drop2}(\upsilon)\ \mathbf{s.t.} \\
\quad \mathrm{queue2}(\upsilon') = \mathrm{get}(\mathrm{queue2}(\upsilon)) \\
\mathbf{where}\ \text{c-drop2}(\upsilon) \triangleq \mathrm{queue2}(\upsilon) \neq \mathrm{empty}
\end{array}
$$

$$
\begin{array}{l}
\mathrm{dup2}(\upsilon) \triangleq \upsilon' \ \mathbf{if}\ \text{c-dup2}(\upsilon)\ \mathbf{s.t.} \\
\quad \mathrm{queue2}(\upsilon') = \mathrm{top}(\mathrm{queue2}(\upsilon))\,, \mathrm{queue2}(\upsilon) \\
\mathbf{where}\ \text{c-dup2}(\upsilon) \triangleq \mathrm{queue2}(\upsilon) \neq \mathrm{empty}
\end{array}
$$

where the comma in $\mathrm{top}(\mathrm{queue}i(\upsilon))\,, \mathrm{queue}i(\upsilon)$ for $i = 1, 2$ is the constructor of non-empty queues, top returns the top element of a given queue if it is not empty, get returns the queue obtained by deleting the top element from a given queue if it is not empty, and put returns the queue obtained by adding a given element to a given queue at the end.

The reliable communication property wrt ABP can be formalized as an invariant property wrt $\mathcal{S}_{\mathrm{ABP}}$ with the state predicate that is defined as follows:

$$
\begin{array}{l}
\mathrm{rc2}(\upsilon) \triangleq (\mathrm{bit1}(\upsilon) = \mathrm{bit2}(\upsilon) \Rightarrow \mathrm{mk}(\mathrm{index}(\upsilon)) = (\mathrm{index}(\upsilon)\ \mathrm{list}(\upsilon))) \wedge \\
\qquad\qquad (\mathrm{bit1}(\upsilon) \neq \mathrm{bit2}(\upsilon) \Rightarrow \mathrm{mk}(\mathrm{index}(\upsilon)) = \mathrm{list}(\upsilon))
\end{array}
$$

The verification that ABP satisfies the reliable communication property is equivalent to the proof of $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{ABP}}}.\ \mathrm{rc2}(\upsilon)$.

A simulation candidate r2 from $\mathcal{S}_{\mathrm{ABP}}$ to $\mathcal{S}_{\mathrm{SCP}}$ is as follows:

$$
\begin{array}{l}
\mathrm{r2}(\upsilon_2, \upsilon_1) \triangleq (\mathrm{bit1}(\upsilon_2) = \mathrm{bit1}(\upsilon_1)) \wedge (\mathrm{bit2}(\upsilon_2) = \mathrm{bit2}(\upsilon_1)) \wedge \\
\qquad\qquad (\mathrm{index}(\upsilon_2) = \mathrm{index}(\upsilon_1)) \wedge (\mathrm{lits}(\upsilon_2) = \mathrm{list}(\upsilon_1)) \wedge \\
\qquad\qquad \forall pq : \mathrm{PQueue}.\ ((\mathrm{queue1}(\upsilon_2) = \langle \mathrm{bit1}(\upsilon_2), \mathrm{index}(\upsilon_2)\rangle\,, pq) \\
\qquad\qquad\qquad\qquad \Rightarrow\ (\mathrm{cell1}(\upsilon_1) = \mathrm{c}(\langle \mathrm{bit1}(\upsilon_2), \mathrm{index}(\upsilon_2)\rangle))) \wedge \\
\qquad\qquad \forall bq : \mathrm{BQueue}.\ ((\mathrm{queue2}(\upsilon_2) = \mathrm{bit2}(\upsilon_2)\,, bq) \\
\qquad\qquad\qquad\qquad \Rightarrow\ (\mathrm{cell2}(\upsilon_1) = \mathrm{c}(\mathrm{bit2}(\upsilon_2))))
\end{array}
$$

We prove from Theorem 6.2 that $\exists \upsilon_1 : \mathcal{R}_{\mathcal{S}_{\mathrm{SCP}}}.\, \mathrm{r2}(\upsilon_2, \upsilon_1) \wedge \mathrm{r1}(\upsilon_1, \upsilon_0)$ is a simulation from $\mathcal{S}_{\mathrm{ABP}}$ to $\mathcal{S}_{\mathrm{BCP}}$ by proving that r2 is a simulation from $\mathcal{S}_{\mathrm{ABP}}$ to $\mathcal{S}_{\mathrm{SCP}}$ because r1 is a simulation from $\mathcal{S}_{\mathrm{SCP}}$ to $\mathcal{S}_{\mathrm{BCP}}$.

