Compositional Writing of Proof Scores[*]

Kazuhiro Ogata[†,‡]   Kokichi Futatsugi[‡]

† NEC Software Hokuriku, Ltd.
1 Anyoji, Tsurugi, Ishikawa 920-2141, Japan

‡ Graduate School of Information Science
Japan Advanced Institute of Science and Technology
1-1 Asahidai, Tatsunokuchi, Ishikawa 923-1292, Japan

Abstract

The foundation of a way to write proof scores showing that systems have invariants in algebraic specification languages is described. The way, called the compositional writing of proof scores, has been devised through several case studies. The compositional writing of proof scores makes it possible to divide a formula stating an invariant into reasonably small ones, each of which is proved by writing proof scores individually. This relieves the load to reduce terms denoting logical formulas and decreases the number of subcases into which the case is split in case analysis.
Keywords: algebraic specification, CafeOBJ, OTS, proof scores, specification, verification

1 Introduction

Equations are the most basic logical formulas and equational reasoning is the most fundamental way of reasoning[13], which can moderate the difficulties of

[*]An earlier version of the paper appears in [27].

proofs that might otherwise become too hard to understand. Algebraic specification languages make it possible to describe systems in terms of equations and verify that systems have properties by means of equational reasoning. Writing proofs, or proof scores in algebraic specification languages has been mainly promoted by researchers of the OBJ community[9, 11, 8].

We have been successfully applying such algebraic techniques to modeling, specification and verification of distributed systems such as distributed mutual exclusion algorithms[20, 21, 22] and security protocols[23, 25, 26, 28]. In our method called the OTS/CafeOBJ method, systems are modeled as observational transition systems, or OTSs, which are the definition of transition systems for writing them in terms of equations, and OTSs are written in CafeOBJ[2, 5], an algebraic specification language. We then verify that the OTSs have properties by writing proof scores in CafeOBJ and executing them with the CafeOBJ system[18].

In this paper, we describe the foundation of a way to write proof scores of invariants in the OTS/CafeOBJ method. Invariants are the most basic and important properties among various kinds of properties because proofs of other kinds of properties often need invariants. We have devised the way, called the compositional writing of proof scores, through several case studies. The compositional writing of proof scores makes it possible to divide a formula stating an invariant property under discussion into reasonably small ones, each of which is proved by writing proof scores individually. This relieves the load to reduce terms denoting logical formulas and decreases the number of subcases into which the case is split in case analysis. Invariants are mainly proved by induction, and the proofs of the small formulas may depend on each other in the sense that the proof of one uses some other to strengthen inductive hypotheses and vice versa.

The rest of the paper is organized as follows. Section 2 describes the OTS/CafeOBJ method. We use a small example to exemplify writing proof scores compositionally in Section 3. We describe the foundation of a way to write proof scores compositionally in Section 4, and describe how to write proof scores based on the foundation in Section 5. Section 6 discusses the advantages of the compositional writing of proof scores and compares with related work. We conclude the paper with Section 7.

2 The OTS/CafeOBJ Method

2.1 CafeOBJ in a Nutshell

CafeOBJ[2, 5] can be used to specify abstract machines as well as abstract data types. A visible sort denotes an abstract data type, while a hidden sort the state space of an abstract machine[10, 6]. There are two kinds of operators to hidden sorts, which are action and observation operators. An action operator can change states of an abstract machine. Only observation operators can be used to observe the inside of an abstract machine. An action operator is basically specified with equations by describing how the value returned by each observation operator changes.

Visible sorts are declared by enclosing `[` and `]`, and hidden ones by enclosing `*[` and `]*`. Action and observation operators are declared by starting with `bop`, and other operators by starting with `op`. After `bop` or `op`, an operator name is written, followed by a colon `:` and a list of sorts, and then, `->` and a sort are written. The list of sorts is called the arity of the operator, and the sort after `->` the coarity of the operator. The pair of the arity and coarity is called the rank of the operator. When declaring more than one operators which ranks are the same simultaneously, `bops` and `ops` are used instead of `bop` and `op`. Operators with the empty arity are called constants.

Operators are defined in terms of equations. An equation is declared by starting with `eq`, and a conditional one by starting with `ceq`. After `eq`, two terms connected with `=` are written, ended with a full stop. After `ceq`, two terms connected with `=` are written, followed by `if`, and then, a term denoting the condition and a full stop are written.

The CafeOBJ system uses declared equations as left-to-right rewrite rules and rewrites (or reduces) a given term. This executability makes it possible to simulate a specified system and verify that a specified system has properties.

CafeOBJ inherits OBJ3's powerful module facilities[12]. The CafeOBJ system provides built-in modules, one of which is `BOOL` where visible sort `Bool` denoting truth values is declared. Module `Bool` is implicitly imported unless it is explicitly declared not to import the module. In the module, constants `true` and `false` with the usual meaning, and operators `not_`, `_and_`, `_or_`, `_implies_` and `_iff_` denoting $\neg$, $\wedge$, $\vee$, $\Rightarrow$ and $\Leftrightarrow$ are declared. An underscore `_` indicates the place where an argument is put. The operator `if_then_else_fi` corresponding to the **if** statement in programming languages is also declared.

2.2 Observational Transition Systems

Observational transition systems, or OTSs are the definition of transition systems for writing transition systems in terms of equations. We assume that there exists a universal state space called Υ. We also suppose that each data type used has been defined beforehand, including the equivalence between two values v_1, v_2 of the data type denoted by $v_1 = v_2$. An OTS $\mathcal{S}$ consists of $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ where

- $\mathcal{O}$: A set of observable values. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \rightarrow D$, where D is a data type and may be different for each observable value. Given an OTS $\mathcal{S}$ and two states $v_1, v_2 \in \Upsilon$, the equivalence between two states, denoted by $v_1 =_{\mathcal{S}} v_2$, with respect to $\mathcal{S}$ is defined as $v_1 =_{\mathcal{S}} v_2 \overset{\text{def}}{=} \forall o \in \mathcal{O}. o(v_1) = o(v_2)$.

- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subset \Upsilon$.

- $\mathcal{T}$: A set of conditional transition rules. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon/{=_{\mathcal{S}}} \rightarrow \Upsilon/{=_{\mathcal{S}}}$ on equivalence classes of Υ with respect to $=_{\mathcal{S}}$. Let $\tau(v)$ be the representative element of $\tau([v])$ for each $v \in \Upsilon$ and it is called *the successor state* of v with respect to τ. The condition c_τ for a transition rule $\tau \in \mathcal{T}$, which is a predicate on states, is called *the effective condition*. The effective condition is supposed to satisfy the following requirement: given a state $\upsilon \in \Upsilon$, if c_τ is false in υ, namely τ is not *effective* in υ, then $\upsilon =_{\mathcal{S}} \tau(\upsilon)$.

