IEICE TRANS. INF. & SYST., VOL.E91–D, NO.5 MAY 2008

PAPER

A Specification Translation from Behavioral Specifications to Rewrite Specifications*

Masaki NAKAMURA[†a)], ***Member***, **Weiqiang KONG**[†], ***Nonmember***, **Kazuhiro OGATA**[†], ***Member***, ***and*** **Kokichi FUTATSUGI**[†], ***Nonmember***

SUMMARY There are two ways to describe a state machine as an algebraic specification: a behavioral specification and a rewrite specification. In this study, we propose a translation system from behavioral specifications to rewrite specifications to obtain a verification system which has the strong points of verification techniques for both specifications. Since our translation system is complete with respect to invariant properties, it helps us to obtain a counter-example for an invariant property through automatic exhaustive searching for a rewrite specification.

key words: *specification translation, verification, algebraic specification, behavioral specification, rewirte specification, CafeOBJ, Maude*

1. Introduction

There are many kinds of formal specification languages to support formal methods. Algebraic specification languages, e.g. OBJ3, CafeOBJ, Maude, are formal specification languages whose specifications denote algebras. Unlike specification languages based on first-order predicate logic, for example, Z notation, algebraic specification languages have been developed with initial algebras as a mathematical theory of abstract data types together with term rewriting as a computational theory of abstract data types. In this paper we focus on two kinds of algebraic specifications: behavioral specifications and rewrite specifications.

A behavioral specification specifies behaviors of a system, and it denotes a set of all algebras satisfying the described behavior, that is, it specifies all implementations satisfying the behavior. A rewrite specification specifies local concurrent transitions of a system, and it denotes the term algebra (or an initial algebra) with the rewrite relation, that is, it specifies essentially just one implementation. Roughly speaking, we can specify a system in a higher abstract level by a behavioral specification than a rewrite specification. When we verify a property for a behavioral specification, all its implementations are guaranteed to satisfy the property. A fully-automatic verification system, for example, the search command and a model checker, can be applied to rewrite specifications and cannot be applied to behavioral specifications directly. It gives us a way not only to prove a property but also to disprove it with a counter-example. For example, we describe a semaphore system in this paper. In a behavioral specification the set of processes can be an abstract set, and any kind of processes sets can be a model of the specification. On the other hand, to describe a rewrite specification, we need to decide a concrete set of processes. In addition we need to restrict the number of processes to finite to apply a fully-automatic verification system.

We propose a translation system from behavioral specifications to rewrite specifications, and show the translation is complete w.r.t. invariant properties. The invariant property is a foundational property for state transition systems. If a state property is invariant, the property holds for every reachable state. The invariant property is often used to express safety properties, for example, the property that no intruder can decrypt any encrypted data in a security protocol. Our translation system takes a behavioral specification written in CafeOBJ language[**] [4] and returns a rewrite specification written in Maude language[***]. The CafeOBJ system has a semi-automatic equational reasoning, which helps to verify a property interactively. The Maude system supports fully-automatic exhaustive search command and a model checker. By our translation system, we may find a counter-example of a CafeOBJ behavioral specification through translating it into a Maude specification and applying the Maude search command.

2. Preliminaries

A finite sequence of $a_1, a_2, \ldots a_n$ is denoted by $\overrightarrow{a}$, whose length is observed by $ln(\overrightarrow{a}) = n$. We may use set notations for a sequence if there is no confusion, e.g., $a \in \overrightarrow{b}$ stands for $\exists i.a = b_i$, $\{\mathtt{a}_i \mid 0 < i < 4\}$ stands for $\mathtt{a}_1\ \mathtt{a}_2\ \mathtt{a}_3$, etc. For a set S, an S-sorted set A is a family $\{A_s \mid s \in S\}$ whose element is a set associated to each $s \in S$. In this section we introduce the notion of basic algebraic specifications which is a common part of both CafeOBJ and Maude specifications, and is used for data types of a target system.

An algebraic specification consists of modules. The following is an example of CafeOBJ modules which specify natural numbers and the addition operation on them.

```
mod! BASIC-NAT{
 [Zero NzNat < Nat]
 op 0 : -> Zero
```

Manuscript received January 22, 2007.
Manuscript revised August 6, 2007.

[†]The authors are with School of Information Science, Japan Advanced Institute of Science and Technology, Nomi-shi, 923–1211 Japan.

[*]A preliminary version of this article appeared in [9].

a) E-mail: masaki-n@jaist.ac.jp

DOI: 10.1093/ietisy/e91–d.5.1492

[**]http://www.ldl.jaist.ac.jp/cafeobj/

[***] http://maude.cs.uiuc.edu/

```
 op s_ : Nat -> NzNat
}
mod! NAT+{
 pr(BASIC-NAT)
 op _+_ : Nat Nat -> Nat
 vars M N : Nat
 eq N + 0 = N .
 eq N + s M = s(N + M) .
}
```

A module consists of an import part, a signature part, an axiom part. In the import part, submodules M_i imported by the main module are listed with their import modes, e.g. `pr`(M_1) `ex`(M_2)$\cdots$. In the signature part, sorts and operations are declared. A sort s is a kind of types, which interpreted into a carrier set M_s in its denotational model (algebra) M. An inclusion relation can be given on sorts. An operation symbol f with an arity $\vec{s}$ of a sequence of sorts and a coarity s of a sort denotes an operation or a function $M_f : M_{s_1} \times \cdots \times M_{s_n} \to M_s$. For example, the algebra N of natural numbers is a model of `BASIC-NAT`, whose interpretation is the following: $N_{\texttt{Zero}} = \{0\}$, $N_{\texttt{NzNat}} = \mathcal{N}_+ = \mathcal{N} - \{0\}$, $N_{\texttt{Nat}} = \mathcal{N} = \{0, 1, 2, \ldots\}$, $N_{\texttt{0}} = 0$ and $N_{\texttt{s}}(n) = n + 1$. A term is a well-sorted tree whose nodes are operation symbols and leaves are variables. For a given S-sorted set V of variables, an assignment $a : V \to M$ is an S-sorted map where a_s is a map from V_s to M_s. For a term t of a sort s and an assignment $a : V \to M$ whose V is the set of all variables in t, t is interpreted into an element of M_s, denoted by $a(t)$, as follows: $a(t) = a(x)$ if $t = x \in V$ and $a(t) = M_f(a(t_1), \ldots, a(t_n))$ if $t = f(\vec{t})$. An equation is a pair of terms of a same sort. A conditional equation is a pair of an equation and a conditional term of Boolean sort. They are declared after `eq` and `ceq` respectively. We may call both of them just equations. In the axiom part equations are declared. An algebra is a model of a specification if and only if the left-hand side and the right-hand side of each equation are interpreted into a same element for any assignment. The algebra N can also be a model of `NAT+` if + is interpreted into $N_+(m, n) = m + n$.

CafeOBJ has two kinds of denotations: loose and tight. A specification with the loose denotation, written by `mod*`, denotes the set of all models. A tight specification, `mod!`, denotes the set of all initial models. An initial model is a model M satisfying that each element $e \in M_s$ has a corresponding term t, i.e. $M_t = e$, and that $t = t'$ can be deduced from the axiom whenever $M_t = M_{t'}$. The algebra N is an initial model of `NAT+`. For example, Boolean algebra (with $B_0 = false$, $B_+ = \vee$, etc) or the algebra of integers can be a model of `NAT+` but is not an initial model. CafeOBJ supports imports of modules. A CafeOBJ specification can import several modules, which already have been described or are built-in modules of CafeOBJ system. There are several kinds of imports. Specifications with import declarations also have loose or tight denotation declarations. Formal semantics of specifications with imports can be found in [4]. In this study, we mainly treat a loose specification with protected imports, denoted by `pr`(SP). Roughly speaking, the denotation of a loose specification SP which imports SP' with the protect mode is a set of all algebras M which satisfy (1) all equations in SP as well as SP', and (2) that M is an expansion of M' for some M' in the denotation of SP', where an expansion means that $M_e = M'_e$ for all sorts and operation symbols e of SP'[†].

A Maude data type specification, called a functional specification, denotes the term algebra, which is one of the initial algebras. In the term of CafeOBJ denotations, any Maude specification has the tight denotation. Maude does not support a loose denotation. Another limitation of Maude functional specifications is that they should be complete in the meaning of the term rewriting system (See [1] for the definition of the completeness). Roughly speaking, if a specification is complete, (i) it is decidable whether $t = t'$ can be deduced from the equations in the specification or not, (ii) each term has its unique normal form, which can be seen as a representative term of an interpreted element or an equivalence class of terms $[t] = \{t' \mid t = t'\}$. For example, `NAT+` is complete, and `s s 0` is a (unique) normal form of terms `s 0 + s 0`, `s s 0 + 0` and `0 + s s 0`, all of which are interpreted into 2. Thanks to the limitation of the tightness and the completeness, Maude supports useful automatic verification tools. For a complete specification SP, one of the denotation of SP is an algebra whose carrier set of sort s is the set of all ground (i.e. variable-free) normal forms of sort s (denoted by NF_s), called the term algebra. The term algebra is an initial model of SP.