The remaining thing to do so as to prove $\forall \upsilon : \mathcal{R}_{\mathcal{S}_{\mathrm{ABP}}}.\, \mathrm{rc2}(\upsilon)$ is to prove $\mathrm{rc0}(\upsilon_0) \Rightarrow \mathrm{rc2}(\upsilon_2)$ for arbitrary states $\upsilon_0, \upsilon_2$ with $\exists \upsilon_1 : \mathcal{R}_{\mathcal{S}_{\mathrm{SCP}}}.\, \mathrm{r2}(\upsilon_2, \upsilon_1) \wedge \mathrm{r1}(\upsilon_1, \upsilon_0)$ or to prove $\mathrm{rc1}(\upsilon_1) \Rightarrow \mathrm{rc2}(\upsilon_2)$ for arbitrary states $\upsilon_1, \upsilon_2$ with $\mathrm{r2}(\upsilon_2, \upsilon_1)$. Either proof can be straightforwardly done.

## 6.2 *Specification of ABP*

$\mathcal{S}_{\mathrm{ABP}}$ is specified in the module ABP. In the module, declared are the following hidden sort, hidden constant, observation operators and action operators:

```
*[Sys2]*
op  init    : -> Sys2
bop queue1  : Sys2 -> PQueue   bop queue2 : Sys2 -> BQueue
bop bit1    : Sys2 -> Bool     bop bit2   : Sys2 -> Bool
bop index   : Sys2 -> Nat      bop list   : Sys2 -> List
bop send1   : Sys2 -> Sys2     bop rec1   : Sys2 -> Sys2
bop send2   : Sys2 -> Sys2     bop rec2   : Sys2 -> Sys2
bop drop1   : Sys2 -> Sys2     bop dup1   : Sys2 -> Sys2
bop drop2   : Sys2 -> Sys2     bop dup2   : Sys2 -> Sys2
```

The set of equations defining init is as follows:

```
eq queue1(init) = empty .  eq queue2(init) = empty .
eq bit1(init)   = false .  eq bit2(init)   = false .
eq index(init)  = 0 .      eq list(init)   = nil .
```

In this paper, the set of equations defining rec2 is shown:

```
op c-rec2 : Sys2 -> Bool
eq c-rec2(S) = not(queue1(S) = empty) .
ceq queue1(rec2(S)) = get(queue1(S)) if c-rec2(S) .
eq  queue2(rec2(S)) = queue2(S) .
eq  bit1(rec2(S))   = bit1(S) .
ceq bit2(rec2(S))
    = (if bit2(S) = fst(top(queue1(S)))
        then not bit2(S) else bit2(S) fi) if c-rec2(S) .
eq  index(rec2(S))  = index(S) .
ceq list(rec2(S))
    = (if bit2(S) = fst(top(queue1(S)))
       then (snd(top(queue1(S))) list(S)) else list(S) fi)
    if c-rec2(S) .
```

```
  ceq rec2(S)         = S if not c-rec2(S) .
```

`S` is a CafeOBJ variable of `Sys2`.

## 6.3 Verification that ABP is Simulated by BCP

In the module `INV`, the module `ABP` is imported. In the module `SIM`, the following things are declared:

```
  op ABP2SCP-sim1 : Sys2 Sys1 PQueue BQueue -> Bool
  eq ABP2SCP-sim1(S2,S1,PQ,BQ)
     = (bit1(S2) = bit1(S1)) and (bit2(S2) = bit2(S1)) and
       (index(S2) = index(S1)) and (list(S2) = list(S1)) and
       (queue1(S2) = < bit1(S2),index(S2) >,PQ
        implies cell1(S1) = c(< bit1(S1),index(S1) >)) and
       (queue2(S2) = bit2(S2),BQ
        implies cell2(S1) = c(bit2(S1))) .
```

where `S2`, `PQ` and `BQ` are CafeOBJ variables of `Sys2`, `PQueue` and `BQueue`, respectively, which are declared in `SIM`. In the module, also declared are the constants `pq` and `bq` of `PQueue` and `BQueue` denoting arbitrary values of `PQueue` and `BQueue`, respectively.