Multiple similar observable values and transition rules may be indexed. Generally, observable values and transition rules are denoted by $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, respectively, provided that $m, n \geq 0$ and we assume that there exist data types D_k such that $k \in D_k\,(k = i_1, \ldots, i_m, j_1, \ldots, j_n)$. For example, an integer array a possessed by a process p may be denoted by an observable value a_p, and the increment of the ith element of the array may be denoted by a transition rule *inc*-$a_{p,i}$.

An execution of $\mathcal{S}$ is an infinite sequence $\upsilon_0, \upsilon_1, \ldots$ of states satisfying[1]

- *Initiation*: $\upsilon_0 \in \mathcal{I}$,
- *Consecution*: For each $i \in \{0, 1, \ldots\}$, $\upsilon_{i+1} =_{\mathcal{S}} \tau(\upsilon_i)$ for some $\tau \in \mathcal{T}$.

A state is called *reachable* with respect to $\mathcal{S}$ if and only if there exists an execution of $\mathcal{S}$ in which the state appears. Let $\mathcal{R}_{\mathcal{S}}$ be the set of all the reachable states with respect to $\mathcal{S}$.

All properties considered in this paper are invariants[2], which are defined as follows:

$$\mathsf{invariant}\ p \overset{\text{def}}{=} \forall \upsilon \in \mathcal{R}_{\mathcal{S}}.p(\upsilon)\,.$$

[1]If we want to discuss liveness properties, an execution of $\mathcal{S}$ should also satisfy *Fairness*: for each $\tau \in \mathcal{T}$, there exist an infinite number of indexes $i \in \{0, 1, \ldots\}$ such that $\upsilon_{i+1} =_{\mathcal{S}} \tau(\upsilon_i)$.

[2]In addition to invariant properties, there are unless, stable, ensures and leads-to properties, which are inspired by UNITY[3]. The way to write proof scores described in this paper can also be applied to unless, stable, ensures properties.

Let $\boldsymbol{x}$ be all free variables except for one for states in p. We suppose that $\mathsf{invariant}\, p$ is interpreted as $\forall \boldsymbol{x}.(\mathsf{invariant}\, p)$ in this paper. When proof scores are written to prove $\forall \boldsymbol{x}.(\mathsf{invariant}\, p)$, $\boldsymbol{x}$ are replaced with constants that denote arbitrary values corresponding to $\boldsymbol{x}$ and the universally quantifier is eliminated. If a variable is existentially quantified in $\mathsf{invariant}\, p$, the variable is replaced with a Skolem constant or function and the existential quantifier is eliminated.

2.3 Description of OTSs in CafeOBJ

An OTS $\mathcal{S}$ is described in CafeOBJ. The universal state space Υ is denoted by a hidden sort, say H.

An observable value $o_{i_1,\ldots,i_m} \in \mathcal{O}$ is denoted by a CafeOBJ observation operator. We assume that the data types $D_k\,(k = i_1, \ldots, i_m)$ and D are described and there exist visible sorts $V_k\,(k = i_1, \ldots, i_m)$ and V corresponding to the data types. The CafeOBJ observation operator denoting $o_{i_1,\ldots,i_m}$ is declared as follows:

$$\texttt{bop}\ o\ \texttt{:}\ H\ V_{i_1}\ \ldots\ V_{i_m}\ \texttt{->}\ V$$

Any initial state in $\mathcal{I}$ is denoted by a constant, say *init*, which is declared as follows:

$$\texttt{op}\ \mathit{init}\ \texttt{: ->}\ H$$

Suppose that the initial value of $o_{i_1,\ldots,i_m}$ is $f(i_1, \ldots, i_m)$. This is expressed by this equation

$$\texttt{eq}\ o(\mathit{init}, X_{i_1}, \ldots, X_{i_m})\ \texttt{=}\ f(X_{i_1}, \ldots, X_{i_m})\ \texttt{.}$$

where each X_k is a CafeOBJ variable for V_k where $k = i_1, \ldots, i_m$ and $f(X_{i_1}, \ldots, X_{i_m})$ is a term denoting $f(i_1, \ldots, i_m)$.

A transition rule $\tau_{j_1,\ldots,j_n} \in \mathcal{T}$ is denoted by a CafeOBJ action operator. We assume that the data types $D_k\,(k = j_1, \ldots, j_n)$ are described and there exist visible sorts $V_k\,(k = j_1, \ldots, j_n)$ corresponding to the data types. The CafeOBJ action operator denoting $\tau_{j_1,\ldots,j_n}$ is declared as follows:

$$\texttt{bop}\ \mathrm{a} : H\ V_{j_1}\ \ldots\ V_{j_n}\ \texttt{->}\ H$$

If $\tau_{j_1,\ldots,j_n}$ is applied in a state in which it is effective, the value returned by $o_{i_1,\ldots,i_m}$ may change, which can be generally described in CafeOBJ as follows:

$$\begin{aligned}&\texttt{ceq}\ o(\mathrm{a}(S, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m})\\ &\quad \texttt{=}\ e\text{-}\mathrm{a}(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})\\ &\quad\ \texttt{if}\ c\text{-}\mathrm{a}(S, X_{j_1}, \ldots, X_{j_n})\ .\end{aligned}$$

where S is a CafeOBJ variable for H and each X_k is a CafeOBJ variable for V_k where $k = i_1, \ldots, i_m, j_1, \ldots, j_n$. $\mathrm{a}(S, X_{j_1}, \ldots, X_{j_n})$ denotes the successor state of S with respect to $\tau_{j_1,\ldots,j_n}$. $e\text{-}\mathrm{a}(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})$ denotes the value returned by $o_{i_1,\ldots,i_m}$ in the successor state, which is determined by only the values returned by observable values in S. $c\text{-}\mathrm{a}(S, X_{j_1}, \ldots, X_{j_n})$ denotes the effective condition $c_{\tau_{j_1,\ldots,j_n}}$ of $\tau_{j_1,\ldots,j_n}$ in S.