3. Algebraic Description of State Machines

In this section, we introduce how to describe a state machine in CafeOBJ and Maude.

3.1 State Machines

A state machine consists of a set S of states, a set I of initial states and a set T of transitions, where $\tau \in T$ is a binary relation on S. We write $\tau(s) = s'$ if $(s, s') \in \tau$ for $\tau \in T$. For a state machine $\mathcal{S} = (S, I, T)$, the set $Re_{\mathcal{S}}$ of reachable states is defined as the smallest set satisfying the following: $I \subseteq Re_{\mathcal{S}}$ and $s' \in Re_{\mathcal{S}}$ if $\tau(s) = s'$ for some $s' \in Re_{\mathcal{S}}$ and $\tau \in T$. A state property P is a predicate on S. We say P is invariant, denoted by $S \models Inv_P$, if and only if $P(s)$ for any $s \in Re_{\mathcal{S}}$.

3.2 OTS/CafeOBJ Specifications

Observation Transition System (OTS) is a useful notion to describe a state machine in CafeOBJ [10]. An OTS/CafeOBJ specification has a special sort, called a hidden sort, and special operation symbols, called observation

[†]It is possible that there is no algebra which satisfies the conditions (1) and (2). For that case, the specification denotes the empty set, and we call the specification inconsistent. We assume that an input specification is not inconsistent in this paper.

symbols and transition symbols whose arities include exactly one hidden sort. We call a non-hidden sort a visible sort. The co-arity of an observation symbol (or a transition symbol) is a visible sort (or a hidden sort). In a denotational model M of an OTS/CafeOBJ specification, the carrier set M_H of the hidden sort H is called a state space, and an element $s \in M_H$ is called a state. States s and t are observationally equivalent, denoted by $s \sim t$, if and only if $M_o(s, \vec{u}) = M_o(t, \vec{u})$ for each observation symbol o and each elements $\vec{u}$ corresponding to the visible arguments of o. Without loss of generality, each observation symbol (and transition symbol) is assumed to have the hidden sort at the first argument. A transition symbol τ must preserve the observational equivalence, i.e. $M_\tau(s, \vec{u}) \sim M_\tau(t, \vec{u})$ whenever $s \sim t$. In this section, we give a syntactical definition of inputs of our translation system, and show that each input specification is an OTS/CafeOBJ specification. We can say the syntactical definition is another definition (in the narrow sense) of OTS/CafeOBJ specifications since the OTS/CafeOBJ specifications satisfy the definition or can be easily modified into those satisfying it in most practical applications of OTS [11]–[14]. Hereafter, we use the word "an OTS/CafeOBJ specification" for a specification satisfying the following definition.

Definition 3.1: An OTS/CafeOBJ specification has the following syntax:

$$
\begin{aligned}
&\mathit{OTS} ::= \overrightarrow{\mathit{SubModule}}\ \mathit{MainModule}\\
&\mathit{SubModule}_i ::= \texttt{mod(!|*)}\ \ SM_i\ \ \texttt{\{}\cdots\texttt{\}}\\
&\mathit{MainModule} ::= \texttt{mod*}\ \ M\ \ \texttt{\{}\\
&\qquad\qquad \texttt{pr(}SM_1\texttt{)}\cdots\texttt{pr(}SM_l\texttt{)}\\
&\qquad\qquad \texttt{*[}H\texttt{]*}\\
&\qquad\qquad \mathit{Signature}\\
&\qquad\qquad \mathit{Axiom}\\
&\qquad\qquad \texttt{\}}\\
&\mathit{Signature} ::= \mathit{Init}\ \overrightarrow{\mathit{Obs}}\ \overrightarrow{\mathit{Trans}}\\
&\mathit{Init} ::= \texttt{op init : -> }H\texttt{.}\\
&\mathit{Obs}_i ::= \texttt{bop}\ \ o_i\ \ \texttt{:}\ \ H\ \overrightarrow{V_{o_i}}\ \ \texttt{->}\ \ V_{o_i}\\
&\mathit{Trans}_i ::= BT_i\ CT_i\\
&BT_i ::= \texttt{bop}\ \ \tau_i\ \ \texttt{:}\ \ H\ \overrightarrow{V_{\tau_i}}\ \ \texttt{->}\ \ H\\
&CT_i ::= \texttt{bop c-}\tau_i\ \ \texttt{:}\ \ H\ \overrightarrow{V_{\tau_i}}\ \ \texttt{->}\ \ \texttt{Bool}\\
&\mathit{Axiom} ::= \mathit{Vars}\ A_{\mathit{init}}\ A_{\tau_1}\cdots A_{\tau_m}\\
&\mathit{Vars} ::= VH\ \overrightarrow{VO_1}\cdots\overrightarrow{VO_m}\ \overrightarrow{VT_1}\cdots\overrightarrow{VT_n}\\
&VH ::= \texttt{var S:}H\\
&VO_{ij} ::= \texttt{var X}ij\texttt{:}\ \ V_{o_ij}\\
&VT_{ij} ::= \texttt{var Y}ij\texttt{:}\ \ V_{\tau_ij}\\
&A_{\mathit{init}} ::= A_{\mathit{init}}^{o_1}\cdots A_{\mathit{init}}^{o_m}\\
&A_{\mathit{init}}^{o_i} ::= \texttt{eq}\ \ o_i\texttt{(init,}X_i\texttt{)}\ =\ \mathit{rinit}_i\texttt{.}\\
&A_{\tau_i} ::= A_{\tau_i}^{o_1}\cdots A_{\tau_i}^{o_m}\ A_{\tau_i}^{\perp}\ A_{\tau_i}^{C}\\
&A_{\tau_i}^{o_j} ::= \texttt{ceq}\ \ o_j\texttt{(}\tau_i\texttt{(S,}Y_i\texttt{),}X_j\texttt{)}\ =\ ro_j\tau_i\\
&\qquad\qquad \texttt{if c-}\tau_i\texttt{(S,}Y_i\texttt{)}\ \texttt{.}\\
&A_{\tau_i}^{\perp} ::= \texttt{bceq}\ \ \tau_i\texttt{(S,}Y_i\texttt{)}\ =\ \texttt{S}\\
&\qquad\qquad \texttt{if not(c-}\tau_i\texttt{(S,}Y_i\texttt{))}\ \texttt{.}\\
&A_{\tau_i}^{C} ::= \texttt{eq c-}\tau_i\texttt{(S,}Y_i\texttt{)}\ =\ cr\tau_i\texttt{.}\\
&X_i ::= \texttt{X}i1\texttt{,X}i2\ldots\texttt{X}im_i\\
&Y_j ::= \texttt{Y}j1\texttt{,Y}j2\ldots\texttt{Y}jn_j
\end{aligned}
$$

where each $\mathit{SubModule}_i$ is a module specifying data type, explained in the previous section. SM_i is a name of the module $\mathit{SubModule}_i$. H is a hidden sort. o_i is an observation symbol and τ_i is a transition symbol. Any identifier can be used for H, o_i and τ_i. V_{o_i}, V_{τ_i} are visible sorts which are declared in the submodules. rinit_i are terms constructed by operation symbols declared in submodules and variables occurred in the left-hand sides, which means that no observation symbols and transition symbols occur. $ro_j\tau_i$ and $cr\tau_i$ are terms constructed by the observation symbols, operation symbols declared in submodules, and variables occurred in the left-hand sides, which means that no transition symbols occur. All terms should not have nested observation symbols, which means that in any path from the root to a leaf at most one observation symbol exists, e.g. $o_j(\mathtt{S}, o_k(\mathtt{S}))$ is not allowed. $ln(\overrightarrow{\mathit{SubModule}}) = l$, $ln(\overrightarrow{\mathit{Obs}}) = m$, $ln(\overrightarrow{\mathit{Trans}}) = n$, $ln(\overrightarrow{V_{o_i}}) = m_i$, $ln(\overrightarrow{V_{\tau_i}}) = n_i$. Note that `beq` and `bceq` are used for equations on hidden terms, called behavioral equations. In OTS, the behavioral equation is identified with the observational equation $\sim$. □

Hereafter, all definitions and proofs are built on the notation of Definition 3.1, for example, when we take an observation symbol o_i, its sort is $H\ \overrightarrow{V_{o_i}}$ -> V_{o_i}.