In the module `SIM-ISTEP`, the following things are declared:

```
  op ABP2SCP-istep1 : Sys1 PQueue BQueue -> Bool
  eq ABP2SCP-istep1(S1',PQ,BQ) = ABP2SCP-sim1(s2,s1,PQ,BQ)
                                 implies ABP2SCP-sim1(s2',S1',PQ,BQ) .
```

`s2` and `s2'` are constants of `Sys2`, and `S1'` is a CafeOBJ variable of `Sys1`. The constants and the variable are declared in the module.

The proof passage of the first condition is shown:

```
open SIM
  red ABP2SCP-sim1(init,init,pq,bq) .
close
```

CafeOBJ returns `true` for the proof passage.

The proof of the second condition is first split into the eight cases that correspond to the eight transitions of $\mathcal{S}_{\text{ABP}}$ (or eight actions of SCP), respectively. We describe the case for `rec2` in this paper.

The case for `rec2` is split into 11 sub-cases. The 11 sets of equations for the 11 sub-cases are as follows:

(i) `queue1(s2) = < b10,n10 >,pq10`, `(bit2(s2) = b10) = false`

(ii) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`(bit1(s2) = b10) = false`

(iii) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `(index(s2) = n10) = false`

(iv) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `index(s2) = n10`, `(bit2(s1) = b10) = false`

(v) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `index(s2) = n10`, `bit2(s1) = b10`,
`(list(s1) = list(s2)) = false`

(vi) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `index(s2) = n10`, `bit2(s1) = b10`,
`list(s1) = list(s2)`, `(index(s1) = n10) = false`

(vii) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `index(s2) = n10`, `bit2(s1) = b10`,
`list(s1) = list(s2)`, `index(s1) = n10`, `(bit1(s1) = b10) = false`

(viii) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `index(s2) = n10`, `bit2(s1) = b10`,
`list(s1) = list(s2)`, `index(s1) = n10`, `bit1(s1) = b10`,
`(cell1(s1) = c(< b10,n10 >)) = false`

(ix) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `index(s2) = n10`, `bit2(s1) = b10`,
`list(s1) = list(s2)`, `index(s1) = n10`, `bit1(s1) = b10`,
`cell1(s1) = c(< b10,n10 >)`, `b10 = true`

(x) `queue1(s2) = < b10,n10 >,pq10`, `bit2(s2) = b10`,
`bit1(s2) = b10`, `index(s2) = n10`, `bit2(s1) = b10`,
`list(s1) = list(s2)`, `index(s1) = n10`, `bit1(s1) = b10`,
`cell1(s1) = c(< b10,n10 >)`, `b10 = false`

(xi) `c-rec2(s2) = false`

where the constants `b10`, `n10` and `pq10` denote an arbitrary Boolean value, an arbitrary natural number and an arbitrary queue of pairs of Boolean values and natural numbers, respectively. Note that the equation `queue1(s2) = < b10,n10 >,pq10` is equivalent to the equation `c-rec2(s1) = true`.

The proof passage of the first sub-case is as follows:

```
open SIM-ISTEP
  op b10 : -> Bool .  op n10 : -> Nat .  op pq10 : -> PQueue .
  eq queue1(s2) = < b10,n10 >,pq10 .
  eq (bit2(s2) = b10) = false .
  eq s2' = rec2(s2) .  eq s1' = send1(s1) .
  red ABP2SCP-istep1(s1',pq,bq) .
close
```

CafeOBJ returns `true` for the proof passage.

