Modular Implementation of a Translator from Behavioral Specifications to Rewrite Theory Specifications (Extended Version)

Min Zhang, Kazuhiro Ogata

School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)

Registration Number: **IS-RR-2010-002**
Published Date: **2010-03-10**

Modular Implementation of a Translator from Behavioral Specifications to Rewrite Theory Specifications (Extended Version)

Min Zhang, Kazuhiro Ogata
School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
1-1 Asahidai, Nomi, Ishikawa 923-1292, Japan
{zhangmin, ogata}@jaist.ac.jp

Abstract

Specification translation plays an important part in the integration of theorem proving and model checking techniques for system verification. Much effort is required to implement a translation tool in conventional programming languages. Maude provides powerful meta-programming facilities that allow us to develop formal translation tools with less effort. In this paper, we present a modular implementation of a translator that is developed in Maude. The translator takes a behavioral specification and produces a rewrite theory specification. The implementation of the translator is modular so that multiple translation strategies can be modularized and embedded in the translator. Therefore, multiple styles of rewrite theory specifications can be generated for one behavioral specification.

1. Introduction

Specification translation plays an important part in the integration of theorem proving and model checking techniques for system verification [1], [2]. CafeOBJ is equipped with the capability of theorem proving by interactive equational reasoning [3], [4] and Maude [5] provides powerful model checking facilities such as an LTL model checker [6]. Both of them have their own strengths and weaknesses: (1) model checkers can verify automatically that systems enjoy properties, provided that systems should be modeled as state machines whose (reachable) state spaces are bounded; (2) model checkers can provide automatically counterexamples; (3) interactive theorem provers can verify if systems enjoy some properties even if state space of the system is unbounded, although computer-human interaction is needed; (4) interactive theorem provers can help humans understand systems more profoundly by revealing hidden facts (lemmas) of the systems. Many efforts have been made to integrate these two verification techniques so that we are able to find possible "bugs" of a system at the early stage of verifying it with theorem proving facility of CafeOBJ [7], [8].

Observational transition systems (OTSs) are used to model systems when CafeOBJ is used as an interactive theorem prover. CafeOBJ specifications of OTSs are a class of behavioral specifications [9], [10], which are called OTS/CafeOBJ specifications. On the other hand, Maude specifications of systems to be model checked are called rewrite theory specifications [5]. When we want to use both CafeOBJ and Maude to analyse a dynamic system, we need to write both an OTS/CafeOBJ specification and a rewrite theory specification of the system in CafeOBJ and Maude languages, respectively. Not only is it time-consuming to write multiple different specifications for one system by hand, but some significant differences between different specifications may arise. One possible way to solve the problem is to automatically translate one specification into the other [1], [8].

However, it requires much effort to develop translation tools in conventional programming languages like Java. Instead, Maude, as a formal meta-tool, provides powerful meta-programming facilities that allow us to develop formal tools for specification translation with less effort [11], [12]. Given a specification of two formalisms and a translation strategy, we can develop a translator with meta-programming in Maude with less effort than in a conventional programming language. Many applications have been developed with Maude meta-programming facilities for building execution environments for a range of languages and logics, such as Real-Time Maude [13] and an Inductive Theorem Prover (ITP) [14].

We have developed a translator with Maude meta-level facilities. The purpose of the translator is two-fold. On the one hand, it automatically translates OTS/CafeOBJ specifications into Maude rewrite theory ones[1], which integrates the CafeOBJ and Maude for system verification in a lightweight way; on the other hand, its implementation is *modular* which shows the advantages of Maude meta-programming, in the sense that the implementation is modularized and modules can be reused. Two different translation strategies are modularized and embedded in the current implementation of translator to generate two different styles of rewrite theory specifications, without changing any other modules. With rewrite theory specifications, we are able to model check a system in Maude without manually specifying the system once again. The translator is implemented completely in Maude language. There are only less than 1,500 lines of code for the translator. Compared to an existing translator *Cafe2Maude* [1], which is implemented in Java with 3,000 lines of code, it shows that meta-programming facilities of Maude alleviate burdens of developing formal translators for translations between different formalisms of systems. Moreover, *Cafe2Maude* generates only one style of rewrite theory specifications and it is not easy to extend the translator with other translation strategies. The modular implementation of the translator can be regarded as a method for developing formal translation tools with Maude

1. The translator is available at website http://www.jaist.ac.jp/~s0820005

meta-programming facilities.