Theorem 3.2: Each OTS/CafeOBJ specification satisfies the OTS condition, i.e. transitions preserve the observational equivalence.

Proof. Let τ_i be a transition symbol and o_j an observation symbol. Let M be a denotational model of the OTS/CafeOBJ specification, (s, s') a pair of observationally equivalent states, i.e. $s \sim s'$, and $\vec{u}$, $\vec{v}$ sequences of elements of M such that $u_k \in M_{V_{\tau_i k}}$ and $v_k \in M_{V_{o_j k}}$. It suffices to show that $M_{o_j}(M_{\tau_i}(s, \vec{u}), \vec{v}) = M_{o_j}(M_{\tau_i}(s', \vec{u}), \vec{v})$. Let a, a' be assignments such that $a(\mathtt{S}) = s$, $a'(\mathtt{S}) = s'$, $a(\mathtt{X}_{jk}) = a'(\mathtt{X}_{jk}) = v_k$ and $a(\mathtt{Y}_{ik}) = a'(\mathtt{Y}_{ik}) = u_k$. Note that those variable symbols $\mathtt{S}$, $\mathtt{X}_{jk}$ and $\mathtt{Y}_{ik}$ are declared in the OTS/CafeOBJ specification (See Definition 3.1). First we prove $M_{\mathtt{C}\text{-}\tau_i}(s, \vec{u}) = M_{\mathtt{C}\text{-}\tau_i}(s', \vec{u})$. Since M is a denotational model, $a(l) = a(r)$ holds for each equation $l = r$ in OTS. Thus, $M_{\mathtt{C}\text{-}\tau_i}(s, \vec{u}) = a(\mathtt{c}\text{-}\tau_i(\mathtt{S}, \mathtt{X}_j)) = a(cr\tau_i)$ and $M_{\mathtt{C}\text{-}\tau_i}(s', \vec{u}) = a'(cr\tau_i)$ hold. The only difference between $a(cr\tau_i)$ and $a'(cr\tau_i)$ is s and s'. Since the term $cr\tau_i$ has no transition symbols, s (and s') should be directly under an observation symbol, i.e. like $o(s, \ldots)$ and $o(s', \ldots)$. Since $s \sim s'$, $M_o(s, \ldots) = M_o(s', \ldots)$. Thus, $M_{\mathtt{C}\text{-}\tau_i}(s, \vec{u}) = M_{\mathtt{C}\text{-}\tau_i}(s', \vec{u})$.

Case 1: Assume $M_{\mathbf{C}-\tau_i}(s,\vec{u}) = M_{\mathbf{C}-\tau_i}(s',\vec{u}) = false$[†]. Because of the equation $A^{\perp}_{\tau_i}$, $M_{\tau_i}(s,\vec{u}) = s$ and $M_{\tau_i}(s',\vec{u}) = s'$. Thus, $M_{o_j}(M_{\tau_i}(s,\vec{u}),\vec{v}) = M_{o_j}(s,\vec{v}) = M_{o_j}(s',\vec{v}) = M_{o_j}(M_{\tau_i}(s',\vec{u}),\vec{v})$. The second equation comes from $s \sim s'$.
Case 2: Assume $M_{\mathbf{C}-\tau_i}(s,\vec{u}) = M_{\mathbf{C}-\tau_i}(s',\vec{u}) = true$. From the equation $A^{o_j}_{\tau_i}$, $M_{o_j}(M_{\tau_i}(s,\vec{u}),\vec{v}) = a(o_j(\tau_i(\mathrm{S},\mathrm{Y}_i),\mathrm{X}_j)) = a(ro_j\tau_i)$ and $M_{o_j}(M_{\tau_i}(s',\vec{u}),\vec{v}) = a'(ro_j\tau_i)$. The equation $a(ro_j\tau_i) = a'(ro_j\tau_i)$ holds from the same reason with the above equality $a(cr\tau_i) = a'(cr\tau_i)$. □

Since the observational equivalence is an equivalence relation in each model M, we can define an equivalence class of states as $\overline{s} = \{s' \in M_H \mid s \sim s'\}$. The set of the equivalent classes is denoted by $\overline{M_H}$. For a sequence of visible elements $\vec{v}$, we can define the function $M_{o_i\vec{v}} : M_H \to M_{V_{o_i}}$ as $\overline{M_{o_i\vec{v}}}(x) = M_{o_i}(x,\vec{v})$. $M_{\tau_i\vec{u}}$ is defined similiarly. A transition $\overline{M_{\tau_i\vec{a}}}$ on $\overline{M_H}$ can be defined straightforwardly. We define the state machine $\mathcal{S}_M$ as $(\overline{M_H}, \overline{M_{\mathtt{init}}}, T)$ where $T = \{\overline{M_{\tau_i\vec{u}}} \mid \tau_i \in OTS, u_j \in M_{V_{\tau_i j}}\}$. To avoid complex notations, we often use a notation x instead of M_x for some model M, where x is a sort, a constant, an operation and so on, and $o(\vec{v})$ and $\tau(\vec{u})$ instead of $\overline{M_{o\vec{a}}}$ and $\overline{M_{\tau\vec{a}}}$.

Example 3.3: Semaphore is a mechanism for restricting access to shared resources to a fixed number of processes at a time. We call the place where a process can use the shared resources the critical section. Semaphore has an integer variable x. To enter the critical section, a process p executes P operation: When $x > 0$, p enters the critical section and x is decreased. When $x \leq 0$, nothing happens. To leave the critical section, p executes V operation: It makes p leave and x increased. We give an OTS/CafeOBJ specification of Semaphore. The built-in module `INT` specifies integers with usual operations, like +, -, >, etc. The following module `PROCESS` specifies process identifiers.

```
mod* PROCESS { [Pid] }
```

Since `PROCESS` is loose and has only one sort, no operation symbols and no equations, any set can be a denotational model of `PROCESS`. A main module `SEMAPHORE` specifies Semaphore.

```
mod* SEMAPHORE {
 pr(INT)
 pr(PROCESS)

*[Sys]*
op init : -> Sys
bop using : Sys Pid -> Bool
bop semaphore : Sys -> Int
bops down up : Sys Pid -> Sys
ops c-down c-up : Sys Pid -> Bool

var S : Sys
var X11 : Pid
var Y11 : Pid
var Y21 : Pid

eq using(init,X11) = false .

eq semaphore(init) = 1 .

ceq using(down(S,Y11),X11) = (if X11 == Y11
    then true else using(S,X11) fi)
   if c-down(S,Y11) .
ceq semaphore(down(S,Y11)) =
   semaphore(S) - 1
   if c-down(S,Y11) .
bceq down(S,Y11) = S if not(c-down(S,Y11)) .
eq c-down(S,Y11) = not using(S,Y11) and
   semaphore(S) > 0 .

ceq using(up(S,Y21),X11) = (if X11 == Y21
    then false else using(S,X11) fi)
   if c-up(S,Y21) .
ceq semaphore(up(S,Y21)) = semaphore(S) + 1
   if c-up(S, Y21) .
bceq up(S,Y21) = S if not(c-up(S,Y21)) .
eq c-up(S,Y21) = using(S,Y21) .
}
```

Submodules `INT` and `PROCESS` are imported. A sort `Sys` stands for a state space. Observations identify a system state. An observation `using(s,p)` checks whether Process `p` is using the shared resource or not at the state `s`. An observation `semaphore(s)` is the value of Semaphore variable. At the initial state `init`, no process is using the shared resource and the semaphore value is 0 (See the first two equations). P and V operations are denoted by `down` and `up`. The states `down(s,p)` and `up(s,p)` stand for the result states after applying `down` and `up` to `s` respectively. For example, the state `down(down(init, p0), p1)` is the state after the following executions: `p0` executes P operation at the initial state, then `p1` executes P operation. We can observe the state via obsevations, like `using(down(down(init, p0), p1), p1)`. The CafeOBJ system reduces this term into `false : Bool`, which means that the observed value by `using(_,p1)` for that state is false, i.e. the try to enter the critical section by `p1` has been failed. When the initial semaphore value is n, n processes can enter. If `semaphore(init) = 2`, the above `p1`'s try succeeds, and CafeOBJ returns `true : Bool`. □

3.3 State Property

A state property is given by a Boolean term. In this paper, we express a state property as a Boolean term t which satisfies that (i) t has exactly one hidden variable, (ii) the sort of each variable in t should be involved in the arity of an observation symbol, and (iii) no transition symbol occurs. Thus, the hidden variable `S` should occur directly under observation symbols, like $o_i(\mathrm{S},\ldots)$. For `SEMAPHORE`, only a variable

[†]Any CafeOBJ specification implicitly imports the built-in module `BOOL`. The denotational model of `BOOL` is Boolean algebra $(\{true, false\}, \{\wedge, \vee, \neg, \ldots\})$. `BOOL` is imported with the protect mode. Thus, any denotational model of any CafeOBJ specification must have Boolean algebra itself. Hence, for example, $a = true \vee a = false$ is true for any Boolean constant a.

of the hidden sort and variables of `Pid` can be involved in a state property, since `Pid` is in the arity of `using`. The transition symbols `down` and `up` are not allowed to occur in a state property. The following is an example of state properties:

```
using(S, P) and using(S, Q) implies P == Q
```

which means that if processes P and Q are using the resource then they should be identical, that is, at most one process is using the resource. Operation symbols `and`, `implies`, == are involved in the built-in module `BOOL`. Let SP be an OTS/CafeOBJ specification, and P a state property whose non-hidden variables are of sort $\vec{S}$. For a denotational model M of SP, P is interpreted into a predicate M_P which takes elements including a state and returns true or false. We write $SP \models Inv_P$ if $\mathcal{S}_M \models Inv_Q$ for any denotational model M of SP and the state property Q on $\mathcal{S}_M$ defined as $Q(\overline{s}) \stackrel{\text{def}}{\Longleftrightarrow} \forall \vec{t} \in M_{S_1} \times \cdots \times M_{S_x}.M_P(s, \vec{t})$.