If $\tau_{j_1,\ldots,j_n}$ is applied in a state in which it is not effective, the value returned by any observable value does not change. Therefore all we have to do is to declare this equation

$$\texttt{ceq}\ \mathrm{a}(S, X_{j_1}, \ldots, X_{j_n})\ \texttt{=}\ S\ \texttt{if not}\ c\text{-}\mathrm{a}(S, X_{j_1}, \ldots, X_{j_n})\ .$$

If the value returned by $o_{i_1,\ldots,i_m}$ is not affected by applying $\tau_{j_1,\ldots,j_n}$ in any state (regardless of the truth value of $c_{\tau_{j_1,\ldots,j_n}}$), we may declare this equation

$$\texttt{eq}\ o(a(S, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m}) \ \texttt{=}\ o(S, X_{i_1}, \ldots, X_{i_m})\ .$$

3 Example: Mutual Exclusion Algorithm

A mutual exclusion algorithm using a queue is used as an example. The pseudo-code executed by each process i repeatedly can be described as this

```
l1: $put(queue, i)$
l2: **repeat while** $top(queue) = i$
    Critical Section
cs: $get(queue)$
```

queue is the queue of process IDs shared by all processes. $put(queue, i)$ puts a process ID i into *queue* at the end, $get(queue)$ deletes the top element from *queue*, and $top(queue)$ returns the top element of *queue*. They are atomically processed. Moreover, each iteration of the loop at label l2 is supposed to be atomically processed. Initially each process i is at label l1 and *queue* is empty.

In this section, we describe how to write proof scores compositionally, which show that there exists at most one process at any given time in the critical section of the system in which processes execute the pseudo-code. In the two succeeding sections, we will describe the foundation of a way to write proof scores compositionally.

3.1 Description of Example

We use two kinds of observable values and three kinds of transition rules to model the system as an OTS, which are as follows:

- Observable values.
  - *queue* returns the queue of process IDs shared by all processes. It initially returns the empty queue.
  - pc_i ($i \in Pid$) returns the label of a command that process i will execute next, where Pid is the set of process IDs. Each pc_i initially returns label l1.
- Transition rules.
  - $wait_i$ ($i \in Pid$) denotes that process i executes the command at label l1.
  - try_i ($i \in Pid$) denotes that process i executes one iteration of the loop at label l2.
  - $exit_i$ ($i \in Pid$) denotes that process i executes the command at label cs.

The OTS is described in CafeOBJ. The signature of the CafeOBJ specification is as follows:

```
  *[Sys]*
-- any initial state
```

```
  op init : -> Sys
-- observation operators
  bop pc    : Sys Pid -> Label
  bop queue : Sys -> Queue
-- action operators
  bop want : Sys Pid -> Sys
  bop try  : Sys Pid -> Sys
  bop exit : Sys Pid -> Sys
```

A comment starts with `--` and terminates at the end of the line. `Sys` is the hidden sort denoting the state space Υ. `Pid`, `Label` and `Queue` are the visible sorts denoting process IDs, labels and queues of process IDs, respectively. Constant `init` denotes any initial state of the system. Observation operators `pc` and `queue` denote observable values pc_i and $queue$, and action operators `want`, `try` and `exit` denote transition rules $want_i$, try_i and $exit_i$.

In the following, let `X`, `I` and `J` be CafeOBJ variables for `Sys`, `Pid` and `Pid`, respectively. Action operator `want` is defined with these equations

```
op c-want : Sys Pid -> Bool
eq c-want(S,I) = (pc(S,I) = l1) .
--
ceq pc(want(S,I),J)
    = (if I = J then l2 else pc(S,J) fi) if c-want(S,I) .
ceq queue(want(S,I)) = put(queue(S),I)   if c-want(S,I) .
ceq want(S,I)        = S                 if not c-want(S,I) .
```

`put(queue(S),I)` denotes the queue obtained by putting `I` into the queue denoted by `queue(S)` at the end.

Action operator `try` is defined with these equations

```
op c-try : Sys Pid -> Bool
eq c-try(S,I) = (pc(S,I) = l2 and top(queue(S)) = I) .
--
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) .
```

`top(queue(S))` denotes the top element of the queue denoted by `queue(S)`.

Action operator `exit` is defined with these equations

```
op c-exit : Sys Pid -> Bool
eq c-exit(S,I) = (pc(S,I) = cs) .
--
ceq pc(exit(S,I),J)
    = (if I = J then l1 else pc(S,J) fi) if c-exit(S,I) .
ceq queue(exit(S,I)) = get(queue(S))     if c-exit(S,I) .
ceq exit(S,I)        = S                 if not c-exit(S,I) .
```

`get(queue(S))` denotes the queue obtained by deleting the top element from the queue denoted by `queue(S)`.

3.2 Verification of Example

We verify that there exists at most one process at any given time in the critical section of the system, namely that the algorithm is mutually exclusive. To this end, all we have to do is to prove the following predicate invariant to the OTS modeling the system:

$$(pc_i(\upsilon) = \mathrm{cs} \wedge pc_j(\upsilon) = \mathrm{cs}) \Rightarrow (i = j)\,. \tag{1}$$

To prove (1) invariant to the OTS, we need to prove the following three predicates invariant to the OTS:

$$(pc_i(\upsilon) = \mathrm{cs}) \Rightarrow (\mathit{top}(\mathit{queue}(\upsilon)) = i)\,, \tag{2}$$

$$(pc_i(\upsilon) = \mathrm{l2} \vee pc_i(\upsilon) = \mathrm{cs}) \Rightarrow \neg \mathit{empty?}(\mathit{queue}(\upsilon)\,, \tag{3}$$

$$(pc_i(\upsilon) = \mathrm{l2}) \Rightarrow (i \in \mathit{queue}(\upsilon))\,. \tag{4}$$

empty? is a predicate to check if a given queue is empty and $\in$ is a membership predicate of queues.

We may consider proving (2), (3) and (4) invariant to the OTS first, and then proving (1) invariant to the OTS. But, this strategy does not work in this case because the proofs of (2) and (3) need (1). The dependence of their proofs can be depicted as Figure 1.