The proof passage of the second sub-case is as follows:

```
open SIM-ISTEP
  op b10 : -> Bool .  op n10 : -> Nat .  op pq10 : -> PQueue .
  eq queue1(s2) = < b10,n10 >,pq10 .
  eq bit2(s2) = b10 .  eq (bit1(s2) = b10) = false .
  eq s2' = rec2(s2) .  eq s1' = s1 .
  red ABP-inv3(s2) implies ABP2SCP-istep1(s1',pq,bq) .
close
```

The proof passage uses the invariant property `ABP-inv3` wrt $\mathcal{S}_{\mathrm{ABP}}$. The invariant property declared in the module `INV` is as follows:

```
  eq ABP-inv3(S2)
     = (not(queue1(S2) = empty)
           and bit2(S2) = fst(top(queue1(S2))))
       implies
       (bit1(S2) = fst(top(queue1(S2)))
           and index(S2) = snd(top(queue1(S2)))) .
```

CafeOBJ returns `true` for the proof passage.

The proof passage of the eighth sub-case is as follows:

```
open SIM-ISTEP
  op b10 : -> Bool .  op n10 : -> Nat .  op pq10 : -> PQueue .
  eq queue1(s2) = < b10,n10 >,pq10 .
  eq bit2(s2) = b10 .  eq bit1(s2) = b10 .  eq index(s2) = n10 .
  eq bit2(s1) = b10 .  eq list(s1) = list(s2) .
  eq index(s1) = n10 .  eq bit1(s1) = b10 .
  eq (cell1(s1) = c(< b10,n10 >)) = false .
  eq s2' = rec2(s2) .  eq s1' = s1 .
  red ABP2SCP-sim1(s2,s1,pq10,bq)
      implies ABP2SCP-istep1(s1',pq,bq) .
close
```

The proof passage uses another instance `ABP2SCP-sim1(s2,s1,pq10,bq)` of `ABP2SCP-sim1(s2,s1,PQ,BQ)`. CafeOBJ returns `true` for the proof passage.

The proof passages of the remaining eight sub-cases can be written likewise. The proof passage of the third sub-case uses the invariant property `ABP-inv3` wrt $\mathcal{S}_{\mathrm{ABP}}$, while the others use neither lemmas nor invariant properties.

The proof passages of the remaining cases for `send1`, `rec1`, `send2`, `drop1`, `dup1`, `drop2` and `dup2` can be written likewise. The case for `send1` uses one invariant property wrt $\mathcal{S}_{\mathrm{SCP}}$:

```
eq ABP-inv2(S2) = (not(queue2(S2) = empty)
                    and not(bit1(S2) = top(queue2(S2))))
                   implies (bit2(S2) = top(queue2(S2))) .
```

The invariant property is declared in the module `INV`. The other cases use neither lemmas nor invariant properties.

## 7 Related Work

Simulations have been intensively used to prove that I/O automata satisfy trace properties [12]. In I/O automata, actions are largely classified into external and internal ones, which the definitions of simulations from I/O automata to I/O automata take into consideration. Traces are sequences of external actions. That an I/O automaton satisfies a trace property that is a set of traces means that every trace generated by the I/O automaton is a member of the set. On the other hand, there is only one kind of transition in OTSs and every OTS transition corresponds to internal actions in I/O automata.

(Bi)simulations have also been used to prove that a process is equivalent to another in a sense in process algebras such as CCS [13]. In process algebras, actions are also classified into external and internal ones.

Refinement has been explored in some formal specification languages such as Z [3], B [1] and VDM [10]. In this paper, we take a look at refinement in Event-B [2] because discrete models in Event-B are similar to OTSs. When each event (transition) in a concrete model $\mathsf{N}$ is supposed to refine a certain event in an abstract model $\mathsf{M}$, there is one constraint (called REF1) to prove:

$$\begin{aligned}&P(s,c) \wedge I(s,c,v) \wedge Q(s,t,c,d) \wedge J(s,t,c,d,v,w) \wedge\\ &H(s,t,c,d,w) \wedge S(s,t,c,d,w,w')\\ &\Rightarrow G(s,c,v) \wedge \exists v'.\,(R(s,c,v,v') \wedge J(s,t,c,d,v',w'))\end{aligned}$$

