PAPER

Proof Score Approach to Verification of Liveness Properties*

Kazuhiro OGATA[†a)], ***Member and*** **Kokichi FUTATSUGI**[†], ***Nonmember***

SUMMARY Proofs written in algebraic specification languages are called proof scores. The proof score approach to design verification is attractive because it provides a flexible way to prove that designs for systems satisfy properties. Thus far, however, the approach has focused on safety properties. In this paper, we describe a way to verify that designs for systems satisfy liveness properties with the approach. A mutual exclusion protocol using a queue is used as an example. We describe the design verification and explain how it is verified that the protocol satisfies the lockout freedom property.

key words: *CafeOBJ, equations, observational transition systems (OTSs), rewriting, specification*

1. Introduction

The *proof score approach to design verification* is a formal method of verifying that a design for a system satisfies a property. In the approach, a design for a system is specified in an algebraic specification language, a property is expressed in the language, and it is verified that the design satisfies the property by writing proofs (or proof plans) called *proof scores* in the language and executing the proof scores with a processor of the language.

We have argued in [2] that the the approach has several attractive characteristics thanks to (1) balanced human-computer interaction and (2) flexible but clear structure of proof scores. The former means that humans are able to focus on proof plans, while tedious and detailed computations can be left to computers; humans do not necessarily have to know what deductive rules or equations should be applied to goals to prove. The latter means that lemmas do not need to be proved in advance and proof scores can help humans comprehend the corresponding proofs; a proof that a system satisfies a property can be conducted even when all lemmas used have not been proved, and assumptions used are explicitly and clearly written in proof scores.

The *OTS/CafeOBJ method* [3]–[5] is an instance of the proof score approach to design verification. The main ingredients of the OTS/CafeOBJ method are *observational transition systems (OTSs)* and *CafeOBJ*. OTSs are a kind of transition system (or state machine), which can be used as mathematical models of designs for systems, and CafeOBJ [6] is an executable algebraic specification language and system. Given a problem that a design for a system satisfies a property, in the OTS/CafeOBJ method, (1) the design is modeled as an OTS, which is written in CafeOBJ, (2) the property is expressed as a CafeOBJ term, and (3) it is verified that the OTS satisfies the property by writing proof scores in CafeOBJ and executing them with CafeOBJ. A survey of proof scores in CafeOBJ is described in [7].

We have conducted case studies [8]–[10] to demonstrate the usefulness of the OTS/CafeOBJ method. Thus far, however, the OTS/CafeOBJ method has mainly focused on invariant properties, which are safety properties. This paper shows that the method can also deal with other kinds of properties, especially ensures and leads-to properties, which are *liveness properties*. A mutual exclusion protocol using a queue, which is called *Qlock* [5], is used as an example; it is verified that Qlock satisfies the lockout (or starvation) freedom property, which can be expressed as a leads-to property.

The OTS/CafeOBJ method has been largely influenced by UNITY [11]. Although UNITY has been widely used to design parallel and distributed systems, UNITY itself does not provide any tool support. Therefore, some formal methods and tools have been used to support UNITY [12]–[14], The OTS/CafeOBJ method may be regarded as one of such formal methods and tools. None of the existing formal methods and tools to support UNITY is based on the proof score approach to design verification. This is the essential difference between the OTS/CafeOBJ method and the others.

The rest of the paper is organized as follows. Section 2 describes OTSs including the five basic properties. Section 3 introduces CafeOBJ. Section 4 describes proof scores of ensures and leads-to properties. Section 5 reports on a case study in which it is verified that Qlock satisfies the lockout freedom property. Section 6 mentions some related work. Section 7 concludes the paper.

2. Observational Transition Systems (OTSs)

We describe the definitions of basic concepts on observational transition systems, or OTSs, and five basic properties with respect to (wrt) OTSs.

2.1 Definitions

We suppose that there exists a universal state space denoted

Manuscript received May 11, 2007.
Manuscript revised August 1, 2008.

[†]The authors are with the School of Information Science, JAIST, Nomi-shi, 923–1292 Japan.

*This paper is an extended and revised version of the paper [1] presented at 17th SEKE.

a) E-mail: ogata@jaist.ac.jp

DOI: 10.1093/ietisy/e91–d.12.2804

Υ 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}.

Definition 1 (OTSs): An OTS $\mathcal{S}$ is $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ such that

- $\mathcal{O}$: A finite set of observers. Each *observer* $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \to D_o$ 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 $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \to D_o$ 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 $[\upsilon] \in \Upsilon/=_{\mathcal{S}}$, each $\upsilon_1, \upsilon_2 \in [\upsilon]$ 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 means that if $c\text{-}t_{y_1,\ldots,y_n}(\upsilon)$ holds for a given state υ, then $t_{y_1,\ldots,y_n}$ moves υ to υ' 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 υ. 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 υ, 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 observer $o \in \mathcal{O}$ (a transition $t \in \mathcal{T}$) and values $a_k : D_{ok}, \ldots, a_m : D_{om}$ ($b_1 : D_{t1}, \ldots, b_n : D_{tn}$), $o_{a_1,\ldots,a_m}$ ($t_{b_1,\ldots,b_n}$) is called an *instance* of the observer (the transition). If an observer (a transition) does not have any indexes, the observer (the transition) itself is the instance of the observer (the transition).

Given an OTS $\mathcal{S}$ and two states $\upsilon, \upsilon' \in \Upsilon$, if there exists an instance $t_{b_1,\ldots,b_n}$ of a transition $t \in \mathcal{T}$ such that $t_{b_1,\ldots,b_n}(\upsilon) =_{\mathcal{S}} \upsilon'$, we write $\upsilon \leadsto_{\mathcal{S}}^{t_{b_1,\ldots,b_n}} \upsilon'$ and call υ' a *t-successor state* of υ wrt $\mathcal{S}$. $t_{b_1,\ldots,b_n}$ may be omitted from $\upsilon \leadsto_{\mathcal{S}}^{t_{b_1,\ldots,b_n}} \upsilon'$ and υ' may be called a successor state of υ wrt $\mathcal{S}$.

A mutual exclusion protocol called Qlock using a queue is used as an example.

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

Loop
 rs: put(*queue*, i)
 wt:**repeat until** top(*queue*) = i
  Critical Section
 cs: get(*queue*)

queue is the queue of process IDs shared by all processes. put, top and get are the usual functions of queues. put(*queue*, i) puts a process ID i into *queue* at the end, get(*queue*) deletes the top element from *queue* if *queue* is not empty, and top(*queue*) returns the top element of *queue* if *queue* is not empty. rs, wt and cs are labels given to the pseudo-code, standing for the remainder section, waiting and the critical section. Initially, each process i is at label rs and *queue* is empty. Let Label, Pid and Queue be the types of labels, process IDs and queues of process IDs, respectively.

Qlock can be modeled as the OTS $\mathcal{S}_{\text{Qlock}}$ such that

$$\begin{aligned}
\mathcal{O}_{\text{Qlock}} &\triangleq \{\text{pc}_{i:\text{Pid}} : \Upsilon \to \text{Label}, \text{queue} : \Upsilon \to \text{Queue}\} \\
\mathcal{I}_{\text{Qlock}} &\triangleq \{\upsilon \in \Upsilon \mid \forall i : \text{Pid}.(\text{pc}_i(\upsilon) = \text{rs}) \wedge \\
&\qquad \text{queue}(\upsilon) = \text{empty}\} \\
\mathcal{T}_{\text{Qlock}} &\triangleq \{\text{want}_{i:\text{Pid}} : \Upsilon \to \Upsilon, \text{try}_{i:\text{Pid}} : \Upsilon \to \Upsilon, \\
&\qquad \text{exit}_{i:\text{Pid}} : \Upsilon \to \Upsilon\}
\end{aligned}$$

where empty is the empty queue. The three transitions are defined as follows:

$\text{want}_i(\upsilon) \triangleq \upsilon'$ **if** $\text{c-want}_i(\upsilon)$ **s.t.**
 $\text{pc}_j(\upsilon') =$ **if** $i = j$ **then** wt **else** $\text{pc}_j(\upsilon)$
 $\text{queue}(\upsilon') = \text{put}(\text{queue}(\upsilon), i)$
where $\text{c-want}_i(\upsilon) \triangleq \text{pc}_i(\upsilon) = \text{rs}$

$\text{try}_i(\upsilon) \triangleq \upsilon'$ **if** $\text{c-try}_i(\upsilon)$ **s.t.**
 $\text{pc}_j(\upsilon') =$ **if** $i = j$ **then** cs **else** $\text{pc}_j(\upsilon)$
where $\text{c-try}_i(\upsilon) \triangleq \text{pc}_i(\upsilon) = \text{wt} \wedge \text{top}(\text{queue}(\upsilon)) = i$

$\text{exit}_i(\upsilon) \triangleq \upsilon'$ **if** $\text{c-exit}_i(\upsilon)$ **s.t.**
 $\text{pc}_j(\upsilon') =$ **if** $i = j$ **then** rs **else** $\text{pc}_j(\upsilon)$
 $\text{queue}(\upsilon') = \text{get}(\text{queue}(\upsilon))$
where $\text{c-exit}_i(\upsilon) \triangleq \text{pc}_i(\upsilon) = \text{cs}$

where put, top and get are the same functions appearing in the pseudo-code. □

Definition 2 (Reachable states): 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 υ is reachable wrt $\mathcal{S}$, so is υ'.