3.4 Maude Rewrite Specification

In Maude, besides equations, rewrite rules can be declared in a rewrite specification, also called a system specification in Maude manual. A rewrite rule has the following syntax: `crl [`*Label*`]` l `=>` r `if` C . The condition part can be omitted if it is `true`. In that case, the rewrite rule is written with `rl`. The following is an example of rewrite specifications:

```
mod NNAT is
  inc NAT+ .
  sort NNat .  subsort Nat < NNat .
  op _|_ : NNat NNat -> NNat [assoc] .
  vars M N : Nat .
  rl [L] : N | M => N .
  rl [R] : N | M => M .
endm
```

For a Maude rewrite specification SP, the rewrite relation $\rightarrow_{SP}$ is defined as follow: $s \rightarrow_{SP} t$ if and only if (1) s has a subterm s', (2) $s' =_E l\theta$ for a rewrite rule l `=>` r `if` $C \in S$ and a ground substitution θ, (3) $C\theta =_E$ `true`, (4) t' is the result term of replacing s' into $r\theta$, and (5) $t =_E t'$, where a ground substitution θ is a map from variables to ground terms and $t\theta$ is the result term with replacing all variable x with $\theta(x)$. The binary relation $=_E$ is the congruent relation on the set E of all equations in SP. Hereafter we sometimes omit the subscript E from $=_E$. E.g. 1+1 | 2+2 | 3+3 $\rightarrow_{\texttt{NNAT}}$ 4 | 6 $\rightarrow_{\texttt{NNAT}}$ 2. We can say a Maude rewrite specification SP with a sort `State` and a term `init` of `State` specifies a state machine $(T_{\texttt{State}}, \{\texttt{init}\}, \rightarrow_{\texttt{SP}})$ where T is the term algebra of SP.

4. Definition of Translation System

We define a translation system from OTS/CafeOBJ specifications to Maude specifications. We call a Maude specification in the range of our translation system an OTS/Maude specification.

4.1 OTS/Maude Specification

We describe an OTS in Maude in the following way. An observed value is represented by a term of the sort `OValue`. Each `OValue` term has the syntax: `(`o`[`$\vec{s}$`]:`t`)`, where o is a name of the observation with parameters, $\vec{s}$ are terms for the parameters and t is the observed value. Only t may be changed by an application of a transition. The following Maude module `STATE` specifies states.

```
mod STATE is
  sorts OValue State .
  subsorts OValue < State .
  op __ : State State -> State [assoc comm] .
  op init : -> State .
endm
```

A snapshot of a state is represented by a term of `State`, which is a (multi)set of `OValue` terms: $O_1\ O_2 \cdots O_n$. A state `init` is prepared for an initial state. A state is identified by a set of observed values. Each rewrite rule has the form: `crl [`*Label*`]` $O_{i_1} \cdots O_{i_k}$ `=>` $O'_{i_1} \cdots O'_{i_k}$ `if` C, where O_j and O'_j are `OValue` terms and only their difference is its observed value, like $O_j =$`(o[`$\vec{s}$`]:`t`)` and $O'_j =$`(o[`$\vec{s}$`]:`t'`)`.

4.2 Refinement of Loose Specification

Since an OTS/CafeOBJ specification may have loose submodules and Maude does not deal with them, we need to translate loose modules to tight ones. Moreover Maude requires data modules to be complete.

Definition 4.1: Let SP be a loose CafeOBJ module and SP' a tight CafeOBJ module. SP' is a refinement module of SP if and only if (1) SP' is tight and complete, (2) $S_{SP} \subseteq S_{SP'}$, $\leq_{SP} \subseteq \leq_{SP'}$ and $\Sigma_{SP} \subseteq \Sigma_{SP'}$, (3) for each equation $l = r$ (or $l = r$ `if` c) in SP and ground substitution θ from variables to ground terms of SP', $l\theta = r\theta$ can be deduced from equations in SP', i.e. $l\theta =_{E'} r\theta$ (whenever $c\theta =_{E'}$ `true`). □

Since SP' has all operation symbols of SP, each state property P of SP is also a state property for a refinement module SP'.

Example 4.2: Consider the following module:

```
mod* COMM+{
  [Nat]
  op _+_ : Nat Nat -> Nat
  vars M N : Nat
  eq M + N = N + M .
}
```

`NAT+` is a refinement module of `COMM+` since `NAT+` is complete and + is commutative for any ground terms of `Nat`, i.e. $\forall m, n \in \mathcal{N}$. `s`^{m}`(0) + s`^{n}`(0) = s`^{n}`(0) + s`^{m}`(0)` can be deduced from the equations of `NAT+`. We omit a proof. □

Example 4.3: The following module is a refinement module of `mod* PROCESS{[Pid]}`:

```
mod! PROCESS{
 pr(INT)
 [Pid]
 op p : Nat -> Pid
}
```

The imported module INT is a built-in module of CafeOBJ (and Maude). We assume built-in modules are complete. The tight PROCESS is complete since it has no equation. □

In our translation system, we use same module name for an original and a refinement modules.

4.3 Specification Translation

We give a translation from an OTS/CafeOBJ specification whose submodules are tight and complete into an OTS/Maude specification. Precisely we give a function F which takes an OTS/CafeOBJ specification and returns an OTS/Maude specification. We write X' Instead of $F(X)$. For example, we write $Signature' = \overrightarrow{Obs'}\ \overrightarrow{Trans'}$ instead of $F(Signature) = F(Obs_1)\ F(Obs_2)\cdots F(Trans_1)\ F(Trans_1)\cdots$, which means that a signature of the translated OTS/Maude specification $F(OTS) = OTS'$ consists of symbols which are obtained by applying F to observation and transition symbols of the input OTS.

For an OTS/CafeOBJ specification OTS, the translated OTS/Maude specification OTS' is defined as follows:

$$
\begin{aligned}
OTS' &= \texttt{mod STATE}\cdots\texttt{endm}\\
&\quad\ \overrightarrow{SubModule'}\ MainModule'\\
SubModule'_i &= \texttt{fmod}\ S\ M_i\cdots\texttt{endfm}\\
MainModule' &= \texttt{mod}\ M\ \texttt{is}\\
&\qquad\qquad \texttt{inc STATE.}\\
&\qquad\qquad \texttt{pr}\ S\ M_1.\cdots\texttt{pr}\ S\ M_l.\\
&\qquad\qquad Signature'\\
&\qquad\qquad Axiom'\\
&\qquad\quad \texttt{endm}\\
Signature' &= \overrightarrow{Obs'}\ \overrightarrow{Trans'}\\
Obs'_i &= \texttt{op}\ (o_i[\ldots]:\_)\\
&\qquad : \overrightarrow{V_{o_i}}\ V_{o_i}\texttt{-> OValue}\ .\\
Trans'_i &= \texttt{op}\ \texttt{c-}\tau_i : \overrightarrow{V_{\tau_i}}\texttt{-> Bool}\ .
\end{aligned}
$$

If $\overrightarrow{V_{o_i}}$ is empty, the square bracket "[...]" is omitted in Obs'_i. This is a signature part of the translation system. Note that we do not need transition symbols in OTS/Maude specifications. A rewrite rule itself denotes a transition. The name of a transition symbol in OTS appears as the label of a rewrite rule in OTS'.