where $v$ is a collection of variables in $\mathsf{M}$, $s$ and $c$ are carrier sets and constants in the context $\mathsf{C}$ that $\mathsf{M}$ sees, $P(s,c)$ is properties that characterize $\mathsf{C}$, $I(s,c,v)$ is invariant properties wrt $\mathsf{M}$, $w$ is a collection of variables in $\mathsf{N}$, $t$ and $d$ are newly added carrier sets and constants in the context $\mathsf{D}$ that $\mathsf{N}$ sees (namely that $s$ and $t$ are carrier sets in $\mathsf{D}$, and $c$ and $d$ are constants in $\mathsf{D}$), $Q(s,t,c,d)$ is properties that characterize $\mathsf{D}$, $J(s,t,c,d,v,w)$ is the gluing invariant that glues the state of $\mathsf{N}$ to that of $\mathsf{M}$, $G(s,c,v)$ and $R(s,c,v,v')$ are the guard and before-after predicate of the event in $\mathsf{M}$, $v'$ is a collection of variables in $\mathsf{M}$ after the event, $H(s,t,c,d,w)$ and $S(s,t,c,d,w,w')$ are the guard and before-after predicate of the event in $\mathsf{N}$, and $w'$ is a collection of variables in $\mathsf{N}$ after the event.

When new events are introduced in a refinement, there are three constraints (called REF2, REF3 and REF4) to prove. REF2 is concerned with invariant properties, while REF3 and REF4 with reachability (or leads-to) properties. Since this paper is only interested in invariant properties, REF2 is only shown:

$$\begin{aligned}&P(s,c) \wedge I(s,c,v) \wedge Q(s,t,c,d) \wedge J(s,t,c,d,v,w) \wedge \\ &H(s,t,c,d,w) \wedge S(s,t,c,d,w,w') \\ &\Rightarrow J(s,t,c,d,v,w')\end{aligned}$$

where $H(s,t,c,d,w)$ and $S(s,t,c,d,w,w')$ are the guard and before-after predicate of the newly introduced event in $\mathsf{N}$.

REF1 and REF2 correspond to the second condition in Definition 4.1. Since OTSs are made as closed systems and do not have any contexts, the second condition in Definition 4.1 does not have anything that corresponds to $P(s,c)$ and $Q(s,t,c,d)$. $J(s,t,c,d,v,w)$ corresponds to $r$ in Definition 4.1. Although Definition 4.1 does not mention explicitly any invariant properties wrt $\mathcal{S}$ and $\mathcal{S}_{\mathrm{A}}$, we can (and need to) use invariant properties wrt $\mathcal{S}$ and/or $\mathcal{S}_{\mathrm{A}}$, which can exclude non-reachable states, so as to prove that $r$ is really a simulation from $\mathcal{S}$ to $\mathcal{S}_{\mathrm{A}}$. $I(s,c,v)$ and part of $J(s,t,c,d,v,w)$ correspond to such invariant properties. One concrete event corresponds to one or zero abstract event in a refinement of Event-B, while one concrete transition corresponds to zero or more abstract transitions (or a sequence of abstract transitions) in a simulation from OTSs to OTSs. Moreover, one concrete transition may correspond to multiple sequences of abstract transitions in a simulation from OTSs to OTSs. In Subsection 5.2, we have described the case for `rec2` in the proof of the second condition, which is split into eight sub-cases. We have shown the proof passages of the three sub-cases out of eight. The first proof passage says that the concrete transition (action) `rec2` corresponds to the abstract transition (action) `send`, while the second and third proof passages say that `rec2` corresponds to the empty sequence.

We make a comparison of the OTS/CafeOBJ method with I/O automata, Z, B, VDM and Event-B on one more point: whether do events (or transitions or actions) have guards (outside which events cannot occur), preconditions (outside which events can occur but their outcomes are not defined) or both. Actions have guards in I/O automata and events have preconditions in Z, B and VDM, while Event-B has both gourds and preconditions. Effective conditions that transitions have in the OTS/CafeOBJ method are more similar to guards than preconditions, but are not exactly the same as guards. In the OTS/CafeOBJ method, transitions can be applied in any state even if their effective conditions do not hold in the state. If transitions are applies in a state where their effective conditions do not hold, however, nothing changes.