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

Example 2 (Reachable states wrt $\mathcal{S}_{\text{Qlock}}$): Let $\upsilon_0 \in \mathcal{I}_{\text{Qlock}}$. Both $\text{want}_i(\upsilon_0)$ and $\text{try}_i(\text{want}_i(\upsilon_0))$ are reachable wrt $\mathcal{S}_{\text{Qlock}}$. υ_{-1} such that $\text{pc}_i(\upsilon_{-1}) = \text{cs} \wedge \text{pc}_j(\upsilon_{-1}) = \text{cs} \wedge i \neq j$ is not reachable wrt $\mathcal{S}_{\text{Qlock}}$, although it needs to be verified. □

2.2 Properties

Predicates whose types are $\Upsilon \to$ Bool are called *state predicates*. We suppose that every state predicate considered in this paper does not have any quantifiers unless otherwise explicitly stated. Note that variables freely occurring in formulas are equivalent to universally quantified variables. There are five basic properties wrt $\mathcal{S}$, which are inspired by UNITY [11].

Definition 3 (Five Basic Properies): Let p, q, r, p_j be arbitrary state predicates, and J be an arbitrary set. Given an OTS $\mathcal{S}$, the five properties are defined:

(1) p $\mathsf{unless}_{\mathcal{S}}$ $q \triangleq \forall \upsilon, \upsilon' : \mathcal{R}_{\mathcal{S}}.$
$(\upsilon \leadsto_{\mathcal{S}} \upsilon' \wedge p(\upsilon) \wedge \neg q(\upsilon) \Rightarrow p(\upsilon') \vee q(\upsilon'))$

(2) $\mathsf{stable}_{\mathcal{S}}$ $p \triangleq p$ $\mathsf{unless}_{\mathcal{S}}$ false.

(3) $\mathsf{invariant}_{\mathcal{S}}$ $q \triangleq \forall \upsilon : \mathcal{R}_{\mathcal{S}}.\, p(\upsilon)$

(4) p $\mathsf{ensures}_{\mathcal{S}}$ $q \triangleq (p$ $\mathsf{unless}_{\mathcal{S}}$ $q) \wedge$
$\exists t : \mathcal{T}.\exists b_1 : D_{t1}.\ldots.\exists b_m : D_{tm}.\forall \upsilon, \upsilon' : \mathcal{R}_{\mathcal{S}}.$
$(\upsilon \leadsto_{\mathcal{S}}^{t_{b_1,\ldots,b_m}} \upsilon' \wedge p(\upsilon) \wedge \neg q(\upsilon) \Rightarrow q(\upsilon'))$

(5) p $\mathsf{leads\text{-}to}_{\mathcal{S}}$ q (or $p \mapsto_{\mathcal{S}} q$) holds
if and only if this can be deduced by applying
the three deductive rules finitely often:

1. $\dfrac{p \ \mathsf{ensures}_{\mathcal{S}}\ q}{p \mapsto_{\mathcal{S}} q}$
2. $\dfrac{p \mapsto_{\mathcal{S}} q,\ q \mapsto_{\mathcal{S}} r}{p \mapsto_{\mathcal{S}} r}$
3. $\dfrac{\forall j : J.(p_j \mapsto_{\mathcal{S}} q)}{(\exists j : J.\, p_j) \mapsto_{\mathcal{S}} q}$

Note that $\mathsf{invariant}_{\mathcal{S}}$ q can also be defined as $\forall \upsilon : \mathcal{I}.\, p(\upsilon) \wedge$ stable p, which is equivalent to $\forall \upsilon : \mathcal{R}_{\mathcal{S}}.\, p(\upsilon)$. The second conjunct of the definition of p $\mathsf{ensures}_{\mathcal{S}}$ q may be abbreviated as p $\mathsf{eventually}_{\mathcal{S}}$ q. □

$\mathcal{S}$ may be omitted from $\mathsf{unless}_{\mathcal{S}}$, $\mathsf{stable}_{\mathcal{S}}$, $\mathsf{invariant}_{\mathcal{S}}$, $\mathsf{ensures}_{\mathcal{S}}$ and $\mathsf{leads\text{-}to}_{\mathcal{S}}$ ($\mapsto_{\mathcal{S}}$) if it is clear from context. The first three properties are classified into safety properties, while the remaining are classified into liveness properties.

We informally describe what the five properties say. p $\mathsf{unless}_{\mathcal{S}}$ q says that whenever each instance of every transition is applied in a state where p holds and q does not hold, p or q holds in the successor state. It does not mention what occurs when an instance of a transition is applied in a state where p does not hold or q holds. It can be interpreted as follows: once p holds, p keeps holding unless q becomes true. $\mathsf{stable}_{\mathcal{S}}$ p says that once p holds, p keeps holding, although p may never get true. $\mathsf{invariant}_{\mathcal{S}}$ p says that p holds in every reachable state wrt $\mathcal{S}$.

Before describing what the remaining two properties say, we define executions of an OTS $\mathcal{S}$. An arbitrary infinite sequence $\upsilon_0, \upsilon_1, \ldots$ of states satisfying the following three conditions is called an *execution* of $\mathcal{S}$:

- *Initiation*: $\upsilon_0 \in \mathcal{I}$.
- *Consecution*: For each $i \in \{0, 1, \ldots\}$, $\upsilon_i \leadsto_{\mathcal{S}} \upsilon_{i+1}$.
- *Fairness*: For each instance $t_{y_1,\ldots,y_n}$ of every transition, there exist an infinite number of indexes i such that $\upsilon_i \leadsto_{\mathcal{S}}^{t_{y_1,\ldots,y_n}} \upsilon_{i+1}$.

The first and second conditions guarantee that every state appearing in executions of $\mathcal{S}$ is reachable wrt $\mathcal{S}$. The third condition says that each instance $t_{y_1,\ldots,y_n}$ of every transition is applied infinitely many times in the course of every execution of $\mathcal{S}$, although the effective condition of $t_{y_1,\ldots,y_n}$ may not hold in states in which $t_{y_1,\ldots,y_n}$ is applied.

p $\mathsf{ensures}_{\mathcal{S}}$ q says that whenever there exists a state υ_i in an arbitrary execution of $\mathcal{S}$ such that p holds in υ_i, there exists a state υ_j in the execution such that $j \geq i$ and q holds in υ_j. This is because p $\mathsf{ensures}_{\mathcal{S}}$ q specifies that p or q holds in the successor state after applying each instance of every transition in a state where p holds and q does not hold and that there exists an instance $t_{b_1,\ldots,b_n}$ of a transition that makes q true when $t_{b_1,\ldots,b_n}$ is applied in a state where p holds and q does not hold, and such an instance of a transition is eventually applied thanks to *Fairness*. p $\mathsf{ensures}_{\mathcal{S}}$ q can be interpreted as follows: whenever $\mathcal{S}$ reaches a state where p holds, $\mathcal{S}$ will eventually reach a state where q holds. Although $p \mapsto_{\mathcal{S}} q$ resembles p $\mathsf{ensures}_{\mathcal{S}}$ q, p does not necessarily keep holding until q gets true in $p \mapsto_{\mathcal{S}} q$. $\mathsf{leads\text{-}to}$ properties are transitive from the definition, but $\mathsf{ensures}$ properties are not.

The role of *Fairness* is to make it possible to interpret p $\mathsf{ensures}_{\mathcal{S}}$ q as described in the previous paragraph. If we do not assume *Fairness*, p $\mathsf{ensures}_{\mathcal{S}}$ q does not necessarily mean that whenever $\mathcal{S}$ reaches a state where p holds, $\mathcal{S}$ will eventually reach a state where q holds because some transitions may be never applied from some time on. When $\mathsf{ensures}$ and $\mathsf{leads\text{-}to}$ properties are verified, we do not need to explicitly care about *Fairness* because the definitions of $\mathsf{ensures}$ and $\mathsf{leads\text{-}to}$ properties do not refer to *Fairness*.

Some readers may wonder if the three deductive rules for $\mathsf{leads\text{-}to}$ properties are sound and complete. Let $\mathsf{LT}(p, q)$ be the property that whenever $\mathcal{S}$ reaches a state where p holds, $\mathcal{S}$ will eventually reach a state where q holds. Whenever $p \mapsto_{\mathcal{S}} q$ is deduced by applying the rules finitely often, the rules are called sound if $\mathsf{LT}(p, q)$ holds. Whenever $\mathsf{LT}(p, q)$ holds, the rules are called complete if $p \mapsto_{\mathcal{S}} q$ is deduced by applying the rules finitely often. When we argue the soundness, all we have to do is to check if each rule is sound. The first rule is sound because p $\mathsf{ensures}_{\mathcal{S}}$ q can be interpreted as $\mathsf{LT}(p, q)$ as described above. Since it is natural that LT is transitive, namely that $\mathsf{LT}(p, r)$ comes from $\mathsf{LT}(p, q)$ and $\mathsf{LT}(q, r)$, the second rule is sound. It is also natural that if $\mathsf{LT}(p_j, q)$ holds for all $j \in J$, then $\mathsf{LT}((\exists j : J.\, p_i), q)$ holds because $\exists j : J.\, p_i$ implies that some p_k holds. Therefore, the third rule is sound. For the completeness, we refer to the article [15].