Example 4.4: The following is the first half of the translated SEMAPHORE:

```
mod SEMAPHORE is
  inc STATE .  pr INT .  pr PROCESS .
  op (using[_]:_)  : Pid Bool -> OValue .
  op (semaphore:_) : Int -> OValue .
```

□

Next, we give an axiom part of our translation system. In a model of an OTS/CafeOBJ specification, infinitely many observations and transitions may exist even if those symbols are finite since the number of elements for parameters may be infinite. e.g. $M_{\texttt{down(p 1)}}, M_{\texttt{down(p 2)}}, \ldots, M_{\texttt{down(p n)}}, \ldots$. In order to obtain a finite OTS/Maude specification, we need to choose a suitable finite set for each parameter sort. For the sort V_{ok} (or $V_{\tau k}$) of the k-th parameter of each observation symbol o (or transition symbol τ), we give a finite subset $FT_{V_{ok}} \subseteq NF_{V_{ok}}$ (or $FT_{V_{\tau k}} \subseteq NF_{V_{\tau k}}$) of ground normal forms. We write FT_{V_τ} instead of the sequence $FT_{V_{\tau 1}} \times FT_{V_{\tau 2}} \times \ldots \times FT_{V_{\tau m}}$ where m is the number of parameters of τ. Observation and transition symbols may share sorts, like o : H Nat -> Bool and τ : H Nat -> H. For such cases, we give one $FT_{\texttt{Nat}}$ for both parameters, i.e. if $V_{xi} = V_{yj}$ then $FT_{V_{xi}} = FT_{V_{yj}}$ ($x, y = o$ or τ). Hereafter, O is the set of all observation symbols of OTS, i.e., $o_i \in O$ if and only if there exists bop o_i :$\cdots$ in the signature part of the input OTS. To define $Axiom'$, we define each component first, and then we give a definition of $Axiom'$ lastly. For given FTs, $Vars'$ is defined as follows:

$$
\begin{aligned}
Vars' &= Vars'_1\ Vars'_2\ \cdots\ Vars'_m\\
Vars'_j &= \{\texttt{var}\ \mathtt{Z}_{o_j(\vec{s})}\ \texttt{:}\ V_{o_j}\ \texttt{.}\ \mid \vec{s} \in FT_{V_{o_j}}\}
\end{aligned}
$$

For a transition symbol τ_i of a given OTS/CafeOBJ specification, the rewrite rule $A'_{\tau_i(\vec{t})}$ is defined as follow:

$$
\begin{aligned}
A'_{\tau_i} &= \{A'_{\tau_i(\vec{t})} \mid \vec{t} \in FT_{V_{\tau_i}}\}\\
A'_{\tau_i(\vec{t})} &= \texttt{crl}\ \ [\tau_i(\vec{t})]\ \ \texttt{:}L^{o_1}_{\tau_i(\vec{t})}\cdots L^{o_m}_{\tau_i(\vec{t})}\\
&\qquad \texttt{=>}\ R^{o_1}_{\tau_i(\vec{t})}\cdots R^{o_m}_{\tau_i(\vec{t})}\ \texttt{if}\ cr\tau_i^{\vec{t}'}\ \texttt{.}\\
L^{o_j}_{\tau_i(\vec{t})} &= \{(o_j[\vec{s}]\texttt{:}\mathtt{Z}_{o_j(\vec{s})}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}\\
R^{o_j}_{\tau_i(\vec{t})} &= \{(o_j[\vec{s}]\texttt{:}ro_j^{\vec{s}}\tau_i^{\vec{t}'}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}\\
ro_j^{\vec{s}}\tau_i^{\vec{t}'} &= ro_j^{\vec{s}}\tau_i^{\vec{t}}\\
&\qquad [o_j(\mathtt{S}, \vec{s}) \leftarrow \mathtt{Z}_{o_j(\vec{s}_\downarrow)} \mid o_j \in O, \vec{s}_\downarrow \in FT_{V_{o_j}}]\\
ro_j^{\vec{s}}\tau_i^{\vec{t}} &= ro_j\tau_i\left[X_j \leftarrow \vec{s}, Y_i \leftarrow \vec{t}\right]\\
cr\tau_i^{\vec{t}'} &= cr\tau_i^{\vec{t}}\\
&\qquad [o_j(\mathtt{S}, \vec{s}) \leftarrow \mathtt{Z}_{o_j(\vec{s}_\downarrow)} \mid o_j \in O, \vec{s}_\downarrow \in FT_{V_{o_j}}]\\
cr\tau_i^{\vec{t}} &= cr\tau_i\left[Y_i \leftarrow \vec{t}\right]
\end{aligned}
$$

where $\vec{u}_\downarrow$ stands for the sequence of the normal forms of terms u_i. $s[\vec{t} \leftarrow \vec{u}]$ or $s[t_1 \leftarrow u_1, \ldots, t_n \leftarrow u_n]$ stands for the result term of replacing all occurrences of $\vec{t}$ in s into $\vec{u}$[†].

[†]In general such replacement is not well-defined since a pattern t may overlap, e.g. $\mathtt{s(s(0))[s}(t) \leftarrow \mathtt{0}]$ can be both 0 and s(0). In our case it is well-defined since observation symbols are not nested.

Note that a variable name in a label is not a variable but we let them be instantiated by this definition. The following is $A'_{\texttt{down(p(0))}}$:[†]

```
crl [down_p_0] :
  (using[p(0)]: Zp0)  (using[p(1)]: Zp1)
  (semaphore: Zs)
=>
  (using[p(0)]:
    (if p(0) == p(0) then true else Zu0 fi))
  (using[p(1)]:
    (if p(1) == p(0) then true else Zu1 fi))
  (semaphore: (Zs - 1))
if
  not Zu0 and Zs > 0 .
```

Unfortunately, even if we instantiate all parameter variables, some terms may not be well-formed. For example, if there exists a subterm `o(S, I+1)` in the right-hand side of an equation in the original OTS/CafeOBJ specification, where `I` is a variable of `Int`. For any $FT_{\texttt{Int}}$, there exists the maximal integer $m \in FT_{\texttt{Int}}$. Thus, the variable $\texttt{Z}_{\texttt{o}(m+1)}$ for `o(S,` m`+1)` does not included in $Vars'$. To obtain a feasible Maude specification, we need remove such rewrite rules. We let *Filter* be a function which takes a set of rewrite rules translated by A'_τ and returns the set of all well-formed and feasible (no extra variables) rewrite rules.

Next, we give an rewrite rule for the initial state.

$$A'_{init} = \texttt{eq init =} R^{o_1}_{init} \cdots R^{o_m}_{init}$$
$$R^{o_j}_{init} = \{(o_j[\vec{s}] : rinit^{\vec{s}}_j) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}$$
$$rinit^{\vec{s}}_j = rinit_j\left[X_j \leftarrow \vec{s}\right]$$

Finally we define $Axiom'$ as follow:

$$Axiom' = Vars' \cup A'_{init} \cup Filter(A'_{\tau_1} \cup \cdots \cup A'_{\tau_n})$$

Example 4.5: When $FT_{\texttt{Pid}} = \{\texttt{p(0)}, \texttt{p(1)}\}$. The latter half of Maude `SEMAPHORE` is obtained as follows:

```
mod SEMAPHORE is
 ...
var Zu0 : Bool . var Zu1 : Bool .
var Zs : Int .

eq init = (using[p(0)]: false)
          (using[p(1)]: false)
          (semaphore: 1) .

crl [down_p_0] :
 (using[p(0)]: Zp0) (using[p(1)]: Zp1) ...
=>
 (using[p(0)]: (if p(0) == p(0) then ...))
 (using[p(1)]: (if p(1) == p(0) then ...))
 (semaphore: (Zs - 1))
if
 not Zu0 and Zs > 0 .

crl [down_p_1] : ..
crl [up_p_0] : ...
crl [up_p_1] : ...
endm
```

□

4.4 Property Translation

For an OTS/Maude specification SP', a state property P' is defined as a `Bool` term which includes only $\texttt{Z}_{o_j(\vec{s})}$s (for $j \in \{1, \ldots, m\}$ and $\vec{s} \in FT_{o_j}$) as variables. We write $SP' \models Inv_{P'}$ if and only if $P'[\texttt{Z}_{o_j(\vec{s})} \leftarrow t_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] = \texttt{true}$ for any state $\{(o_j[\vec{s}] : t_{o_j(\vec{s})}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}$ reachable from `init` by applying rewrite rules.