We connect the four predicates with conjunctions to make this predicate

$$\begin{array}{ll} & (pc_{i_1}(\upsilon) = \mathrm{cs} \wedge pc_{i_2}(\upsilon) = \mathrm{cs}) \Rightarrow (i_1 = i_2) \\ \wedge & (pc_{i_3}(\upsilon) = \mathrm{cs}) \Rightarrow (\mathit{top}(\mathit{queue}(\upsilon)) = i_3) \\ \wedge & (pc_{i_4}(\upsilon) = \mathrm{l2} \vee pc_{i_4}(\upsilon) = \mathrm{cs}) \Rightarrow \neg \mathit{empty?}(\mathit{queue}(\upsilon) \\ \wedge & (pc_{i_5}(\upsilon) = \mathrm{l2}) \Rightarrow (i_5 \in \mathit{queue}(\upsilon)) \end{array} \tag{5}$$

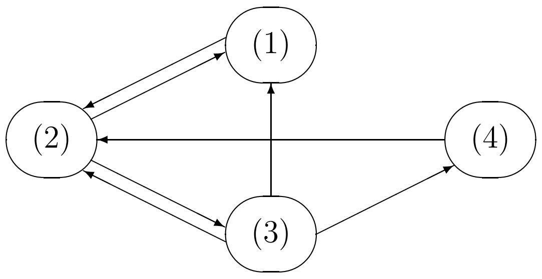

Figure 1: Dependence of proofs

and may consider proving the large predicate invariant to the OTS. This strategy can work. But, the larger the predicate is, the longer and more complicated the corresponding proof score is.

Instead of directly proving (5) invariant to the OTS, we simultaneously prove the four predicates (1), (2), (3) and (4) invariant to the OTS. Although this strategy is similar to the second strategy, this strategy makes it possible to write the proof score of each of the four predicates individually.

We write proof scores to prove the four predicates invariant to the OTS according to the third strategy by induction on the number of transition rules applied. We first write a module, say `INV`, in which the four predicates are expressed as CafeOBJ terms as follows:

```
op inv1 : Sys Pid Pid -> Bool

op inv2 : Sys Pid -> Bool

op inv3 : Sys Pid -> Bool

op inv4 : Sys Pid -> Bool

eq inv1(S,I,J) = (pc(S,I) = cs and pc(S,J) = cs
                 implies I = J) .
eq inv2(S,I) = (pc(S,I) = cs implies top(queue(S)) = I) .
eq inv3(S,I) = (pc(S,I) = l2 or pc(S,I) = cs
                implies not empty?(queue(S))) .
eq inv4(S,I) = (pc(S,I) = l2 implies I \in queue(S)) .
```

In the module, constants `i` and `j` for `Pid` are declared, denoting an arbitrary process ID.

We next write a module, say `ISTEP` that imports module `INV`, in which formulas to prove in each inductive case are expressed as CafeOBJ terms as follows:

```
op istep1 : Pid Pid -> Bool
op istep2 : Pid -> Bool
op istep3 : Pid -> Bool
op istep4 : Pid -> Bool
eq istep1(I,J) = inv1(s,I,J) implies inv1(s',I,J) .
eq istep2(I) = inv2(s,I) implies inv2(s',I) .
eq istep3(I) = inv3(s,I) implies inv3(s',I) .
eq istep4(I) = inv4(s,I) implies inv4(s',I) .
```

`s` and `s'` are constants for `Sys`, denoting an arbitrary state and the successor state after applying a transition rule in the state denoted by `s`. The constants are declared in the module.

Let us consider the inductive case in which any transition rule denoted by action operator `try` preserves (1) denoted by `inv1`. In this inductive case, basically we should prove the formula denoted by this CafeOBJ term

```
inv1(s,i,j) implies inv1(try(s,k),i,j)
```

where `k` is a constant for `Pid`, denoting an arbitrary process ID. But, it is impossible to prove the formula because the inductive hypothesis denoted by `inv1(s,i,j)` is too weak to imply the formula denoted by `inv1(try(s,k),i,j)`. (2) is used to strengthen the inductive hypothesis and then we instead prove the formula denoted by this CafeOBJ term

```
(inv2(s,i) and inv2(s,j) and inv1(s,i,j))
 implies inv1(try(s,k),i,j).
```

Although it is possible to prove this formula, we need to split the state space, or the case. When the effective condition of any transition rule denoted by `try` holds, the case is split into these four subcases

1. $(\mathtt{i} = \mathtt{k}) \wedge (\mathtt{j} = \mathtt{k})$, which needs nothing,
2. $(\mathtt{i} = \mathtt{k}) \wedge (\mathtt{j} \neq \mathtt{k})$, which needs `inv2(s,j)`,
3. $(\mathtt{i} \neq \mathtt{k}) \wedge (\mathtt{j} = \mathtt{k})$, which needs `inv2(s,i)`, and
4. $(\mathtt{i} \neq \mathtt{k}) \wedge (\mathtt{j} \neq \mathtt{k})$, which needs nothing.

We show the passage of the proof score for subcase 2, which is as follows:

```
open ISTEP
-- arbitrary objects
```

```
  op k : -> Pid .
-- assumptions
  -- eq c-try(s,k) = true .
  eq pc(s,k) = l2 .  eq top(queue(s)) = k .
  --
  eq i = k .  eq (j = k) = false .
-- successor state
  eq s' = try(s,k) .
-- check
  red inv2(s,j) implies istep1(i,j) .
close
```

The proof passage is a very basic unit of proof scores, which consists of four parts. Constants are first declared, denoting arbitrary values for intended sorts. In the proof passage, constant `k` is declared, denoting an arbitrary process ID. Equations are secondly declared, denoting the assumptions, or the result of case analysis. In the proof passage, four equations are declared. The first two equations correspond to the effective condition of any transition rule denoted by `try`, and the remaining to subcase 2. An equation is thirdly declared, meaning that `s'` denotes the successor state after applying a transition rule. In this proof passage, the equation means that `s'` denotes the successor state after applying any transition rule denoted by `try` in a state denoted by `s`. A term denoting a formula to prove is finally reduced. In the proof passage, term `inv2(s,j) implies istep1(i,j)` is reduced. The term is actually reduced to `true`, which means that any transition rule denoted by `try` preserves (1) denoted by `inv1` in subcase 2.

One more proof passage is shown. Then let us consider the inductive case in which any transition rule denoted by `exit` preserves (2) denoted by `inv2`. In this inductive case, basically we should prove the formula denoted by this CafeOBJ term

```
inv2(s,i) implies inv2(exit(s,k),i).
```

But, it is impossible to prove the formula because the inductive hypothesis denoted by `inv2(s,i)` is too weak to imply the formula denoted by `inv2(exit(s,k),i)`. (1) is used to strengthen the inductive hypothesis and then we instead prove the formula denoted by this CafeOBJ term

```
(inv1(s,i,k) and inv2(s,i)) implies inv2(exit(s,k),i).
```

We need to split the case in order to prove this formula. When the effective condition of any transition rule denoted by `exit` holds, the case is split into these three subcases

1. $\mathtt{i} = \mathtt{k}$, which needs nothing,
2. $(\mathtt{i} \neq \mathtt{k}) \wedge (\mathtt{pc(s,i)} = \mathtt{cs})$, which needs `inv1(s,i,k)`, and
3. $(\mathtt{i} \neq \mathtt{k}) \wedge (\mathtt{pc(s,i)} \neq \mathtt{cs})$, which needs nothing.