Example 3 (Properties of $\mathcal{S}_{\text{Qlock}}$): Let $p(\upsilon, i)$, $q(\upsilon, i)$ and $r(\upsilon, i)$ be $\text{pc}_i(\upsilon) = \text{cs}$, $\text{pc}_i(\upsilon) = \text{wt}$ and $\text{top}(\text{queue}(\upsilon)) = i$, respectively. Some properties of $\mathcal{S}_{\text{Qlock}}$ are as follows:

1. $(q(\upsilon, i) \wedge r(\upsilon, i))$ unless $p(\upsilon, i)$

2. stable $((p(v, i) \wedge p(v, j)) \Rightarrow (i = j))$
3. invariant $((p(v, i) \wedge p(v, j)) \Rightarrow (i = j))$
4. $(q(v, i) \wedge r(v, i))$ ensures $p(v, i)$
5. $q(v, i) \mapsto p(v, i)$.

The third property is called the *mutual exclusion property* and the fifth property is called the *lockout (or starvation) freedom property*. The five properties need to be verified. We will describe the verification of the fifth property in this paper. The verification needs eight invariant properties, three ensures properties and four leads-to properties as lemmas. Among the lemmas are the third and fourth properties. □

3. CafeOBJ

CafeOBJ [6] is an algebraic specification language and system mainly based on order-sorted algebras and hidden algebras [16], [17]. 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 (¬), conjunction (∧), disjunction (∨), exclusive disjunction (xor), implication (⇒) and logical equivalence (⇔), respectively. An underscore `_` indicates the place where an argument is put such as `a and b`. The operator `if_then_else_fi` corresponding to the **if** construct in programming languages is also declared. CafeOBJ uses the Hsiang term rewriting system [18] 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.

In the rest of this section, we describe how to specify data types and OTSs, and how to prove that data types satisfy properties by giving some examples, which will be used later.

3.1 Specification in CafeOBJ

We first specify Label, Pid, Nat (that is the type of natural numbers) and Queues. Label is specified in the module `LABEL`:

```
mod! LABEL { [Label]
  ops rs wt cs : -> Label
  op _=_ : Label Label -> Bool {comm}
  var L : Label
  eq (L = L) = true .
  eq (rs = wt) = false .
  eq (rs = cs) = false .
  eq (wt = cs) = false .
}
```

The keyword `mod!` indicates that the module is a tight semantics declaration, meaning the smallest model (implementation) that respect all requirements written in the module. Visible sorts are declared by enclosing them with `[` and `]`. `Label` is the visible sort of labels. The keyword `op` is used to declare (non-observation and action) operators, and `ops` to declare more than one such operator simultaneously. The operator `_=_` checks if two labels are equal. The keyword `comm` specifies that the operator `_=_` is commutative, namely that `l1 = l2` equals `l2 = l1`. The keyword `var` is used to declare variables, and `vars` to declare more than one variable simultaneously. `L` is a variable of `Label`. The keyword `eq` is used to declare equations, and `ceq` to declare conditional equations in which conditions are written after the keyword `if`. Equations and conditional equations are used to define operators and specify properties of operators. CafeOBJ uses declared equations as left-to-right rewrite rules to reduce terms.

Pid is specified in the module `PID`:

```
mod* PID { [Pid]
  op _=_ : Pid Pid -> Bool {comm}
  var I : Pid
  eq (I = I) = true .
}
```

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.

Nat is specified in the module `PNAT`:

```
mod! PNAT { [Nat]
  op 0 : -> Nat  op s : Nat -> Nat
  op _<_ : Nat Nat -> Bool
  op _=_ : Nat Nat -> Bool {comm}
  vars X Y : Nat
```

```
  eq 0 < 0 = false .  eq 0 < s(X) = true .
  eq s(X) < s(Y) = X < Y .
  eq (X = X) = true .
  eq (s(X) = 0) = false .
  eq (s(X) = s(Y)) = (X = Y) .
}
```

The constant `0` denotes zero and the operator `s` is the successor function of natural numbers. The operator `_<_` is the less-than predicate of natural numbers.

We first specify generic queues in the module `QUEUE`:

```
mod! QUEUE(D :: EQTRIV) { pr(PNAT) [Queue]
  op empty : -> Queue
  op _,_  : Elt.D Queue -> Queue
  op put : Queue Elt.D -> Queue
  op get : Queue -> Queue
  op top : Queue -> Elt.D
  op _\in_ : Elt.D Queue -> Bool
  op del : Queue Elt.D -> Queue
  op where : Queue Elt.D -> Nat
  op aux-where : Queue Elt.D -> Nat
  op _=_ : Queue Queue -> Bool {comm}
  vars Q R : Queue  vars X Y : Elt.D
  var N : Nat
  eq put(empty,X) = X,empty .
  eq put((Y,Q),X) = Y,put(Q,X) .
  eq get(empty) = empty . eq get((X,Q)) = Q .
  eq top((X,Q)) = X .
  eq X \in empty = false .
  eq X \in (Y,Q) = (X = Y) or X \in Q .
  eq del(empty,Y) = empty .
  eq del((X,Q),Y)
     = if X = Y then Q else X,del(Q,Y) fi .
  eq where((X,Q),Y)
     = if Y \in (X,Q) then aux-where((X,Q),Y)
                     else where(empty,Y) fi .
  eq aux-where((X,Q),Y) = if X = Y
      then 0 else s(aux-where(Q,Y)) fi .
  eq (Q = Q) = true .
  eq (X,Q = empty) = false .
  eq (X,Q = Y,R) = (X = Y and Q = R) .
}
```

The keyword `pr` is used to import modules. `QUEUE` imports `PNAT`. The constant `empty` denotes the empty queue and the operator `_,_` is the constructor of non-empty queues. The operators `put`, `get` and `top` are usual functions of queues, and the operator `_\in_` is the membership predicate of queues. The operator `del` deletes a given element from a given queue if any. The operator `where` returns the nearest position from top where a given element appears in a given queue if any.

`QUEUE` has one formal parameter `D`. Given a module that respects all the requirements in the module `EQTRIV` as an actual parameter, `QUEUE` is instantiated. `EQTRIV` is as follows:

```
mod* EQTRIV { [Elt]
  op _=_ : Elt Elt -> Bool {comm}
  var X : Elt
  eq (X = X) = true .
}
```

When `QUEUE` is instantiated with an actual parameter M, visible sort `Elt.D` is replaced with the visible sort in M corresponding to the visible sort `Elt` in `EQTRIV`. For example, `QUEUE(PID)` is the module obtained by instantiating `QUEUE` with `PID`, specifying Queue.

Now that we have specified the data types used in $\mathcal{S}_{\mathrm{Qlock}}$, we next specify $\mathcal{S}_{\mathrm{Qlock}}$ in the module `QLOCK`:

```
mod* QLOCK { pr(LABEL) pr(PID) pr(QUEUE(PID))
  *[Sys]*
-- an arbitrary initial state
  op init : -> Sys
-- observers
  bop pc    : Sys Pid -> Label
  bop queue : Sys -> Queue
-- actions
  bops want try exit : Sys Pid -> Sys
-- variables
  var S : Sys  vars I J : Pid
-- equations defining init
  eq pc(init,I)  = rs .
  eq queue(init) = empty .
-- equations defining want
  op c-want : Sys Pid -> Bool
  eq c-want(S,I) = (pc(S,I) = rs) .
  ceq pc(want(S,I),J)
      = (if I = J then wt 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) .
-- equations defining try
  op c-try : Sys Pid -> Bool
  eq c-try(S,I)
     = (pc(S,I) = wt 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) .
-- equations defining try
  ceq try(S,I) = S if not c-try(S,I) .
  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 rs 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) .
}
```

A comment starts with `--` and terminates at the end of the line. Hidden sorts are declared by enclosing them with `*[`

and]*. Sys is the hidden sort denoting Υ. The keyword bop is used to declare observation and action operators, and bops to declare more than one such operator simultaneously. The hidden constant init denotes an arbitrary initial state and the two observation and three action operators correspond to the two observers and three transitions of $\mathcal{S}_{\text{Qlock}}$, respectively. The operator c-want, c-try and c-exit denote the effective conditions c-want, c-try and c-exit, respectively. The module has the four sets of equations that define init, want, try and exit.

3.2 Verification with CafeOBJ

In this paper, we will discuss the verification that Qlock satisfies the lockout freedom property. The verification needs two lemmas on natural numbers and 10 lemmas on queues (see Appendix A). We describe the proofs of two of the 12 lemmas. The two lemmas are as follows:

```
eq nat-lemma2(X) = not(X = s(X)) .
eq queue-lemma3(Q,X,Y) = (X \in put(Q,Y)
                  iff (X = Y or X \in Q)) .
```

where X in nat-lemma2 is a CafeOBJ variable of Nat, and X, Y and Q in queue-lemma3 are CafeOBJ variables of Elt.D, Elt.D and Queue, respectively. The first equation is declared in the module NAT, and the second in the module Queue. The first lemma is proved by induction on X, and the second by induction on Q.

The proof (written in CafeOBJ) of the first lemma is as follows:

```
open PNAT
  op x : -> Nat .  eq x = 0 .
  red nat-lemma2(x) .
close

open PNAT
  ops x y : -> Nat .  eq x = s(y) .
  red nat-lemma2(y) implies nat-lemma2(x) .
close
```

The command open makes a temporary module that imports a given module and the command close destroys such a temporary module. The constants x and y denote arbitrary natural numbers. The term nat-lemma2(y) is the induction hypothesis. Such a proof written in CafeOBJ is called a *proof score*. Fragments enclosed with open and close in proof scores are called *proof passages*. The above proof score consists of two proof passages.