We define the translated state property P' from a state property P of an OTS/CafeOBJ specification as follows:

$$P' = \begin{cases} Q_1 \texttt{ and} \cdots \texttt{and } Q_k & \text{if all } Q_i \text{ are well-formed} \\ \texttt{true} & \text{o.w.} \end{cases}$$
$$Q_i = R_i\left[o_j(\texttt{S}, \vec{s}) \leftarrow \texttt{Z}_{o_j(\vec{s}_\downarrow)} \mid o_j \in O, \vec{s}_\downarrow \in FT_{V_{o_j}}\right]$$
$$\vec{R} = \left\{P[\vec{W} \leftarrow \vec{w}] \mid w_i \in FT_{S_i}, i = 1, 2, \ldots\right\}$$

where $\{\vec{W}\}$ are the set of all variables except the hidden variable `S` in P, and S_i is the sort of W_i. Thus, R_i is an instance of P w.r.t. FTs. $\vec{R}$ is the set of all instances of P w.r.t. FTs. Q_i is the result of replacing all occurrences of $o(\texttt{S}, \vec{s})$ with the corresponding variables $\texttt{Z}_{o(\vec{s})}$. From the same reason of the specification translation, Q_i may not be well-formed, i.e. there may be $o(\texttt{S}, \vec{s})$ in P such that $s_{i\downarrow}$ is not in FT_S.

Example 4.6: For $FT_{\texttt{Pid}} = \{\texttt{p(0)}, \texttt{p(1)}\}$, `using(S,P)` and `using(S,Q)` `implies` `P == Q` is translated into

```
      Zu0 and Zu0 implies p(0) == p(0)
and   Zu0 and Zu1 implies p(0) == p(1)
and   Zu1 and Zu0 implies p(1) == p(0)
and   Zu1 and Zu1 implies p(1) == p(1)
```

For the translated specification and property we can apply search command to prove the state property invariant as follows:

```
search in SEMAPHORE :
init =>* (using[p(0)]:Zu0)(using[p(1)]:Zu1)
(semaphore:Zs)
such that
not((Zu0 and Zu0 implies p(0) == p(0)) and
    (Zu0 and Zu1 implies p(0) == p(1)) and
    (Zu1 and Zu0 implies p(1) == p(0)) and
    (Zu1 and Zu1 implies p(1) == p(1)) ) .
```

The Maude system returns `no solution` for this input, which means the translated state property is invariant for the translated OTS/Maude specification `SEMAPHORE`.

[†]Because of the limitation of Maude syntax, we cannot use the bracket symbols in a label and a variable name. Thus, instead of the bracket symbols we use the underbars in the translated specification. Moreover, we use a simple name of a variable, e.g. the full version of `Zu0` is `Zu_p_0`.

If we rewrite the initial value of Semaphore into 2, i.e. `init = (using[p(0)]:false) (using[p(1)]:false) (semaphore:2)`. Maude system returns the counter-example as follows:

```
Solution 1 (state 3)
states: 4  rewrites: 162 ...
Zs --> 0
Zu0 --> true
Zu1 --> true
```

A path of the counter-example can be obtained as follows:

```
Maude> show path labels 3 .
down_P0
down_P1
```

□

5. Completeness

In this section we show that the existence of a counter-example in the translated specification implies the existence of a counter-example in the original one. That is equivalent to the completeness of the translation. A translation is complete w.r.t. a invariant property if and only if $SP' \models Inv_{P'}$ whenever $SP \models Inv_P$. A translation is sound if and only if $SP \models Inv_P$ whenever $SP' \models Inv_{P'}$. If a translation is complete and there is a counter-example in the translated specification, then the original specification should also have a counter-example.

Lemma 5.1: Let SP be an OTS/CafeOBJ specification. Let SP' be a refinement specification of SP. Let P be a state property for SP. If $SP \models Inv_P$ then $SP' \models Inv_P$.

Proof. It is trivial since any denotational model of SP' is also a denotational mode of SP. □

Lemma 5.2: Let SP be an OTS/CafeOBJ specification whose submodules are tight and complete, and SP' an OTS/Maude specification obtained by our translation. Assume that

- **(i)** $\{(o_j[\vec{s}]:u_{o_j(\vec{s})}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}$ is rewritten into $\{(o_j[\vec{s}]:v_{o_j(\vec{s})}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}$ by the rewrite rule labeled by $\tau_i(\vec{t})$ in SP' where $u\_$ and $v\_$ are ground normal forms of the submodules, and
- **(ii)** each $u_{o_j(\vec{s})}$ is the normal form of $o_j(s, \vec{s})$ for a ground term s of the sort H in SP.

Then, $o_j(\tau_i(s, \vec{t}), \vec{s}) = v_{o_j(\vec{s})}$ for each $o_j \in O$ and $\vec{s} \in FT_{V_{o_j}}$.

Proof. To prove $o_j(\tau_i(s, \vec{t}), \vec{s}) = v_{o_j(\vec{s})}$, we show that a conditional equation `ceq` $o_j(\tau_i(S, Y_i), X_j) = ro_j\tau_i$ `if` $\mathtt{c}\text{-}\tau_i(S, Y_i)$ in SP can be applied to obtain the equation, that means that for a ground substitution $\theta = [S, X_j, Y_i \leftarrow s, \vec{s}, \vec{t}]$, we show $\mathtt{c}\text{-}\tau_i(S, Y_i)\theta = \mathtt{true}$ and $ro_j\tau_i\theta = v_{o_j(\vec{s})}$.

From the assumption **(i)**, the condition of the rewrite rule is true, that is, we have $cr\vec{\tau_i'}\sigma = \mathtt{true}$ for a ground substitution $\sigma = [Z_{o_j(\vec{s})} \leftarrow u_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}]$ in SP' (and also in SP since they have common submodules). From the definition,

$$
\begin{aligned}
& cr\vec{\tau_i'}\sigma \\
&= cr\vec{\tau_i}[o_j(S, \vec{s}) \leftarrow Z_{o_j(\vec{s}_\downarrow)} \mid o_j \in O, \vec{s}_\downarrow \in FT_{V_{o_j}}]\sigma \\
&= cr\vec{\tau_i}[o_j(S, \vec{s}) \leftarrow u_{o_j(\vec{s}_\downarrow)} \mid o_j \in O, \vec{s}_\downarrow \in FT_{V_{o_j}}] \\
&\quad \cdots (\mathbf{a})
\end{aligned}
$$

Note that $\vec{u}_\downarrow \in FT_{V_{o_j}}$. From the assumption **(ii)**, we have $o_j(s, \vec{s}) = u_{o_j(\vec{s})}$ in SP. Then, $\mathtt{c}\text{-}\tau_i(S, Y_i)\theta = \mathtt{c}\text{-}\tau_i(s, \vec{t}) = cr\vec{\tau_i}[S \leftarrow s]$ and

$$
\begin{aligned}
& cr\vec{\tau_i}[S \leftarrow s] \\
&= cr\vec{\tau_i}[S \leftarrow s] \\
&\qquad [o_j(s, \vec{s}) \leftarrow u_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&= cr\vec{\tau_i}[o_j(S, \vec{s}) \leftarrow o_j(s, \vec{s}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&\qquad [o_j(s, \vec{s}) \leftarrow u_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&= cr\vec{\tau_i}[o_j(S, \vec{s}) \leftarrow u_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&\quad \cdots (\mathbf{a}')
\end{aligned}
$$

The second equation holds since the state variable S should occur directly under some observation symbol. Since $s = s_\downarrow$ for any term s, (**a**) and (**a**′) are connected and $\mathtt{c}\text{-}\tau_i(S, Y_i)\theta = \mathtt{true}$ holds. In a similar say, $ro_j\tau_i\theta = ro_j^{\vec{s}}\vec{\tau_i}[o_j(S, \vec{s}) \leftarrow u_{o_j(\vec{s}_\downarrow)}] = ro_j^{\vec{s}}\vec{\tau_i'}\sigma = v_{o_j(\vec{s})}$ also holds. By applying `ceq` $o_j(\tau_i(S, Y_i), X_j) = ro_j\tau_i$ `if` $\mathtt{c}\text{-}\tau_i(S, Y_i)$ in SP, we obtain $o_j(\tau_i(s, \vec{t}), \vec{s}) = o_j(\tau_i(S, Y_i), X_j)\theta = ro_j\tau_i\theta = v_{o_j(\vec{s})}$. □

The following theorem is about a kind of completeness of our translation system. The theorem says that if a state property is invariant on an OTS/CafeOBJ specification the translated state property is also invariant on the translated OTS/Maude specification.

Theorem 5.3: Let SP be an OTS/CafeOBJ specification and P a state property on SP. Assume SP' and P' are translated from SP and P. Then, $SP' \models Inv_{P'}$ whenever $SP \models Inv_P$.