We show the passage of the proof score for subcase 2, which is as follows:

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

The proof passage consists of four pars as the previous proof passage.

We will describe the foundation of a way to write proof scores compositionally in the two succeeding sections, which assures that the compositional writing of proof scores is sound.

4 Compositional Proof of Invariants

Suppose that we prove that a predicate is invariant to a system. The system is first modeled as an OTS, which is written in CafeOBJ. Let H be the hidden sort denoting the state space Υ, and let the predicate be $\mathit{pred}_1(s, x_{11}, \ldots, x_{1m_1})$, where s is a free variable for H and $x_{11}, \ldots, x_{1m_1}$

the other free variables for data types. It is often impossible to prove invariant $\mathit{pred}_1(s, x_{11}, \ldots, x_{1m_1})$ by itself. Suppose that it is possible to prove that $\mathit{pred}_1(s, x_{11}, \ldots, x_{1m_1})$, together with $n-1$ predicates, is invariant to the OTS. Let the $n-1$ predicates be $\mathit{pred}_2(s, x_{21}, \ldots, x_{2m_2})$, ..., $\mathit{pred}_n(s, x_{n1}, \ldots, x_{nm_n})$, and let $\mathit{pred}(s, x_{11}, \ldots, x_{1m_1}, \ldots, x_{n1}, \ldots, x_{nm_n})$ be $\mathit{pred}_1(s, x_{11}, \ldots, x_{1m_1}) \wedge \ldots \wedge \mathit{pred}_n(s, x_{n1}, \ldots, x_{nm_n})$. We may use $\boldsymbol{x}_i$ to denote $x_{i1}, \ldots, x_{im_i}$ where $i = 1, \ldots, n$. Then we prove invariant $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$, instead of the original invariant, from which the original invariant can be deduced.

Proofs of invariants often need induction, especially induction on the number of transition rules applied. Suppose that invariant $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ is proved by induction on the number of transition rules applied. First let us consider the base case and let *init* denote any initial state of the system. For the base case, all we have to do is to prove $\mathit{pred}(\mathit{init}, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. We may prove each $\mathit{pred}_i(\mathit{init}, \boldsymbol{x}_i)$, where $i = 1, \ldots, n$, individually.

Next let us consider an inductive case in which it is shown that any transition rule denoted by a CafeOBJ action operator a preserves $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. To this end, it is sufficient to show this formula

$$\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \Rightarrow \mathit{pred}(\mathrm{a}(s, y_1, \ldots, y_m), \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \qquad (6)$$

for arbitrary $s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n, y_1, \ldots, y_m$, where $y_1, \ldots, y_m$ are the arguments of the CafeOBJ action operator except for s. We may use $\boldsymbol{y}$ to denote $y_1, \ldots, y_m$.

Instead of proving (6) directly, it is sufficient to prove these n formulas

$$\begin{array}{c} \mathit{pred}_1(s, \boldsymbol{x}_1) \Rightarrow \mathit{pred}_1(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_1)\,, \\ \vdots \\ \mathit{pred}_n(s, \boldsymbol{x}_n) \Rightarrow \mathit{pred}_n(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_n) \end{array} \tag{7}$$

because (6) can be deduced from these n formulas. But some of (7), especially the first formula from the assumption, may not be proved by themselves.

Let $\mathit{pred}_i(s, \boldsymbol{x}_i) \Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)$ be one of such formulas. The reason why it is impossible to prove the formula by itself is that the inductive hypothesis $\mathit{pred}_i(s, \boldsymbol{x}_i)$ is too weak to imply $\mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)$. Generally what strengthens the inductive hypothesis can be $\mathit{pred}_{j_1^i}(s, \boldsymbol{t}_{j_1^i}) \wedge \ldots \wedge \mathit{pred}_{j_{u_i}^i}(s, \boldsymbol{t}_{j_{u_i}^i})$, where $1 \leq j_1^i, \ldots, j_{u_i}^i \leq n$, $\boldsymbol{t}_j\,(j = j_1^i, \ldots, j_{u_i}^i)$ is $t_{j1}, \ldots, t_{jm_j}$ and $t_k\,(k = j1, \ldots, jm_j)$ is a term that is x_k, another variable whose sort is the same as x_k's, or more concrete value whose sort is the same as x_k's. Let SIH_i be $\mathit{pred}_{j_1^i}(s, \boldsymbol{t}_{j_1^i}) \wedge \ldots \wedge \mathit{pred}_{j_{u_i}^i}(s, \boldsymbol{t}_{j_{u_i}^i})$. Suppose that SIH_i can be used to strengthen the inductive hypothesis $\mathit{pred}_i(s, \boldsymbol{x}_i)$ to show $\mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)$. Then the proof of the ith formula of (7) can be replaced with the proof of this formula[3]

$$(\mathit{SIH}_i \wedge \mathit{pred}_i(s, \boldsymbol{x}_i)) \Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)\,. \tag{8}$$

[3]Let us consider the case where $u_i = 1$. If invariant $\mathit{pred}_{j_1^i}(s, \boldsymbol{x}_{j_1^i})$ has been proved independent of invariant $\mathit{pred}_i(s, \boldsymbol{x}_i)$, the proof can also be replaced with the proof of this formula

$$(\mathit{pred}_{j_1^i}(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{t}_{j_1^i}) \wedge \mathit{pred}_i(s, \boldsymbol{x}_i)) \Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)$$

because $\mathit{pred}_{j_1^i}(s, \boldsymbol{x}_{j_1^i})$ holds for any reachable state s.