The two proof passages are of the base case and the induction case, respectively. What to prove in the base case is nat-lemma2(0), and what to prove in the induction case is nat-lemma2(s(y)) under the induction hypothesis nat-lemma2(y), where y denotes an arbitrary natural number. The proofs can be conducted by having CafeOBJ execute the proof passages. When CafeOBJ returns true for both the proof passages, the proofs are successful. CafeOBJ indeed returns true for the proof passages.

If CafeOBJ does not return true for some proof passage, you need to split the corresponding case into multiple sub-cases and/or to use some appropriate lemmas [3]. Note that although proof scores could be generated automatically [19], [20], human users basically need to write proof scores in the OTS/CafeOBJ method, which implies that human users are responsible for checking if all cases are covered.

The proof score of the second lemma is as follows:

```
open QUEUE
  op q : -> Queue .  ops x y : -> Elt.D .
  eq q = empty .
  red queue-lemma3(q,x,y) .
close

open QUEUE
  ops q q' : -> Queue .
  ops x y z : -> Elt.D .
  eq q = z,q' .  eq x = z .
  red queue-lemma3(q,x,y) .
close

open QUEUE
  ops q q' : -> Queue .
  ops x y z : -> Elt.D .
  eq q = z,q' .  eq (x = z) = false .
  red queue-lemma3(q',x,y)
      implies queue-lemma3(q,x,y) .
close
```

The first proof passage is of the base case, and the second and third proof passages are of the induction case. The induction case is split into the two sub-cases based on the proposition x = z. In the sub-case corresponding to the second proof passage, the proposition holds. In the sub-case corresponding to the third proof passage, the proposition does not. queue-lemma3(q',x,y) is the induction hypothesis (precisely an instance of the induction hypothesis), which is used in the second sub-case (i.e. the third proof passage). CafeOBJ returns true for each of the three proof passages, which means that the second lemma is successfully proved.

Proof scores of invariant properties [3]–[5] are very similar to ones of lemmas (and theorems) on data types.

4. Proof Scores of Liveness Properties

We have described in [3]–[5] how to verify that an OTS $\mathcal{S}$ satisfies invariant properties based on proof scores. In this section, we describe how to verify that $\mathcal{S}$ satisfies ensures and leads-to properties based on proof scores. Proofs of ensures properties consist of ones of unless properties and stable properties are specialized unless properties. Therefore, proof scores described in this section also covers unless and stable properties. We suppose that $\mathcal{S}$ is specified in a module SYSTEM in which a hidden sort H is declared and invariant properties wrt $\mathcal{S}$ are written in a module INV.

4.1 Proof Scores of ensures Properties

The proof of p ensures$_{\mathcal{S}}$ q consists of those of p unless$_{\mathcal{S}}$ q and p eventually$_{\mathcal{S}}$ q. The former is called the unless case, and the latter the eventually case. We suppose that in addition to υ whose type is $\mathcal{R}_{\mathcal{S}}$, p and q have the free variables $z_1, \ldots, z_\alpha$ whose types are $D_1, \ldots, D_\alpha$.

We first describe the unless case. We declare the operators denoting p and q, and their defining equations in a module `UNL` (which imports `SYSTEM` and `INV`) as follows:

op $\mathtt{unl}_p$: H $\overrightarrow{\mathtt{V}}_\alpha$ -> Bool
op $\mathtt{unl}_q$: H $\overrightarrow{\mathtt{V}}_\alpha$ -> Bool
eq $\mathtt{unl}_p(\mathtt{S},\overrightarrow{\mathtt{Z}}_\alpha)$ = $\mathtt{p}(\mathtt{S},\overrightarrow{\mathtt{Z}}_\alpha)$.
eq $\mathtt{unl}_q(\mathtt{S},\overrightarrow{\mathtt{Z}}_\alpha)$ = $\mathtt{q}(\mathtt{S},\overrightarrow{\mathtt{Z}}_\alpha)$.

$\overrightarrow{\mathtt{V}}_\alpha$ is an abbreviation of $\mathtt{V}_1 \ldots \mathtt{V}_\alpha$ and $\overrightarrow{\mathtt{Z}}_\alpha$ is an abbreviation of $\mathtt{Z}_1, \ldots, \mathtt{Z}_\alpha$. Each $\mathtt{V}_k$ is a visible sort corresponding to D_k and each $\mathtt{Z}_k$ is a CafeOBJ variable of $\mathtt{V}_k$. $\mathtt{p}(\mathtt{S},\overrightarrow{\mathtt{Z}}_\alpha)$ and $\mathtt{q}(\mathtt{S},\overrightarrow{\mathtt{Z}}_\alpha)$ are CafeOBJ terms denoting p and q. We also declare a constant $\mathtt{z}_k$ denoting an arbitrary value of $\mathtt{V}_k$ for $k = 1, \ldots, \alpha$ in `UNL`. Let $\overrightarrow{\mathtt{z}}_\alpha$ be an abbreviation of $\mathtt{z}_1, \ldots, \mathtt{z}_\alpha$.

The basic formula to prove in the unless case is denoted by the operator, which is declared and defined in a module `USTEP` (which imports `UNL`) as follows:

op ustep : $\overrightarrow{\mathtt{V}}_\alpha$ -> Bool
eq $\mathtt{ustep}(\overrightarrow{\mathtt{Z}}_\alpha)$
  = ($\mathtt{unl}_p(\mathtt{s},\overrightarrow{\mathtt{Z}}_\alpha)$ and not($\mathtt{unl}_q(\mathtt{s},\overrightarrow{\mathtt{Z}}_\alpha)$))
   implies
   ($\mathtt{unl}_p(\mathtt{s'},\overrightarrow{\mathtt{Z}}_\alpha)$ or $\mathtt{unl}_q(\mathtt{s'},\overrightarrow{\mathtt{Z}}_\alpha)$) .

`s` and `s'` are constants of `H`. `s` denotes an arbitrary state and `s'` denotes an arbitrary successor state of `s`. The constants are declared in `USTEP`.

All needed is to prove $\mathtt{ustep}(\overrightarrow{\mathtt{z}}_\alpha)$ for each instance of every transition (every action operator). We often need case splitting and lemmas (which are invariant properties wrt $\mathcal{S}$ and/or lemmas on data types). Let us consider proving $\mathtt{ustep}(\overrightarrow{\mathtt{z}}_\alpha)$ for an arbitrary instance $t_{y_1,\ldots,y_n}$ of a transition t, which is denoted by a CafeOBJ action operator `t`. We suppose that the case is split into L sub-cases characterized by L propositions case$_1$, …, case$_L$ such that case$_1 \vee \ldots \vee$ case$_L \Leftrightarrow$ true. Then the proof score of each sub-case l looks like:

open USTEP
-- arbitrary objects
op $\mathtt{y}_1$: -> $\mathtt{V}_{t1}$. $\cdots$ op $\mathtt{y}_n$: -> $\mathtt{V}_{tn}$.
-- assumptions
Declarations of equations denoting case$_l$.
-- successor state
eq s' = $\mathtt{t}(\mathtt{s},\overrightarrow{\mathtt{y}}_n)$.
-- check
red *Lems* implies $\mathtt{ustep}(\overrightarrow{\mathtt{z}}_\alpha)$.
close

Each $\mathtt{V}_{tk}$ is a visible sort corresponding to D_{tk} for $k = 1, \ldots, n$. Each constant $\mathtt{y}_k$ denotes an arbitrary value of $\mathtt{V}_{tk}$ for $k = 1, \ldots, n$. $\overrightarrow{\mathtt{y}}_n$ is an abbreviation of $\mathtt{y}_1, \ldots, \mathtt{y}_n$. A set of equations is used to denote case$_l$. We may declare other constants (which denote arbitrary values) used in equations denoting case$_l$. The constant `s'` is defined as $\mathtt{t}(\mathtt{s},\overrightarrow{\mathtt{y}}_n)$, which denotes an arbitrary t-successor state of `s` wrt $\mathcal{S}$.

Lems is a CafeOBJ term, which can be constructed by combining instances of invariant properties wrt $\mathcal{S}$ and/or lemmas on data types with conjunctions. *Lems* is used to exclude unreachable states from cases to consider. "*Lems* `implies`" may be omitted.

The variables $z_1, \ldots, z_\alpha$ occur freely in p and q. Free variables are equivalent to universally quantified ones. Therefore, if some instance of $\mathtt{unl}_p(\mathtt{s},\overrightarrow{\mathtt{Z}}_\alpha)$ and $\mathtt{not}(\mathtt{unl}_q(\mathtt{s},\overrightarrow{\mathtt{Z}}_\alpha))$ reduces to `false` in the proof passage of the sub-case case$_l$, the proof of the sub-case is discharged. Hence, *Lems* may include such an instance.

When CafeOBJ returns `true` for the proof passage of each sub-case l, the proof of the unless case is successfully completed.