Proof. Assume $SP' \not\models Inv_{P'}$. We show $SP \not\models Inv_P$. Assume SP_r is an intermediate refinement specification from SP to SP'. There exists a conter-example state $\{(o_j[\vec{s}] : t_{o_j(\vec{s})}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}$ reachable from `init` such that $P'[\mathtt{Z}_{o_j(\vec{s})} \leftarrow t_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] = \mathtt{false}$. Assume the rewrite sequence from the initial state to the counter-example in SP' is created by applying the sequence of the rewrite rules $\tau_{i_0}(\vec{t_0}), \tau_{i_1}(\vec{t_1}), \ldots, \tau_{i_x}(\vec{t_x})$. By applying Lemma 5.2 for each rewrite we obtain a state t_x of SP_r satisfying that $t_0 = \mathtt{init}$, $t_{k+1} = \tau_{i_k}(t_k, \vec{t_k})$, $s = \tau_{i_x}(t_x, \vec{t_x})$ and $o_j(s, \vec{s}) = t_{o_j(\vec{s})}$[†]. From the definition in Sect. 4.4 (Property tanslation), we can write $P' = Q_1$ `and` Q_2 `and` $\cdots$ `and` Q_y. Thus, $P'[\mathtt{Z}_{o_j(\vec{s})} \leftarrow t_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] = \mathtt{false}$ implies

[†]In the case of the length of the rewrite sequence in SP' is 0, $s = \mathtt{init}$.

$Q_l[\mathtt{Z}_{o_j(\vec{s})} \leftarrow t_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] = \mathtt{false}$ for some l. Let $\vec{w}$ be the ground terms which instantiate P's variables in R_l. Then,

$$\begin{aligned}
&Q_l[\mathtt{Z}_{o_j(\vec{s})} \leftarrow t_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&\quad = R_l[o_j(S, \vec{s}) \leftarrow \mathtt{Z}_{o_j(\vec{s}_\downarrow)} \mid o_j \in O, \vec{s}_\downarrow \in FT_{V_{o_j}}] \\
&\qquad [\mathtt{Z}_{o_j(\vec{s})} \leftarrow t_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&\quad = R_l[o_j(S, \vec{s}) \leftarrow t_{o_j(\vec{s})} \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&\quad = R_l[o_j(S, \vec{s}) \leftarrow o_j(s, \vec{s}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}] \\
&\quad = R_l[S \leftarrow s] \\
&\quad = P[S, \vec{W} \leftarrow s, \vec{w}]
\end{aligned}$$

The first equation comes from the definition of Q_l in Sect. 4.4. The second equation is from the transitivity of replacements and $s_i = s_{i\downarrow}$ for each term s_i. The third equation comes from $o_j(s, \vec{s}) = t_{o_j(\vec{s})}$. The forth equation holds since S should occur under some o_j. The last equation comes from the definition of R_l. We obtain $P(s, \vec{w}) = \mathtt{false}$ in SP_r. For any denotational model M of SP_r, $P_M(M_s, \vec{M}_w) = false$, and $SP_r \not\models Inv_P$. From Lemma 5.1 $SP \not\models Inv_P$ holds. □

Example 5.4: From Theorem 5.3 we can say that the counter-example of the OTS/Maude specification SEMAPHORE with the initial value 2 in the previous section is also a counter-example of the original OTS/CafeOBJ specification SEMAPHORE. □

The soundness of our translation system (the inverse of Theorem 5.3) does not hold since a refinement OTS/CafeOBJ specification denotes a part of the denotational models of the original OTS/CafeOBJ specification, and a translated OTS/Maude rewrite specification does not cover all transition relation of the original specification.

6. Improvement by Simplification

One of the most difficult part of our translation is to choose suitable finite set of ground normal forms. We give a technique to reduce the task of instantiating parameter variables of transition operation symbols. We call it Simplification. First, we give an example in which a parameter variable of a transition symbol makes translation hard.

Example 6.1: We give another OTS/CafeOBJ specification of Semaphore.

```
mod! PSET {
pr(PROCESS) pr(INT)
[Pid < Pset]
op nil : -> Pset
op __ : Pset Pset -> Pset {assoc comm
                           idr: nil}
op rm : Pid Pset -> Pset
op ln : Pset -> Nat
vars P P' : Pid
var Ps : Pset
eq rm(P, nil) = nil .
eq rm(P, P') =
   if P == P' then nil else P' fi .
eq rm(P, P' Ps) =
   if P == P' then rm(P, Ps)
              else P' rm(P, Ps) fi .
eq ln(nil) = 0 .
eq ln(P) = 1 .
eq ln(P Ps) = 1 + ln(Ps) .
}
```

PSET specifies process sets with operation symbols rm which removes an element from a set and ln which returns the cardinality of a set. A set of processes P, Q and R is represented by the sequence P Q R.

```
mod* SEMAPHORE {
pr(PSET)
*[Sys]*
op init       : -> Sys
bop using     : Sys -> Pset
bop semaphore : Sys -> Int
bop down      : Sys Pid -> Sys
bop up        : Sys Pid Pset -> Sys
var S : Sys
vars Y11 Y21 : Pid
var Y22 : Pset
eq   using(init) = nil .
eq semaphore(init) = 1 .
ceq  using(down(S, Y11)) = Y11 using(S)
    if semaphore(S) > 0 .
ceq  semaphore(down(S, Y11)) =
     semaphore(S) - 1
    if semaphore(S) > 0 .
bceq down(S, Y11)   = S
    if not(semaphore(S) > 0) .
ceq  using(up(S, Y21, Y22)) =
     rm(Y21, using(S))
    if using(S) == Y21 Y22 .
ceq  semaphore(up(S,Y21,Y22)) =
     semaphore(S) + 1
    if using(S) == Y21 Y22 .
bceq up(S, Y21, Y22) = S
    if not(using(S) == Y21 Y22) .
}
```

The difference from the previous specifications is that (i) using is defined for the whole system and returns all processes which are using the share resource, (ii) up takes a process p and a processes set ps and changes the state only if $p \cup ps$ is the set of all using processes in the current state. Since up takes the set of processes, the number of interpreted functions is exponentially larger than that of SEMAPHORE in the previous sections. However, because of the guard condition using(S) == Y21 Y22 for the transition up(S,Y21,Y22), the transition up do nothing for a set unrelated to the current state. Thus, most cases are ignored. □

In order to give an improved translation system, we first

give a translated rewrite rule $A'_{\tau_i(Y_i)}$ in which all parameter variables of observation symbols are instantiated but those of the transition symbol τ_i are not.

$$A'_{\tau_i(Y_i)} = \texttt{crl}\ [\tau_i(Y_i)]\ : L^{o_1}_{\tau_i(Y_i)} \cdots L^{o_m}_{\tau_i(Y_i)} \Rightarrow R^{o_1}_{\tau_i(Y_i)} \cdots R^{o_m}_{\tau_i(Y_i)}\ \texttt{if}\ cr\tau'_i\ .$$

$$L^{o_j}_{\tau_i(Y_i)} = \{(o_j[\vec{s}] : \mathsf{Z}_{o_j(\vec{s})}) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}$$

$$R^{o_j}_{\tau_i(Y_i)} = \{(o_j[\vec{s}] : ro^{\vec{s}}_j\tau'_i) \mid o_j \in O, \vec{s} \in FT_{V_{o_j}}\}$$

$$ro^{\vec{s}}_j\tau'_i = ro^{\vec{s}}_j\tau_i\left[o_j(\mathsf{S}, \vec{u}) \leftarrow Z_{o_j(\vec{u}_\downarrow)}\right]$$

$$ro^{\vec{s}}_j\tau_i = ro_j\tau_i\left[X_j \leftarrow \vec{s}\right]$$

$$cr\tau'_i = cr\tau_i\left[o_j(\mathsf{S}, \vec{u}) \leftarrow Z_{o_j(\vec{u}_\downarrow)}\right]$$

The following is the rewrite rule $A'_{\mathrm{up(P,Ps)}}$ of the translated OTS/Maude SEMAPHORE:

```
crl [up_P_Ps] : (using:Zu) (semaphore:Zs)
=> (using: rm(P,Zu)) (semaphore:Zs + 1)
if Zu == P Ps .
```

To obtain feasible rewrite rules, we have to remove extra variables P and Ps. However, since Ps is of the sort Pset, it is hard to find suitable finite set of ground normal forms.