The rest of this paper is organised as follows. Section 2 introduces OTS/CafeOBJ specifications and Maude rewrite specifications briefly. Section 3 describes how the translator is implemented in Maude. In section 4, a concrete example is presented to illustrate how the translator works on the top of Full Maude [15]. Section 5 mentions some related work. Some conclusions are drawn and directions for future work are mentioned in Section 6.

2. Algebraic Specifications for Dynamic Systems

2.1. OTS/CafeOBJ Specifications

Observation Transition System (OTS for abbreviation) is proposed to model a dynamic system [3]. Assume that there exists a universal state space called Υ. We further suppose that data types have been defined in advance, including equivalence between two data values d_1, d_2 denoted by $d_1 = d_2$. A system is modeled by observing quantities inside each state of Υ and how these quantities are changed by state transitions. An OTS $\mathcal{S}$ is a triple $< \mathcal{O}, \mathcal{I}, \mathcal{T} >$, where:

- $\mathcal{O}$: A finite set of observers. Each observer $o_{x_1:D_{o1},\ldots,x_m:D_{om}} : \Upsilon \rightarrow D_o$ is an indexed function with m indexes $x_1, \ldots, x_m$ of types $D_{o1}, \ldots, D_{om}$ respectively. Given an OTS $\mathcal{S}$ and two state $v_1, v_2 \in \Upsilon$, we say v_1 and v_2 are equivalent w.r.t $\mathcal{S}$ denoted by $v_1 =_{\mathcal{S}} v_2$, if and only if $\forall o_{x_1:D_{o1},\ldots,x_m:D_{om}} \in \mathcal{O}. \forall x_1 : D_{o1} \ldots \forall x_m : D_{om}. o_{x_1,\ldots,x_m}(v_1) = o_{x_1,\ldots,x_m}(v_2)$;
- $\mathcal{I}$: The set of initial states such that $I \subseteq \Upsilon$;
- $\mathcal{T}$: A finite set of transitions. Each transition $t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \rightarrow \Upsilon$ is an indexed function with n indexes of types $D_{t1}, \ldots, D_{tn}$ respectively, provided that $t_{y_1,\ldots,y_n}(v_1) =_{\mathcal{S}} t_{y_1,\ldots,y_n}(v_2)$ for each $[v] \in \Upsilon / =_{\mathcal{S}}$, each $v_1, v_2 \in [v]$ and each y_k of type D_{tk} for $k = 1, \ldots, n$. State $t_{y_1,\ldots,y_n}(v)$ is called a successor state of v w.r.t. $\mathcal{S}$. Each transition $t_{y_1,\ldots,y_n}$ has a condition in form $c\text{-}t_{y_1:D_{t1},\ldots,y_n:D_{tn}} : \Upsilon \rightarrow Bool$, which is called an *effective condition*. In the case that $c\text{-}t_{y_1:D_{t1},\ldots,y_n:D_{tn}}(v)$ does not hold, then $t_{y_1,\ldots,y_n}(v) =_{\mathcal{S}} v$.

An OTS can be specified in CafeOBJ straightforwardly. A specification of an OTS in CafeOBJ is called an OTS/CafeOBJ specification. In OTS/CafeOBJ specifications, there are two kinds of sorts, namely visible sorts and hidden sorts. Visible sorts are mainly used to denote abstract data types, while hidden sorts represent the universal state space Υ. A hidden sort `H` is declared in the form of `*[H]*`. In an OTS/CafeOBJ specification, an arbitrary initial state of the OTS $\mathcal{S}$ is represented by a hidden constant `init`, each observer $o_{x_1,\ldots,x_m}$ by an observation operator `o`, each transition $t_{y_1,\ldots,y_n}$ by an action operator `t` and an effective condition $c\text{-}t_{y_1,\ldots,y_n}$ by a conventional operator `c-t`. Hidden constant `init`, observation operator `o`, action operator `t` and operator `c-t` are declared as follows:

```
op init : -> H
bop o : H Vo1 ... Vom -> Vo
bop t : H Vt1 ... Vtn -> H
op c-t : H Vt1 ... Vtn -> Bool
```