We next describe the eventually case. The basic formula to prove in the eventually case is denoted by the operator, which is declared and defined in a module `ESTEP` (which imports `UNL`) as follows:

op estep : $\overrightarrow{\mathtt{V}}_\alpha$ -> Bool
eq $\mathtt{estep}(\overrightarrow{\mathtt{Z}}_\alpha)$
  = ($\mathtt{unl}_p(\mathtt{s},\overrightarrow{\mathtt{Z}}_\alpha)$ and not($\mathtt{unl}_q(\mathtt{s},\overrightarrow{\mathtt{Z}}_\alpha)$))
   implies $\mathtt{unl}_q(\mathtt{s'},\overrightarrow{\mathtt{Z}}_\alpha)$.

All needed is to prove that there exists a witness, namely an instance of a transition that makes $\mathtt{estep}(\overrightarrow{\mathtt{z}}_\alpha)$ true. We conjecture that $t_{b_1,\ldots,b_n}$ is such an instance of a transition. As the unless case, we often need case splitting and lemmas. We suppose that the case is split into L sub-cases characterized by L propositions case$_1$, …, case$_L$ such that case$_1 \vee \ldots \vee$ case$_L \Leftrightarrow$ true. Then the proof passage for each sub-case l looks like:

open ESTEP
 -- arbitrary objects
 Declarations of constants if necessary.
 -- assumptions
 Declarations of equations denoting case$_l$.
 -- successor state
 eq s' = $\mathtt{t}(\mathtt{s},\overrightarrow{\mathtt{b}}_n)$.
 -- check
 red *Lems* implies $\mathtt{estep}(\overrightarrow{\mathtt{z}}_\alpha)$.
close

$\overrightarrow{\mathtt{b}}_n$ is an abbreviation of $\mathtt{b}_1, \ldots, \mathtt{b}_n$. Each $\mathtt{b}_k$ denotes b_k for $k = 1, \ldots, n$. We may declare constants used in equations denoting case$_l$. As the unless case, *Lem* may contain an instance of $\mathtt{unl}_p(\mathtt{s},\vec{\mathtt{Z}}_\alpha)$ and $\mathtt{not}(\mathtt{unl}_q(\mathtt{s},\vec{\mathtt{Z}}_\alpha))$.

When CafeOBJ returns `true` for the proof passage of each sub-case l, the proof of the eventually case is successfully completed.

4.2 Proof Scores of leads-to Properties

Rewriting is used to verify that $\mathcal{S}$ satisfies leads-to properties based on deductive rules of leads-to. To this end, deductive rules of leads-to are written in terms of equations. When such deductive rules are written, however, we do not use the logical operators such as `_and_` and `_or_` declared and defined in the module `BOOL` and its parent modules. Basically, the left-hand side of an equation should be in normal (irreducible) form so that CafeOBJ can use the equation effectively as a rewrite rule. The normal form of a term such as `p or q` made of the logical operators is an exclusively disjunctive normal form such as `(q and p) xor q xor p`. Exclusive disjunctive normal forms are hard to read. It is inconvenient to use such hard-to-read terms as the left-hand sides of equations.

Therefore, we declare new operators denoting basic logical connectives in a module `OTSLOGIC` as follows:

```
op ~_    : Bool -> Bool {prec: 53}
op _/\_  : Bool Bool -> Bool
              {comm assoc prec: 55 r-assoc}
op _\/_  : Bool Bool -> Bool
              {comm assoc prec: 59 r-assoc}
op _=>_  : Bool Bool -> Bool
              {prec: 61 r-assoc}
op _<=>_ : Bool Bool -> Bool
              {comm prec: 63 r-assoc}
```

The operators denote negation ($\neg$), conjunction ($\wedge$), disjunction ($\vee$), implication ($\Rightarrow$) and logical equivalence ($\Leftrightarrow$), respectively. As `comm`, `assoc`, `r-assoc` and `prec:` are attributes given to operators. `assoc` specifies that an operator `_@_` is associative, namely that `(a @ b) @ c` equals `a @ (b @ c)`, `r-assoc` specifies that an operator `_@_` is right associative, namely that `a @ b @ c` is parsed as `a @ (b @ c)`, and `prec:` specifies the precedence of an operator. A natural number is written after `prec:`. The greater the number, the weaker the precedence.

`OTSLOGIC` has an operator `eval` to evaluate terms made of these operators. The operator is declared and defined as follows:

```
op eval : Bool -> Bool
eq eval(true)    = true .
eq eval(false)   = false .
eq eval(~ P)     = not eval(P) .
eq eval(P /\ Q)  = eval(P) and eval(Q) .
eq eval(P \/ Q)  = eval(P) or eval(Q) .
eq eval(P => Q)  = eval(P) implies eval(Q) .
eq eval(P <=> Q) = eval(P) iff eval(Q) .
```

where `P` and `Q` are CafeOBJ variables of `Bool`. The variables are also used in the rest of this subsection. In addition, let `P1`, `Q1`, `R`, `R1` and `R2` be CafeOBJ variables of `Bool`.

We use invariant, ensures and leads-to properties to reason about leads-to properties. Therefore, we declare the operators denoting these three kinds of properties in `OTSLOGIC` as follows:

```
op invariant_ : Bool -> Bool
op _ensures_  : Bool Bool -> Bool
op _|-->_     : Bool Bool -> Bool
```

The remainder of `OTSLOGIC` are equations denoting conditional deductive rules of leads-to properties. We use three kinds of conditional deductive rules of leads-to properties, which are as follows:

$$\textbf{if } (p \Rightarrow p_1) \wedge (q_1 \Rightarrow q),\ \textbf{then } \frac{p_1\ \mathsf{R}_{\mathcal{S}}\ q_1}{p \mapsto_{\mathcal{S}} q}$$

$$\textbf{if } (p \Rightarrow p_1) \wedge (r \Rightarrow r_1) \wedge (q_1 \Rightarrow q),\ \textbf{then } \frac{p_1\ \mathsf{R1}_{\mathcal{S}}\ r, r_1\ \mathsf{R2}_{\mathcal{S}}\ q_1}{p \mapsto_{\mathcal{S}} q}$$

$$\textbf{if } (p \Rightarrow p_1 \vee q_1) \wedge (r_1 \Leftrightarrow r_2) \wedge (r_1 \Rightarrow r),\ \textbf{then } \frac{p_1\ \mathsf{R1}_{\mathcal{S}}\ r_1, q_1\ \mathsf{R1}_{\mathcal{S}}\ r_2}{p \mapsto_{\mathcal{S}} r}$$

The three rules are parameterized. $\mathsf{R}_{\mathcal{S}}$, $\mathsf{R1}_{\mathcal{S}}$ and $\mathsf{R2}_{\mathcal{S}}$ are parameters. The three rules can be instantiated by replacing each of $\mathsf{R}_{\mathcal{S}}$, $\mathsf{R1}_{\mathcal{S}}$ and $\mathsf{R2}_{\mathcal{S}}$ with $\Rightarrow$, $\mathsf{ensures}_{\mathcal{S}}$ and $\mapsto_{\mathcal{S}}$. Therefore, we have 21 conditional deductive rules of leads-to properties. A conditional deductive rule can be applied provided that the condition holds.

We straightforwardly prove that if $p \Rightarrow q$, then $p \mapsto_{\mathcal{S}} q$ for an arbitrary OTS $\mathcal{S}$. This fact and the three deductive rules of leads-to properties in Definition 3 are used to prove that the 21 conditional deductive rules are valid.

Each of the 21 rules is denoted by one conditional equation. In this paper, the three equations denoting three of the 21 rules are shown:

```
ceq ((P1 ensures Q1) => (P |--> Q)) = true
    if eval(P => P1) and eval(Q1 => Q) .

ceq ((P1 |--> R /\ R1 |--> Q1)
                     => (P |--> Q)) = true
    if eval(P => P1) and
       eval(R => R1) and eval(Q1 => Q) .

ceq ((P1 |--> R1 /\ Q1 |--> R2)
                     => (P |--> R)) = true
    if eval(P => (P1 \/ Q1)) and
       eval(R1 <=> R2) and eval(R1 => R) .
```

In addition to the 21 conditional equations denoting the 21 rules, one more conditional equation denoting a deductive rule of leads-to properties is declared in `OTSLOGIC`:

```
ceq (P |--> Q) = true if eval(P => Q) .
```

The deductive rule denoted by the conditional equation expresses the fact that if $p \Rightarrow q$, then $p \mapsto_{\mathcal{S}} q$ for an arbitrary OTS $\mathcal{S}$.

We use one more conditional deductive rule of `leads-to` properties. The rule is as follows:

$$\textbf{if } (p \Rightarrow p_1) \wedge (q_1 \Rightarrow q), \quad \textbf{then } \frac{(p_1 \wedge M = m) \mapsto_{\mathcal{S}} ((p_1 \wedge M \prec m) \vee q_1)}{p \mapsto_{\mathcal{S}} q}$$

where m is an arbitrary value in an arbitrary set W, M is an arbitrary function mapping Υ to W, and $\prec$ is an arbitrary well-founded relation on W. The definition of leads-to$_{\mathcal{S}}$ and the mathematical induction principle are used to prove that the rule is valid [11].