Before instantiating extra variables, we try to remove them by applying the following simplification function. A simplification function *Simp*, which takes a rewrite rule and returns the simplified rewrite rule, is defined as the result of applying the following procedure to the input rewrite rule: Assume the input rewrite rule has the form: crl l => r if C_1 and C_2 and $\cdots$ and C_n.

```
i := 1;
while(i ≤ n){
  if (C_i = "Z_{o(s)} == t") {
    remove C_i from the conditional part, and
    replace all the other occurrences of Z_{o(s)}
    in the rewrite rule with t  };
  i := i + 1; }
```

Note that if all C_i are removed, if is also removed and crl is changed into rl. The following is an example of simplified rewrite rules:

$Simp(A'_{\mathrm{up(P,Ps)}})$ =

```
rl [up_P_Ps] : (using:P Ps) (semaphore:Zs)
=> (using: rm(P,Z)) (semaphore:Zs + 1) .
```

Variables P and Ps are not extra variables now. The new definition of A'_{τ_i} is the set of all instances in which only the extra variables in $Simp(A'_{\tau_i(Y_i)})$ are instantiated. The following is an example of OTS/Maude specification translated by our improved translation system:

```
mod SEMAPHORE is
inc STATE .  pr INT .  pr PSET .
op (using:_) : Pset -> OValue .
op (semaphore:_) : Int -> OValue .

var P : Pid .    var Ps : Pset .
var Zu : Pset .  var Zs : Int .

eq init = (using: nil)(semaphore: 1) .

crl[down_p_0] : (using: Zu) (semaphore: Zs)
 => (using: (p(0) Zu)) (semaphore: (Zs - 1))
if Zs > 0 .

crl[down_p_1] : (using: Zu) (semaphore: Zs)
 => (using: (p(1) Zu)) (semaphore: (Zs - 1))
if Zs > 0 .

rl[up_P_Ps] : (using: (P Ps)) (semaphore: Zs)
 => (using: rm(P, (P Ps)))
    (semaphore: (Zs + 1)) .
endm
```

A state property for the new OTS/CafeOBJ specification SEMAPHORE is written as P = ln(using(S)) < 2, which means that the number of the processes using the shared source is less than 2. The translated property is P' = ln(Zu) < 2. We can apply the Maude search command as follows:

```
search in SEMAPHORE :
  init =>* (using: Zu) (semaphore: Zs)
  such that not(ln(Zu) < 2) .
```

Then, no solution is returned. The following is the case that the initial Semaphore value is 2:

```
search in SEMAPHORE :
  (using: nil) (semaphore: 2)  =>*
  (using: Zu) (semaphore: Zs) such that
  not(ln(Zu) < 2) .
```

Maude returns three solutions. One of them is

```
Solution 2 (state 4)
Zu --> p(0) p(1)
Zs --> 0
```

which means P' is not invariant. The show path command returns the path of the counter-example, like

```
down_p_0
down_p_1.
```

7. Application

While describing a formal specification in an executable specification language, we may test (candidates of) a specification to find an error before doing formal verifications. Here a test stands for one example sequence of transitions, for example. Although a test may be useful to know what happen for an example of executions, we can check for only finite executions. The search command of the Maude system (or Maude model checker) may check possibly infinite cases. Surely our translation system may restrict the number of transitions to be finite, however, our translation plus

Maude may check possibly infinitely many executions. In our example of Semaphore, only two processes are considered in the translated specification, but executions made from them are infinite. Thus, our translation can be said to be more useful to find errors than tests.

In verification by CafeOBJ, we may need some lemma to verify an invariant property. A user should try to find suitable lemma to prove the main property. Our translation system may also be useful for that task. When a user thinks some property is a candidate of a lemma, the user translates it by our translation system, and applies Maude search command. If a counter-example is returned, the lemma is wrong. The user avoids waste of time to prove the wrong lemma and try to find next candidates. The literature [8] mentions a special lemma, called a necessary lemma, which has a property that if the lemma does not hold then the original theorem also does not hold. By combining the notion of the necessary lemma and our translation system, we can obtain a verification system which is strong in finding a counter-example.

8. Conclusion

We proposed a translation system from OTS/CafeOBJ to OTS/Maude, which is complete, and is useful in disproving a state property to be invariant. A specification translation between different formal specification languages benefits the research area of integrated formal methods. Chocolat/SMV [15] is a tool which translate CafeOBJ specification to SMV[†], which is one of the most famous model checker. In the literature [2], the use of SMV tool for reasoning about temporal properties of CommUnity designs is studied. CommUnity is a formal specification language based on Unity [3]. A translation from RAISE formal specification language [16], [17] to SAL[††], another famous model checker, is mentioned in [7]. An advantage of our translation system is that we can treat flexible user-definable abstract data types. In the above translations, data types which a system specification uses are restricted like a finite range of integers. Since CafeOBJ initial modules and Maude functional modules have essentially same syntax, our translation allows any abstract data type which can be given as a CafeOBJ initial module, for example, arrays, lists, several kinds of trees and so on.

A future work is to find a condition under which Maude search command terminates. A Maude search command does not always terminate in exchange for the expressive power of data types. One of the sufficient conditions is the termination of the rewrite relation. Termination is too restricted to specify practical systems. We are interested in non-terminating systems. Another future work is to find a condition under which our translation system is sound and complete. We are also interested in applying our translation system for properties other than the invariant property, like the liveness property and so on.

[†]http://www.cs.cmu.edu/~modelcheck/smv.html

[††]http://sal.csl.sri.com/

References

[1] TeReSe, Term Rewriting Systems, Cambridge Tracts in Theoretical Computer Science (no.55), 2003.

[2] N. Aguirre, G. Regis, and T. Maibaum, "Verifying temporal properties of commUnity designs," Proc. 6th International Conference on Integrated Formal Methods (IFM 2007), LNCS 4591, pp.1–20, Springer, 2007.

[3] K.M. Chandy and J. Misra, Parallel Program Design: A Foundation, Addison-Wesley Publishing, 1988.

[4] R. Diaconescu and K. Futatsugi, CafeOBJ report, World Scientific Publishing, 1998.

[5] K. Futatsugi, J.A. Goguen, J.P. Jouannaud, and J. Meseguer, "Principles of OBJ2," Proc. 12th ACM Symposium on Principles of Programming Languages, pp.52–66, 1985.

[6] J.A. Goguen, T. Winkler, J. Meseguer, K. Futatsugi, and J.P. Jouannaud, "Introducig OBJ," in Software Engineering with OBJ: algebraic specification in action, ed. J.A. Goguen and G. Malcolm, Kluwer, 2000.

[7] A.E. Haxthausen, C.W. George, and M. Schütz, "Specification and proof of the mondex electronic purse," Proc. 1st Asian Working Conference on Verified Software (AWCVS 2006), UNU-IIST Report no.347, pp.209–224, 2006.

[8] W. Kong, Facilitating Inductive Verication with Counterexample Discovery Capability, School of Information Science, Japan Advanced Institute of Science and Technology, PhD Thesis, Sept. 2006.

[9] M. Nakamura, W. Kong, K. Ogata, and K. Futatsugi, "A complete specification transformation from OTS/CafeOBJ to OTS/Maude," IEICE Technical Report, SS2006-13, June 2006.

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

[11] K. Ogata and K. Futatsugi, "Formal analysis of the iKP electronic payment protocols," Proc. 1st International Symposium on Software Security (1st ISSS), LNCS 2609 (Hot Topics, Software Security - Theories and Systems), pp.441–460, Springer, 2003.

[12] K. Ogata and K. Futatsugi, "Rewriting-based verification of authentication protocols," Proc. 4th International Workshop on Rewriting Logic and its Applications (4th WRLA), ENTCS 71, pp.189–203, Elsevier, 2004.

[13] K. Ogata and K. Futatsugi, "Equational approach to formal verification of SET," Proc. 4th International Conference on Quality Software (4th QSIC), pp.50–59, IEEE Computer Society Press, 2004.

[14] K. Ogata and K. Futatsugi, "Formal analysis of the NetBill electronic commerce protocol," Proc. 2nd International Symposium on Software Security (2nd ISSS), LNCS 3233, pp.45–64, Springer, 2004.

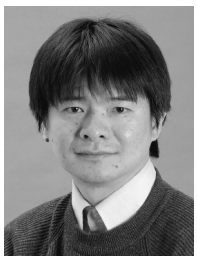
[15] K. Ogata, M. Nakano, M. Nakamura, and K. Futatsugi, "Chocolat/SMV: A translator from CafeOBJ into SMV," Proc. 6th International Conference on Parallel and Distributed Computing, Applications and Technologies (6th PDCAT), pp.416–420, IEEE Computer Society Press, 2005.

[16] The RAISE Language Group, The RAISE Specification Language, BCS Practitioner Series, Prentice Hall, 1992.

[17] The RAISE Method Group, The RAISE Development Method, BCS Practitioner Series, Prentice Hall, 1995.

Masaki Nakamura is an assistant professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his Ph.D. in information science from JAIST in 2002. His research interest includes software engineering, formal methods, algebraic specification and term rewriting.

Weiqiang Kong is presently a researcher of the 21st Century COE Program at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his Ph.D. in information science from JAIST in 2006. His current research interests include formal methods and their application, in particular, combination of theorem proving and (bounded) model checking for software design analysis.



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.