Then, we generally prove these n formulas

$$\begin{aligned}(\mathit{SIH}_1 \land \mathit{pred}_1(s, \boldsymbol{x}_1)) &\Rightarrow \mathit{pred}_1(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_1)\,,\\ &\vdots\\ (\mathit{SIH}_n \land \mathit{pred}_n(s, \boldsymbol{x}_n)) &\Rightarrow \mathit{pred}_n(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_n)\,,\end{aligned} \tag{9}$$

to show that any transition rule denoted by the CafeOBJ action operator a preserves $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$. From these n formulas, we can deduce the formula

$$(\mathit{SIH}_1 \land \ldots \land \mathit{SIH}_n) \land \mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n) \Rightarrow \mathit{pred}(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)\,.$$

$\mathit{SIH}_1 \land \ldots \land \mathit{SIH}_n$ can be used as the inductive hypothesis to show that any transition rule denoted by the CafeOBJ action operator a preserves $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ because $\boldsymbol{x}_1, \ldots, \boldsymbol{x}_n$ are just instantiated in $\mathit{SIH}_1 \land \ldots \land \mathit{SIH}_n$. Therefore, the proof of (9) makes it possible to show what we would like to prove.

In order to prove (8), we may have to do case analysis, namely splitting the state space, or the case into multiple subcases. Suppose that the case is split into l subcases denoted by case^i_1, …, case^i_l that should satisfy this condition

$$(\mathit{case}^i_1 \lor \ldots \lor \mathit{case}^i_l) = \mathrm{true}\,.$$

Then the proof of (8) can be replaced with the proofs of these l formulas

$$\begin{aligned}(\mathit{case}^i_1 \land \mathit{SIH}_i \land \mathit{pred}_i(s, \boldsymbol{x}_i)) &\Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)\,,\\ &\vdots\\ (\mathit{case}^i_l \land \mathit{SIH}_i \land \mathit{pred}_i(s, \boldsymbol{x}_i)) &\Rightarrow \mathit{pred}_i(\mathrm{a}(s, \boldsymbol{y}), \boldsymbol{x}_i)\,.\end{aligned} \tag{10}$$

SIH_i may not be needed for some subcases.

From what has been discussed, it follows that each of the n predicates can be proved invariant to a system individually and compositionally even if the proofs mutually depend on each other, e.g. the $i$the predicate is used to strengthen inductive hypotheses to prove the jth predicate invariant to the system and vice versa. The original predicate $pred_1(s, \boldsymbol{x}_1)$, which we would like to prove invariant to the system, may be divided into multiple predicates.

5 Proof Scores of Invariants

Proof scores in the OTS/CafeOBJ method are based on (10), and therefore, we can write each of them individually and compositionally.

Let us consider writing proof scores of the n invariants discussed in the previous section. We first write a module, say `INV`, in which each $pred_i(s, \boldsymbol{x}_i)$, where $i = 1, \ldots, n$, is expressed as a CafeOBJ term as follows:

$$\begin{array}{l}
\texttt{op}\ inv_1\ \texttt{:}\ H\ V_{11}\ \ldots\ V_{1m_1}\ \texttt{->}\ \texttt{Bool} \\
\qquad\qquad\qquad\qquad\vdots \\
\texttt{op}\ inv_n\ \texttt{:}\ H\ V_{n1}\ \ldots\ V_{nm_n}\ \texttt{->}\ \texttt{Bool} \\
\texttt{eq}\ inv_1(S, X_{11}, \ldots, X_{1m_1})\ \texttt{=}\ pred_1(S, X_{11}, \ldots, X_{1m_1})\ \texttt{.} \\
\qquad\qquad\qquad\qquad\vdots \\
\texttt{eq}\ inv_n(S, X_{n1}, \ldots, X_{nm_n})\ \texttt{=}\ pred_n(S, X_{n1}, \ldots, X_{nm_n})\ \texttt{.}
\end{array}$$

where each V_k is a visible sort corresponding to x_k and each X_k is a CafeOBJ variable for V_k where $k = 11, \ldots,\ 1m_1, \ldots, n1, \ldots, nm_n$, S is a CafeOBJ variable for H and each $pred_i(S, X_{i1}, \ldots, X_{im_i})$ denotes $pred_i(s, \boldsymbol{x}_i)$ where $i = 1, \ldots, n$.

In the module, we also declare a constant x_k for each V_k, where $k = 11, \ldots, 1m_1, \ldots, n1, \ldots, nm_n$. In proof scores, a constant that is not constrained is used for denoting an arbitrary value for the indented sort. For example, if we declare a constant x for `Nat` that is the visible sort for natural numbers, x can be used to denote an arbitrary natural number. Such constants are constrained with equations, which makes it possible to split the state space, or the case. Suppose that the case is split into two: one where x equals 0 and the other where x does not, namely that x is greater than 0. The former is expressed by declaring the equation

$$\texttt{eq}\ x\ \texttt{=}\ 0\ \ \texttt{.}$$

The latter is expressed by declaring the equation

$$\texttt{eq}\ x\ \texttt{=}\ \texttt{succ}(n)\ \ \texttt{.} \tag{11}$$

where `succ` is the successor function of natural numbers and n is a constant that denotes an arbitrary natural numbers. Therefore, $\texttt{succ}(n)$ denotes an arbitrary positive integer.

We are going to mainly describe the proof score of the ith invariant. Let *init* be a constant that denote any initial state of the system under consideration. In order to show that $pred_i(s, \boldsymbol{x}_i)$ holds in any initial state, we write the CafeOBJ code that looks like

```
open INV
 red inv_i(init, x_i1, ..., x_im_i) .
close
```

and execute it with the CafeOBJ system. We may have to split the case in order to prove $pred_i(init, \boldsymbol{x}_i)$.

We next write a module, say `ISTEP`, in which two constants s, s' are declared, denoting an arbitrary state and the successor state after applying a transition rule in the state, and the formulas to prove in each inductive case are expressed as CafeOBJ terms as follows:

```
op $istep_1$ : $V_{11}$ ... $V_{1m_1}$ -> Bool
                                          ⋮
op $istep_n$ : $V_{n1}$ ... $V_{nm_n}$ -> Bool
eq $istep_1(X_{11}, \ldots, X_{1m_1})$
    = $inv_1(s, X_{n1}, \ldots, X_{nm_n})$ implies $inv_1(s', X_{n1}, \ldots, X_{nm_n})$ .
                                          ⋮
eq $istep_1(X_{11}, \ldots, X_{1m_1})$
    = $inv_1(s, X_{n1}, \ldots, X_{nm_n})$ implies $inv_1(s', X_{n1}, \ldots, X_{nm_n})$ .
```

These formulas correspond to (7) in the previous section.