## 8 Conclusion

We have proposed a way to verify that an OTS $\mathcal{S}$ satisfies invariant properties based on a simulation from $\mathcal{S}$ to another OTS, which is more abstract than $\mathcal{S}$, in the OTS/CafeOBJ method. Two case studies have been described, in which we prove that SCP is simulated by SCP, and ABP is simulated by SCP and BCP.

One inevitable question is as follows: which of the simulation-based verification method or the induction-based verification method should be used to verify that an OTS satisfies an invariant property? This must depend on the OTS and the invariant property. When the cost of the tasks (i), (ii), (iii) and (iv) in the simulation-based invariant verification method is smaller than the cost of the proof of $\forall v : \mathcal{R}_{\mathcal{S}}.\, p(v)$ by induction on $v$, the former can be rewarded. In the case studies described in the paper, however, we did not find any clear evidence showing that one method is superior to the other. The reason is as follows. The first case study needs three invariant properties wrt $\mathcal{S}_{\mathrm{SCP}}$ and three lemmas on Boolean values. When we prove $\forall v : \mathcal{R}_{\mathrm{S_{SPC}}}.\, \mathrm{rc1}(v)$ by induction on $v$, we need exactly the same invariant properties wrt $\mathcal{S}_{\mathrm{SCP}}$ and lemmas on Boolean values. The second case study needs two invariant properties wrt $\mathcal{S}_{\mathrm{ABP}}$. When we prove $\forall v : \mathcal{R}_{\mathrm{S_{ABC}}}.\, \mathrm{rc2}(v)$ by induction on $v$, we need exactly the same invariant properties wrt $\mathcal{S}_{\mathrm{ABP}}$.

One piece of our future work is to apply the proposed simulation-based verification method to the Mondex electronic purse system and compare it with the case study described in [11] where the induction-based verification method is used.

## References

[1] Abrial, J.-R., “The B Book – Assigning Programs to Meanings,” Cambridge University Press, 1996.

[2] Abrial, J.-R., *Refinement, decomposition and instantiation of discrete models*, in: *12th International Workshop on Abstract State Machines (12th ASM)*, 2005, pp. 17–40.

[3] Davies, J. and J. Woodcock, “Using Z: Specification, Refinement, and Proof,” Prentice Hall, 1996.

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

[5] Diaconescu, R. and K. Futatsugi, *Behavioural coherence in object-oriented algebraic specification*, Journal of Universal Computer Science **6** (2000), pp. 74–96.

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

[7] Futatsugi, K., *Verifying specifications with proof scores in CafeOBJ*, in: *21st International Conference on Automated Software Engineering (ASE 2006)* (2006), pp. 3–10.

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

[9] Hsiang, J. and N. Dershowitz, *Rewrite methods for clausal and nonclausal theorem proving*, in: *10th EATCS International Colloquium on Automata, Languages, and Programming (10th ICALP)*, LNCS **154** (1983), pp. 331–346.

[10] Jones, C. B., "Systematic Software Development Using VDM," Prentice Hall, 1990.

[11] Kong, W., K. Ogata and K. Futatsugi, *Algebraic approaches to formal analysis of the Mondex electronic purse system*, in: *6th International Conference on Integrated Formal Methods (6th IFM)*, LNCS **4591** (2007), pp. 393–412.

[12] Lynch, N. A., "Distributed Algorithms," Morgan-Kaufmann, 1996.

[13] Milner, R., "Communication and Concurrency," Prentice Hall, 1989.

[14] Ogata, K. and K. Futatsugi, *Formal analysis of the iKP electronic payment protocols*, in: *1st International Symposium on Software Security (1st ISSS)*, LNCS **2609** (2003), pp. 441–460.

[15] Ogata, K. and K. Futatsugi, *Proof scores in the OTS/CafeOBJ method*, in: *6th IFIP WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (6th FMOODS)*, LNCS **2884** (2003), pp. 170–184.

[16] Ogata, K. and K. Futatsugi, *Equational approach to formal analysis of TLS*, in: *25th International Conference on Distributed Computing Systems (25th ICDCS)* (2005), pp. 795–804.

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