Since the rule has W as its parameter, the conditional equation denoting the rule is declared in a parameterized module `INDOFLTO`, which imports `OTSLOGIC`. An actual parameter of the module has to satisfy the requirements declared in a module `EQLTRIV`, which is as follows:

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

The visible sort `Elt` corresponds to W and the operator `_<_` corresponds to $\prec$. The formal parameter of `INDOFLTO` is declared as `(D :: EQLTRIV)`. The conditional equation denoting the rule is declared in `INDOFLTO`:

```
ceq ((P1 /\ (M = X))
     |--> ((P1 /\ (M < X)) \/ Q1))
    => (P |--> Q) = true
    if eval(P => P1) and eval(Q1 => Q) .
```

`M` and `X` are CafeOBJ variables of the visible sort `Elt.D`. When `INDOFLTO` is instantiated with a module M that satisfies `EQLTRIV`, `Elt.D`, `_=_` and `_<_` are replaced with the corresponding sort, the corresponding operator and the corresponding operator in M.

In order to reason about leads-to properties from equations denoting deductive rules of leads-to properties, we declare and define in a module `PROVED` (which imports `SYSTEM` and `OTSLOGIC`) operators denoting invariant and ensures properties. Such invariant and ensures properties need to be proved in order to complete proving leads-to properties. It is not quite necessary, however, to finish proving such invariant and ensures properties in order to start proving leads-to properties. Operators denoting such invariant and ensures properties are defined with `invariant_`, `_ensures_`, `~_`, `_/\_`, `_\/_`, `_=>_` and `_<=>_` declared in `OTSLOGIC`. We declare and define in a module `LTO` (which imports `PROVED`) operators denoting leads-to properties to prove. Such operators are defined with `_|-->_`, `~_`, `_/\_`, `_\/_`, `_=>_` and `_<=>_` declared in `OTSLOGIC`.

Let us consider proving $p \mapsto_{\mathcal{S}} q$. We suppose that in addition to υ whose type is $\mathcal{R}_{\mathcal{S}}$, p and q have the free variables $z_1, \ldots, z_\alpha$ whose types are $D_1, \ldots, D_\alpha$ as we did in the previous subsection. We also use the same abbreviations used in the previous subsection. We declare the operator denoting $p \mapsto_{\mathcal{S}} q$ and its defining equation in `LTO`:

```
op lto :  H V⃗α -> Bool
eq lto(S,Z⃗α) = p(S,Z⃗α) |--> q(S,Z⃗α) .
```

As proof scores of ensures properties, we often need case splitting and lemmas. We suppose that the case is split into L sub-cases characterized by L propositions case$_1$, ..., case$_L$ such that case$_1 \vee \ldots \vee$ case$_L \Leftrightarrow$ true. Then the proof passage of each sub-case l looks like:

```
open LTO
 pr(INDOFLTO(M))
 -- arbitrary objects
 Declarations of constants if necessary.
 -- assumptions
 Declarations of equations denoting case_l.
 -- check
 red Lems implies (Prem => lto(Z⃗α)) .
close
```

"`pr(INDOFLTO(`M`))`" may be omitted. *Prem* is a CafeOBJ term whose form is P or P `/\` Q, where P is a CafeOBJ term denoting an invariant property, an ensures property or a leads-to property, and so is Q. "*Prem* `=>`" may be omitted.

5. Verification that Qlock Satisfies the Lockout Freedom Property

We describe verification that $\mathcal{S}_{\mathrm{Qlock}}$ satisfies the lockout freedom property, namely that $(\mathrm{pc}_i(\upsilon) = \mathrm{wt}) \mapsto (\mathrm{pc}_i(\upsilon) = \mathrm{cs})$. The property is denoted by the operator `lto16`, which is defined in a module `LTO` as follows:

```
eq lto16(S,I)
   = ((pc(S,I) = wt) |--> (pc(S,I) = cs)) .
```

where `S` and `I` are CafeOBJ variables of `Sys` and `Pid`. As described in Example 3, the verification needs eight invariant properties (see Appendix B), three ensures properties and four leads-to properties. Three of the eight invariant properties are directly needed for the verification. The remaining five are needed to verify the three invariant properties and the three ensures properties. Moreover, the verification needs two lemmas on natural numbers and 10 lemmas on queues as mentioned in Sect. 3.2.

The three invariant properties and the three ensures properties are denoted by the operators `inv6`, `inv7`, `inv8`, `ens9`, `ens10` and `ens11`, which are defined in a module `PROVED` (which imports `QLOCK` and `OTSLOGIC`) as shown in Fig. 1. The four leads-to properties are denoted by the operators `lto12`, `lto13`, `lto14` and `lto15`, which are defined in the module `LTO` (which imports `PROVED`) as shown in Fig. 2. In Fig. 1 and Fig. 2, `S`, `I`, `J` and `N` are CafeOBJ variables of `Sys`, `Pid`, `Pid` and `Nat`, respectively. The constants `s`, `i` and `n` of `Sys`, `Pid` and `Nat` are declared in `LTO`.

The leads-to property denoted by `lto16` is deduced from the invariant property denoted by `inv8` and the leads-to property denoted by `lto17`. The corresponding proof score is as follows:

```
eq inv6(S,I) = invariant (pc(S,I) = wt => (pc(S,I) = wt /\ I \in queue(S))) .
eq inv7(S,I) = invariant (((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = 0)
                          => (pc(S,I) = wt /\ top(queue(S)) = I))) .
eq inv8(S,I,N) = invariant ((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = s(N))
                            =>
                            (pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = s(N) /\
                            (pc(S,top(queue(S))) = wt \/ pc(S,top(queue(S))) = cs))) .
eq ens9(S,I) = ((pc(S,I) = wt /\ top(queue(S)) = I) ensures pc(S,I) = cs) .
eq ens10(S,I,J,N) = ((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = s(N)
                                   /\ top(queue(S)) = J /\  pc(S,J) = cs)
                     ensures (pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = N)) .
eq ens11(S,I,J,N) = ((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = s(N)
                                   /\ top(queue(S)) = J /\ pc(S,J) = wt)
                     ensures (pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = s(N)
                                           /\ pc(S,top(queue(S))) = cs)) .
```

Fig. 1 Equations declared in the module PROVED.

```
eq lto12(S,I,N) = ((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = s(N)
                                                   /\  pc(S,top(queue(S))) = wt)
                   |--> (pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = N)) .
eq lto13(S,I,N) = ((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = s(N)
                    /\ (pc(S,top(queue(S))) = wt \/ pc(S,top(queue(S))) = cs))
                   |--> (pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = N)) .
eq lto14(S,I,N) = ((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) = N)
                   |--> ((pc(S,I) = wt /\ I \in queue(S) /\ where(queue(S),I) < N)
                         \/ (pc(S,I) = cs))) .
eq lto15(S,I) = ((pc(S,I) = wt /\ I \in queue(S)) |--> (pc(S,I) = cs)) .
```

Fig. 2 Equations declared in the module LTO.

```
open LTO
  red (inv6(s,i) /\ lto15(s,i))
      => lto16(s,i) .
close
```

CafeOBJ returns true for this proof score.

In the rest of the section, we describe the proof scores of the two leads-to properties denoted by lto14 and lto15, and that of the ensures property denoted by ens11. The proof scores of lto12 and lto13 are written in the same way as that of lto16, and the proof scores of ens9 and ens10 are written in the same way as that of ens11.

5.1 Proof Score of lto14

The leads-to property denoted by lto14 is deduced from the two invariant properties denoted by inv7 and inv8, the ensures property denoted by ens9 and the leads-to property denoted by lto13. The verification also needs the lemma nat-lemma2 on natural numbers. The case is split into four sub-cases. The four sets of equations for the four sub-cases are as follows:

1. n = 0
2. n = s(m), where(queue(s),i) = m, (s(m) = m) = false
3. n = s(m), where(queue(s),i) = m, s(m) = m
4. n = s(m), (where(queue(s),i) = m) = false

where m is a constant of Nat. The term s(m) denotes an arbitrary positive natural number. Note that human users are responsible for checking if all cases are covered.

The proof score (which consists of four proof passages) is as follows:

```
open LTO
  eq n = 0 .
  red (inv7(s,i) /\ ens9(s,i))
      => lto14(s,i,n) .
close

open LTO
  op m : -> Nat .  eq n = s(m) .
  eq where(queue(s),i) = m .
  eq (s(m) = m) = false .
  red lto14(s,i,n) .
close

open LTO
  op m : -> Nat .  eq n = s(m) .
  eq where(queue(s),i) = m .  eq s(m) = m .
  red nat-lemma2(m) implies lto14(s,i,n) .
close

open LTO
```

```
  op m : -> Nat .  eq n = s(m) .
  eq (where(queue(s),i) = m) = false .
  red (inv8(s,i,m) /\ lto13(s,i,m))
      => lto14(s,i,n) .
close
```

CafeOBJ returns `true` for each of the four proof passages.

5.2 Proof Score of `lto15`

The leads-to property denoted by `lto15` is deduced from the leads-to property denoted by `lto14`. The proof score is as follows:

```
open LTO
  pr(INDOFLTO(PNAT)) .
  red lto14(s,i,n) => lto15(s,i) .
close
```

CafeOBJ returns `true` for this proof score.