Let us consider the inductive case in which it is shown that any transition rule denoted by the CafeOBJ action operator a preserves $pred_i(s, \boldsymbol{x}_i)$. As supposed in the previous section, SIH_i is used to strengthen the inductive hypothesis and the case is split into the l subcases denoted by $case^i_1$, ..., $case^i_l$. The CafeOBJ code to show that any transition rule denoted by a

preserves $pred_i(s, \boldsymbol{x}_i)$ in the case denoted by $case^i_j$ looks like this

```
open ISTEP
-- arbitrary objects
  op $y_1$ : -> $V_1$ .
       $\vdots$
  op $y_m$ : -> $V_m$ .
  Declare other constants if necessary.
-- assumptions
  Declare equations denoting $case^i_j$.
-- successor state
  eq $s'$ = $a(s, y_1, \ldots, y_m)$ .
-- check
  red ($pred_{j^i_1}(s, t_{j^i_1 1}, \ldots, t_{j^i_1 m_{j^i_1}})$ and ... and $pred_{j^i_u}(s, t_{j^i_u 1}, \ldots, t_{j^i_u m_{j^i_u}})$)
      implies $istep_i(x_{i1}, \ldots, x_{im_i})$ .
close
```

where each V_k is a visible sort corresponding to y_k where $k = 1, \ldots, m$ and each t_k is a term denoting t_k where $k = j^i_1 1, \ldots, j^i_1 m_{j^i_1}, \ldots, j^i_u, \ldots, j^i_u m_{j^i_u}$. For the case splitting, we may need constants such as n in (11). Such constants may also be declared in the CafeOBJ code.

6 Discussion

The compositional writing of proof scores has the following advantages:

- The CafeOBJ systems reduces a term denoting a logical formula into an exclusive-or normal form à la Hsiang[15], whose response time crucially depends on the length of the formula, essentially the possible number of `or`'s in it. The compositional writing of proof scores makes it possible to focus on each conjunct $pred_i(s, \boldsymbol{x}_i)$, where $i = 1, \ldots, n$, of a large formula $\mathit{pred}(s, \boldsymbol{x}_1, \ldots, \boldsymbol{x}_n)$ and relieve the complexity of the reduction. Let `inv5` be the CafeOBJ operator denoting the formula to prove in each inductive case corresponding to (5). We tried to reduce `inv5(i1,i2,i3,i4,i5)` with the CafeOBJ system on a laptop with 850MHz Pentium III processor and 512MB memory but did not get the result in one hour. On the other hand, it took about four seconds to load the CafeOBJ specification and the four proof scores of (1), (2), (3) and (1) with the CafeOBJ system on the same laptop.

- Since the compositional writing of proof scores makes it possible to focus on each conjunct of a large formula, case analysis can be done with respect to each conjunct only. This can ease the complexity of case analysis. Suppose that we have to consider N_i subcases for $\mathit{pred}_i(s, \boldsymbol{x}_i)$ and N_j subcases for $\mathit{pred}_j(s, \boldsymbol{x}_j)$ in an inductive case to prove them invariant to a system. If we try to prove $\mathit{pred}_i(s, \mathbf{x}_i) \wedge \mathit{pred}_j(s, \mathbf{x}_j)$ invariant to the system, then we may have to consider $N_i \times N_j$ subcases in the inductive case.

Writing proof scores in algebraic specification languages was started by Goguen and his colleagues[9, 11, 8]. If his way to write proof scores is used to prove a predicate invariant to a system and it is impossible to prove the predicate by itself invariant to the system, we first prove other predicates invariant to the system and then use them as lemmas to prove the predicate invariant to the system, or we find a large predicate that implies the predicate and prove the large predicate invariant to the system. The former is the first strategy in Subsection 3.2, and the latter is the second strategy in Subsection 3.2. The first strategy cannot be applied to the verification of the example discussed in Section 3. It also seems impossible to use the second strategy to verify the example with the CafeOBJ system on a usual personal computer.

Many theorem provers and proof assistants have been proposed. Among them are PVS[29] and Isabelle/HOL[19], which are often cited and have been used to verify various systems. If these proof assistants are used to prove predicates invariant to a system, the first and second strategies in Subsection 3.2 are used. It seems difficult to use the third strategy, namely the compositional proof of invariants to prove predicates invariant to a system with such a proof assistant.

PVS and Isabelle/HOL are based on higher-order logic and higher-order unification, while the OTS/CafeOBJ method is based on equations and equational reasoning. Equations and equational reasoning are easier to understand than higher-order logic and higher-order unification. Therefore, specifications and proofs (proof scores) in the OTS/CafeOBJ method are easier to read and understand than those in the proof assistants. On the other hand, the proof assistants can automate more verification processes than the OTS/CafeOBJ method. In the case studies performed so far, proof scores were entirely written by hand. Since proof scores in the OTS/CafeOBJ method are well stylized, most part of proof scores can be automatically generated. Therefore, we have been designing a tool, called Gateau, that is fed scripts to generate proof scores in order to automate more verification processes. A prototype has been implemented. All the Gateau scripts for the four invariants in Subsection 3.2 consist of 142 lines and the generated proof scores consist of 2,946 lines, while the proof scores written by hand consist of 878 lines.

When a predicate is proved invariant to a system with the OTS/CafeOBJ method, it is significant to split the case into multiple subcases and find other predicates that are invariant to the system and can be used to strengthen the inductive hypothesis of the original proof in inductive cases in order to make progress on the proof. The former can be done based on predicates used in specifications such as the equivalence between process IDs in the example describe in Section 3. If you encounter a subcase in which the term denoting a formula to be proved is reduced to `false`, then the subcase may denote unreachable states and you may conjecture other invariants.

7 Conclusion

We have described the foundation of the compositional writing of proof scores in the OTS/CafeOBJ method and used the queuing lock algorithm to demonstrate how to write proof scores. We have been demonstrating the usefulness of the compositional writing of proof scores by performing several case studies, especially verification of security protocols.

The security protocols verified by writing proof scores compositionally are as follows:

- The NSLPK authentication protocol[16]: We have verified that the protocol has nonce secrecy and one-to-many agreement properties[23, 27].

- iKP ($i = 1, 2, 3$) payment protocols[1]: We have found that 2KP and 3KP as well as 1KP do not have a property that seems important while verifying that 2KP and 3KP have the property and proposed a possible solution[24]. The property is that if an acquirer authorizes a payment, then both the buyer and seller concerned always agree on it. We have verified that the modified 2KP and 3KP have the property[25].

- The Horn-Preneel micropayment protocol[14]: We have verified that a payee cannot overcharge a payer[26].

- The SET payment protocol[17]: We have verified that the protocol has several properties. Among the properties are that if the payment gateway authorizes a payment, then both cardholder and merchant concerned always agree on the payment, and at this time the two principals also agree on the transaction amount.

- NetBill electronic commerce protocol[4]: We have found that the protocol does not have a property, called goods atomicity in a strict sense and proposed a possible solution. We have finished the invariant part of the verification that the modified protocol has the property and also verified that it has several other desired properties[28].

- TLS handshake protocol[7]: We have verified that the protocol have several properties. Among the properties are that pre-master secrets cannot be leaked, when a client has negotiated a cipher suite and security parameters with a server, the server has really agreed on them, and client cannot be identified if they dot not send their certificates to servers.

We also have mentioned a tool, called Gateau that help generate proof scores. We have been designing the tool and are going to implement it.

References

[1] Mihir Bellare, Juan Garay, Ralf Hauser, Amir Herzberg, Hugo Krawczyk, Michael Steiner, Gene Tsudik, Els Van Herreweghen, and Michael Waidner. Design, implementation and deployment of the iKP secure electronic payment system. *IEEE Journal of Selected Areas in Communications*, 18(4):611–627, 2000.

[2] CafeOBJ web page. `http://www.ldl.jaist.ac.jp/cafeobj/`.

[3] K. Mani Chandy and Jayadev Misra. *Parallel Program Design: A Foundation.* Addison-Wesley, Reading, MA, 1988.

[4] Benjamin Cox, J. D. Tygar, and Marvin Sirbu. NetBill security and transaction protocol. In *First USENIX Workshop on Electronic Commerce*, pages 77–88, 1995.

[5] Răzvan Diaconescu and Kokichi Futatsugi. *CafeOBJ Report*, volume 6 of *AMAST Series in Computing*. World Scientific, Singapore, 1998.

[6] Răzvan Diaconescu and Kokichi Futatsugi. Behavioural coherence in object-oriented algebraic specification. *Journal of Universal Computer Science*, 6:74–96, 2000.

[7] T. Dierks and C. Allen. The TLS protocol version 1.0. Request for Commnets: 2246, `http://www.ietf.org/rfc/rfc2246.txt`, 1999.

[8] Joseph Goguen. *Theorem Proving and Algebra*. The MIT Press, (to appear).

[9] Joseph Goguen and Grant Malcolm, editors. *Algebraic Semantics of Imperative Programs*. The MIT Press, 1996.

[10] Joseph Goguen and Grant Malcolm. A hidden agenda. *Theoretical Computer Science*, 245:55–101, 2000.

[11] Joseph Goguen and Grant Malcolm, editors. *Software Engineering with OBJ: Algebraic Specification in Action*. Kluwer Academic Publishers, 2000.

[12] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi, and Jean-Pierre Jouannaud. Introducing OBJ. In Joseph Goguen and Grant Malcolm, editors, *Software Engineering with OBJ: algebraic specification in action*, pages 3–167. Kluwer Academic Publishers, 2000.

[13] David Gries and Fred B. Schneider. *A Logical Approach to Discrete Math*. Texts and Monographs in Computer Science. Springer, NY, 1993.

[14] Gunther Horn and Bart Preneel. Authentication and payment in future mobile systems. In *5th European Symposium on Research in Computer Security (ESORICS 98)*, LNCS 1485, pages 277–293. Springer, 1998.

[15] Jieh Hsiang. *Refutational Theorem Proving Using Term Rewriting Systems*. PhD thesis, University of Illinois at Champaign-Urbana, 1981.

[16] Gavin Lowe. An attack on the Needham-Schroeder public-key authentication protocol. *Information Processing Letters*, 56:131–133, 1995.

[17] MasterCard/Visa. SET secure electronic transactions protocol – book 1: Business specifications; book 2: Technical specification; book 3: Formal protocol definition. `http://www.setco.org/set_specifications.html`, May 1997.

[18] Ataru N. Nakagawa, Toshimi Sawada, and Kokichi Futatsugi. CafeOBJ user's manual. `http://www.ldl.jaist.ac.jp/cafeobj/doc/`, 1999.

[19] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*, volume 2283 of *LNCS*. Springer, Berlin, 2002.

[20] Kazuhiro Ogata and Kokichi Futatsugi. Formal verification of the MCS list-based queuing lock. In *5th Asian Computing Science Conference (ASIAN 1999)*, LNCS 1742, pages 281–293. Springer, 1999.

[21] Kazuhiro Ogata and Kokichi Futatsugi. Formally modeling and verifying Ricart&Agrawala distributed mutual exclusion algorithm. In *2nd Asia-Pacific Conference on Quality Software (APAQS 2001)*, pages 357–366. IEEE CS Press, 2001.

[22] Kazuhiro Ogata and Kokichi Futatsugi. Formal analysis of Suzuki&Kasami distributed mutual exclusion algorithm. In *5th IFIP TC6/WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2002)*, pages 181–195. Kluwer Academic Publishers, 2002.

[23] Kazuhiro Ogata and Kokichi Futatsugi. Rewriting-based verification of authentication protocols. In *4th International Workshop on Rewriting Logic and its Applications (WRLA 2002)*, volume 71 of *ENTCS*. Elsevier Science Publishers, 2002.

[24] Kazuhiro Ogata and Kokichi Futatsugi. Flaw and modification of the *i*KP electronic payment protocols. *Information Processing Letters*, 86:57–62, 2003.

[25] Kazuhiro Ogata and Kokichi Futatsugi. Formal analysis of the *i*KP electronic payment protocols. In *International Symposium on Software Security (ISSS 2002)*, volume 2609 of *LNCS*, pages 441–460. Springer, 2003.

[26] Kazuhiro Ogata and Kokichi Futatsugi. Formal verification of the Horn-Preneel micropayment protocol. In *4th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2003)*, volume 2575 of *LNCS*, pages 238–252. Springer, 2003.

[27] Kazuhiro Ogata and Kokichi Futatsugi. Proof scores in the OTS/CafeOBJ method. In *6th IFIP WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (FMOODS 2003)*, volume 2884 of *LNCS*, pages 170–184. Springer, 2003.

[28] Kazuhiro Ogata and Kokichi Futatsugi. Formal analysis of the Net-Bill electronic commerce protocol. In *2nd International Symposium on Software Security (ISSS 2003)*, LNCS, page (to appear). Springer, 2004.

[29] Sam Owre, John Rushby, Natarajan Shankar, and Friedrich von Henke. Formal verification for fault-tolerant architectures: Prolegomena to the

design of PVS. *IEEE Transactions on Software Engineering*, 21:107–125, 1995.