5.3 Proof Score of `ens11`

The two state predicates in the ensures property denoted by `ens11` are denoted by the operators `unl11-1` and `unl11-2`, which are defined in a module `UNL` (which imports `QLOCK` and `INV`) as follows:

```
eq unl11-1(S,I,J,N)
   = (pc(S,I) = wt and I \in queue(S) and
      where(queue(S),I) = s(N) and
      top(queue(S)) = J and pc(S,J) = wt) .
eq unl11-2(S,I,J,N)
   = (pc(S,I) = wt and I \in queue(S) and
      where(queue(S),I) = s(N) and
      pc(S,top(queue(S))) = cs) .
```

`S`, `I`, `J` and `N` are CafeOBJ variables of `Sys`, `Pid`, `Pid` and `Nat`, respectively. The constants `i`, `j` and `n` of `Pid`, `Pid` and `Nat` are declared in `UNL`.

The basic formula to prove in the eventually case is denoted by the operator `estep11`, which is defined in a module `ESTEP` (which imports `UNL`) as follows:

```
eq estep11(I,J,N)
   = (unl11-1(s,I,J,N) and
      not unl11-2(s,I,J,N))
     implies unl11-2(s',I,J,N) .
```

The basic formula to prove in the `unless` case is denoted by the operator `ustep11`, which is defined in a module `USTEP` (which imports `UNL`) as follows:

```
eq ustep11(I,J,N)
   = (unl11-1(s,I,J,N) and
      not unl11-2(s,I,J,N))
     implies (unl11-1(s',I,J,N) or
              unl11-2(s',I,J,N)) .
```

In the eventually case, all needed is to prove that there exists a witness, namely an instance of a transition, that makes `estep11(i,j,m)` true. We conjecture that $\mathrm{try}_{\mathtt{j}}$ is such a witness, which is confirmed by writing a proof score. To this end, the case is split into five sub-cases. The five sets of equations for the five sub-cases are as follows:

1. `queue(s) = empty`
2. `queue(s) = k,q`, `(k = j) = false`
3. `queue(s) = k,q`, `k = j`, `i = j`
4. `queue(s) = k,q`, `k = j`, `(i = j) = false`, `(pc(s,j) = wt) = false`
5. `queue(s) = k,q`, `k = j`, `(i = j) = false`, `pc(s,j) = wt`

where `k` is a constant (of `Pid`) denoting an arbitrary process ID and `q` is a constant (of `Queue`) denoting an arbitrary queue. The term `k,q` denotes an arbitrary non-empty queue.

The proof scores of the fifth sub-case is shown:

```
open ESTEP
  op k : -> Pid .  op q : -> Queue .
  eq queue(s) = k,q .  eq k = j .
  eq (i = j) = false .  eq pc(s,j) = wt .
  eq s' = try(s,j) .
  red estep11(i,j,n) .
close
```

CafeOBJ returns `true` for the proof passage. The proof scores of the remaining four sub-cases are written likewise.

In the `unless` case, all we have to do is to prove `istep11(i,j,n)` for each instance of every transition (every action operator). We describe the proof that an arbitrary instance want_k of the transition want makes `istep11(i,j,n)` true. For the proof, the case is split into five sub-cases. The five sets of equations for the five sub-cases are as follows:

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

where `l` is a constant (of `Pid`) denoting an arbitrary process ID and `q` is a constant (of `Queue`) denoting an arbitrary queue. Note that the equation `pc(s,k) = cs` is equivalent to the equation `c-exit(s,k) = true`.

The proof of the fourth sub-case needs an invariant property wrt $\mathcal{S}_{\mathrm{Qlock}}$ and those of the remaining sub-cases do not need any invariant properties and any lemmas on data types. The invariant property needed for the proof of the fourth sub-case is denoted by the operator `inv2`, which is defined in the module `INV` as follows:

```
eq inv2(S,I) = (pc(S,I) = cs
                implies top(queue(S)) = I) .
```

The proof score of the fourth sub-cases is shown:

```
open USTEP
  op k : -> Pid .  op l : -> Pid .
  op q : -> Queue .
  eq pc(s,k) = cs .  eq (i = k) = false .
  eq queue(s) = l,q .  eq (l = k) = false .
  eq s' = exit(s,k) .
  red inv2(s,k) implies ustep11(i,j,n) .
close
```

CafeOBJ returns `true` for the proof passage.

The proof passages of the remaining four sub-cases are written likewise. The proof passages that arbitrary instances of the remaining transitions want and try make `istep11(i,j,n)` true are also written likewise, which need the three lemmas (`queue-lemma3`, `queue-lemma8` and `queue-lemma9`; see Appendix A) on queues.

6. Related Work

Many methodologies have been proposed to verify that systems and/or programs satisfy liveness as well as safety properties. Among such methodologies are UNITY [11], IOA [21] and TLA [22]. Moreover, many formal tools have been developed, which can be used to formalize (or mechanize) such methodologies. Among such tools are CafeOBJ [6], Larch [23], Isabelle [24], Coq [25] and HOL [26]. IOA has been formalized in Larch [27], TLA has been formalized in Larch and HOL [28], [29], and UNITY has been formalized in Isabelle, Coq and HOL [12]–[14]. The OTS/CafeOBJ method has been largely influenced by UNITY and may be regarded as a method of supporting formal verification of UNITY programs modeled as OTSs. Therefore, we summarize one approach to formalizing UNITY.

Paulson [12] proposes a way to formalize (or mechanize) UNITY in Isabelle [24], which is a proof assistant based on higher-order logic. The UNITY he uses is new UNITY [30], [31], while the UNITY that affects the OTS/CafeOBJ method is classic UNITY [11]. The most primitive property (or temporal operator) is unless in classic UNITY, while it is co (or next) in new UNITY. Given two state predicates p and q, p co q is defined as follows: whenever p holds in a state of a UNITY program, every statement of the program makes q hold in the successor state. unless can be defined in terms of co, namely $(p \wedge \neg q)$ co $(p \vee q)$, which is equal to p unless q. Paulson formalizes the six basic properties (co, stable, invariant, transient, ensures and leads-to) in Isabelle and reason about theorems on safety and liveness properties with Isabelle. Given a state predicate p, transient p is defined as follows: whenever p holds in a state of a UNITY program, there exists a statement of the program that makes p false in the successor state. He also defines the weak version of the six basic properties so that the basic properties satisfy the substitution axiom [32]. The weak version takes into account reachable states of programs. The definition of the five basic properties in this paper correspond to the weak version.

Recently much attention has been paid to model checking [33] because it can verify fully automatically that systems and/or programs satisfy safety and liveness properties. Many model checkers have been developed. Basically, however, systems that can be model checked should be finite state. On the other hand, interactive theorem proving such as the OTS/CafeOBJ method and Isabelle/UNITY can also be applied to infinite-state systems. Although abstraction methods [34] make a finite-state abstract version of a given (infinite-state) system and make it possible to model check the abstract version with respect to a given property, it is necessary to prove that the abstraction used preserves the property, which usually needs (interactive) theorem proving.

7. Conclusion

We have described a way to verify that designs for systems satisfy liveness properties in the OTS/CafeOBJ method. A mutual exclusion protocol called Qlock has been used as an example, and it has been verified that Qlock satisfies the lockout freedom property, which can be expressed as a leads-to property.

Proof scores of ensures properties are quite similar to ones of invariant properties [3]–[5], and so are ones of unless and stable properties. This means that the verification techniques and tips [3] devised for verification of invariant properties can be used for verification of ensure properties.

In addition to the mutual exclusion protocol, we have applied the proposed verification method to a workflow system that takes into consideration some security policies [35], [36]. We still need to apply the method to a wider variety of applications to demonstrate the usefulness of the proposed verification method.

References

[1] K. Ogata and K. Futatsugi, "Proof score approach to verification of liveness properties," 17th International Conference on Software Engineering and Knowledge Engineering (17th SEKE), pp.608–613, Knowledge Systems Institute, 2005.
[2] K. Futatsugi, J.A. Goguen, and K. Ogata, "Verifying design with proof scores," IFIP Working Conference on Verified Software: Theories, Tools, and Experiments (VSTTE), 2005.
[3] K. Ogata and K. Futatsugi, "Some tips on writing proof scores in the OTS/CafeOBJ method," Algebra, Meaning, and Computation: Essays Dedicated to Joseph A. Goguen on the Occasion of His 65th Birthday, LNCS, vol.4060, pp.596–615, Springer, 2006.
[4] K. Ogata and K. Futatsugi, "Proof scores in the OTS/CafeOBJ method," 6th IFIP WG6.1 International Conference on Formal Methods for Open Object-Based Distributed Systems (6th FMOODS), LNCS, vol.2884, pp.170–184, Springer, 2003.
[5] R. Diaconescu, K. Futatsugi, and K. Ogata, "CafeOBJ: Logical foundations and methodologies," Computing and Informatics, vol.22, pp.257–283, 2003.
[6] R. Diaconescu and K. Futatsugi, CafeOBJ report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification, AMAST Series in Computing, vol.6, World Scientific, 1998.
[7] K. Futatsugi, "Verifying specifications with proof scores in CafeOBJ," 21st International Conference on Automated Software Engineering (ASE 2006), pp.3–10, IEEE Computer Society Press, 2006.

[8] K. Ogata and K. Futatsugi, “Formal analysis of the *i*KP electronic payment protocols,” 1st International Symposium on Software Security (1st ISSS), LNCS, vol.2609, pp.441–460, Springer, 2003.
[9] K. Ogata and K. Futatsugi, “Equational approach to formal analysis of TLS,” 5th International Conference on Distributed Computing Systems (25th ICDCS), pp.795–804, IEEE Computer Society Press, 2005.
[10] W. Kong, K. Ogata, and K. Futatsugi, “Algebraic approaches to formal analysis of the Mondex electronic purse system,” 6th International Conference on Integrated Formal Methods (6th IFM), LNCS, vol.4591, pp.393–412, Springer, 2007.
[11] K.M. Chandy and J. Misra, Parallel Program Design: A Foundation, Addison-Wesley, 1988.
[12] L.C. Paulson, “Mechanizing UNITY in Isabelle,” ACM Trans. Computational Logic, vol.1, no.1, pp.3–32, 2000.
[13] B. Heyd and P. Crégut, “A modular coding of UNITY in Coq,” 9th International Conference on Theorem Proving in Higer Order Logics (9th TPHOLs), LNCS, vol.1125, pp.251–266, Springer, 1996.
[14] F. Anderson, K. Petersen, and J. Petterson, “Program verification using HOL-UNITY,” 6th International Workshop on Higher Order Logic and its Applications (6th HUG), LNCS, vol.780, pp.1–16, Springer, 1993.
[15] J. Pachl, “A simple proof of a completeness result for leads-to in the UNITY logic,” Inf. Process. Lett., vol.41, pp.35–38, 1992.
[16] J. Goguen and G. Malcolm, “A hidden agenda,” Theor. Comput. Sci., vol.245, pp.55–101, 2000.
[17] R. Diaconescu and K. Futatsugi, “Behavioural coherence in object-oriented algebraic specification,” J. Universal Computer Science, vol.6, pp.74–96, 2000.
[18] J. Hsiang and N. Dershowitz, “Rewrite methods for clausal and non-clausal theorem proving,” 10th EATCS International Colloquium on Automata, Languages, and Programming (10th ICALP), LNCS, vol.154, pp.331–346, Springer, 1983.
[19] T. Seino, K. Ogata, and K. Futatsugi, “A toolkit for generating and displaying proof scores in the OTS/CafeOBJ method,” 6th International Workshop on Rule-Based Programming (6th RULE), ENTCS, vol.147, pp.57–72, Elsevier, 2006.
[20] M. Nakano, K. Ogata, M. Nakamura, and K. Futatsugi, “Creme: An automatic invariant prover of behavioral specifications,” Int. J. Software Engineering and Knowledge Engineering, vol.17, no.6, pp.783–804, 2007.
[21] N.A. Lynch, Distributed Algorithms, Morgan-Kaufmann, San Francisco, CA, 1996.
[22] L. Lamport, “The temporal logic of actions,” ACM Trans. Programming Languages and Systems, vol.16, no.3, pp.872–923, 1994.
[23] J.V. Guttag, J.J. Horning, S.J. Garland, K.D. Jones, A. Modet, and J.M. Wing, Larch: Languages and Tools for Formal Specification, Springer, 1993.
[24] T. Nipkow, L.C. Paulson, and M. Wenzel, Isabelle/HOL: A Proof Assistant for Higher-Order Logic, LNCS, vol.2283, Springer, Berlin, 2002.
[25] Y. Bertot and P. Castéran, Interactive Theorem Proving and Program Development – Coq’Art: The Calculus of Inductive Constructions, Springer, Berlin, 2004.
[26] M.J.C. Gordon and T.F. Melham, Introduction to HOL: A Theorem Proving Environment for Higher Order Logic, Cambridge University Press, 1993.
[27] S.J. Garland and N.A. Lunch, “The IOA language and toolset: Supprt for designing, analyzing, and building distributed systems,” Technical Report MIT/LCS/TR-762, 1998.
[28] U. Engberg, P. Gronning, and L. Lamport, “Mechanical verification of concurrent systems with TLA,” 4th International Conference on Computer Aided Verification (4th CAV), LNCS, vol.663, pp.44–55, Springer, 1992.
[29] T. Langbacka, “A HOL formalization of the temoral logic of actions,” 7th International Workshop on Higher Order Logic and its Applications (7th HUG), LNCS, vol.859, pp.332–345, Springer, 1994.
[30] J. Misra, “A logic for concurrent programming: Safety,” J. Computer and Software Engineering, vol.3, no.2, pp.239–272, 1995.
[31] J. Misra, “A logic for concurrent programming: Progress,” J. Computer and Software Engineering, vol.3, no.2, pp.273–300, 1995.
[32] B.A. Sanders, “Eliminating the substitution axiom from UNITY logic,” Formal Aspects of Computing, vol.3, no.2, pp.189–205, 1991.
[33] J. Edmund, M. Clarke, O. Grumberg, and D.A. Peled, Model Checking, The MIT Press, 2001.
[34] E.M. Clarke, O. Grumberg, and D.E. Long, “Model checking and abstraction,” ACM TOPLAS, vol.16, no.5, pp.1512–1542, 1994.
[35] W. Kong, K. Ogata, and K. Futatsugi, “Formal analysis of workflow systems with security considerations,” 17th International Conference on Software Engineering and Knowledge Engineering (17th SEKE), pp.531–536, Knowledge Systems Institute, 2005.
[36] W. Kong, K. Ogata, and K. Futatsugi, “Specification and verification of workflows with RBAC mechanism and SoD constraints,” Int. J. Software Engineering and Knowledge Engineering, vol.17, no.1, pp.3–32, 2007.

Appendix A: Lemmas on Data Types

The lemmas on natural numbers that are needed for the verification of Qlock on the lockout freedom property are as follows:

```
eq nat-lemma1(X,Y)
   = (X = Y iff s(X) = s(Y)) .
eq nat-lemma2(X) = not(X = s(X)) .
```

where `X` and `Y` are CafeOBJ variables of `Nat`. The equation is declared in the module `PNAT`.

The lemmas on queues that are needed for the verification are as follows:

```
eq queue-lemma1(Q,X) = X \in put(Q,X) .
eq queue-lemma2(Q,X,Y)
   = X \in Q implies X \in put(Q,Y) .
eq queue-lemma3(Q,X,Y) = (X \in put(Q,Y)
                iff (X = Y or X \in Q)) .
eq queue-lemma4(Q,X)
   = (X \in get(Q) implies X \in Q) .
eq queue-lemma5(Q,X)
   = (X \in del(put(Q,X),X) iff X \in Q) .
eq queue-lemma6(Q,X,Y)
   = (not(X = Y) and X \in del(put(Q,Y),X)
      implies X \in del(Q,X)) .
eq queue-lemma7(Q,X)
   = (X \in del(Q,X) implies X \in Q) .
eq queue-lemma8(Q,X) = X \in Q implies
          (top(Q) = X iff where(Q,X) = 0) .
eq queue-lemma9(Q,X,Y,N) = X \in Q implies
       (aux-where(Q,X) = N
        implies aux-where(put(Q,Y),X) = N) .
eq queue-lemma10(Q,X) = X \in Q implies
            (where(Q,X) = aux-where(Q,X)) .
```

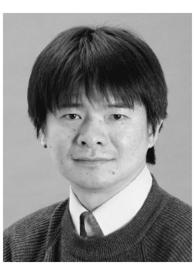
where `Q`, `X`, `Y` and `N` are CafeOBJ variables of `Queue`, `Elt.D`, `Elt.D` and `Nat`, respectively. The equations are declared in the module `QUEUE`.

Appendix B: Invariant Properties

The invariant properties wrt $\mathcal{S}_{\mathrm{Qlock}}$ that are needed for the verification are as follows:

```
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) = (I \in queue(S)
       iff (pc(S,I) = wt or pc(S,I) = cs)) .
eq inv4(S,I) = (top(queue(S)) = I
         implies not(I \in get(queue(S)))) .
eq inv5(S,I) = not(I \in del(queue(S),I)) .
eq inv6(S,I) = (pc(S,I) = wt implies
          pc(S,I) = wt and I \in queue(S)) .
eq inv7(S,I)
   = (pc(S,I) = wt and I \in queue(S)
                and where(queue(S),I) = 0
      implies
      pc(S,I) = wt and top(queue(S)) = I) .
eq inv8(S,I,N)
   = (pc(S,I) = wt and I \in queue(S)
             and where(queue(S),I) = s(N)
      implies
      pc(S,I) = wt and I \in queue(S)
           and where(queue(S),I) = s(N)
           and (pc(S,top(queue(S))) = wt or
                pc(S,top(queue(S))) = cs)) .
```

where `S`, `I`, `J` and `N` are CafeOBJ variables of `Sys`, `Pid`, `Pid`, `Nat`, respectively. The equations are declared in the module `INV`.



Kazuhiro Ogata is a research associate professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his Ph.D. in engineering from Graduate School of Science and Technology, Keio University in 1995. He was a research associate at JAIST from 1995 to 2001, a researcher at SRA Key Technology Laboratory, Inc. from 2001 to 2002, and a research expert at NEC Software Hokuriku, Ltd. from 2002 to 2006. Among his research interests are software engineering, formal methods and formal verification.

Kokichi Futatsugi is a professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). Before getting a full professorship at JAIST in 1993, he was working for ETL (Electrotechnical Lab.) of Japanese Government and was assigned to be Chief Senior Researcher of ETL in 1992. His research interests include formal methods, system verifications, software requirements/specifications, language design, concurrent and cooperative computing. His primary research goal is to design and develop new languages which can open up new application areas, and/or improve the current software technology. His current approach for this goal is CafeOBJ formal specification language. CafeOBJ is multi-paradigm formal specification language which is a modern successor of the most noted algebraic specification language OBJ.