Where, `H` is a hidden sort denoting Υ; each $\mathtt{V}_*$ is a visible sort denoting a data type D_*; `Bool` is a visible sort denoting truth values and $\mathtt{X}_k$ and $\mathtt{Y}_k$ are CafeOBJ variables corresponding to indexes x_k and y_k respectively. Keyword `bop` (or `bops`) is used to declare observation or action operator (or operators), while `op` (or `ops`) for conventional operator (or operators) or hidden constant (or hidden constants).

Assume that the value returned by observer $o_{x_1,\ldots,x_m}$ in the initial state is an expression $f(x_1, \ldots, x_m)$. It can be specified in CafeOBJ by the following equation:

$$\mathtt{eq\ o(init,X_1,\ldots,X_m)\ =\ f(X_1,\ldots,X_m)}. \quad (1)$$

Where, $\mathtt{f(X_1,\ldots,X_m)}$ is a CafeOBJ term corresponding to $f(x_1, \ldots, x_m)$. Equation (1) says that the value observed by observation operator `o` with variables $\mathtt{X}_1, \ldots, \mathtt{X}_m$ is the one that $\mathtt{f(X_1,\ldots,X_m)}$ returns.

Each transition $t_{y_1,\ldots,y_n}$ is defined by describing what the value returned by each observer $o_{x_1,\ldots,x_m}$ in the successor state is changed into, when $t_{y_1,\ldots,y_n}$ is applied to the state v and the effective condition $c\text{-}t_{y_1,\ldots,y_n}$ holds in state v. A conditional equation in the following form is defined to specify the transition w.r.t. $o_{x_1,\ldots,x_m}$:

$$\begin{aligned}&\mathtt{ceq\ o(t(S,Y_1,\ldots,Y_n),X_1,\ldots,X_m)\ =} \quad (2)\\ &\quad \mathtt{e\text{-}t(S,Y_1,\ldots,Y_n,X_1,\ldots,X_m)}\\ &\quad \mathtt{if\ c\text{-}t(S,Y_1,\ldots,Y_n)}\ .\end{aligned}$$

Where, `S` is a variable of `H` that represents the state v; term $\mathtt{t(S,Y_1,\ldots,Y_n)}$ represents the successor state of `S`; $\mathtt{e\text{-}t(S,Y_1,\ldots,Y_n,X_1,\ldots,X_m)}$ is a term that represents the value returned by $o_{x_1,\ldots,x_m}$ in the successor state and the condition part $\mathtt{c\text{-}t(S,Y_1,\ldots,Y_n)}$ represents the effective condition of the transition. For each effective condition, an equation is defined to check if the effective condition holds in the given state.

In the case that the effective condition $c\text{-}t_{y_1,\ldots,y_n}(v)$ always holds in any state v or the value returned by $o_{x_1,\ldots,x_m}$ is not changed in the successor state of any state v, the condition part can be omitted and an unconditional equation is defined for it as follows:

$$\begin{aligned}&\mathtt{eq\ o(t(S,Y_1,\ldots,Y_n),X_1,\ldots,X_m)\ =}\\ &\quad \mathtt{e\text{-}t(S,Y_1,\ldots,Y_n,X_1,\ldots,X_m)}\ .\end{aligned}$$

Where, $\mathtt{e\text{-}t(S,Y_1,\ldots,Y_n,X_1,\ldots,X_m)}$ is equal to $\mathtt{o(S,X_1,\ldots,X_m)}$ in the second case.

When $c\text{-}t_{y_1,\ldots,y_n}(v)$ does not hold in state v, values in the successor state of v are not changed by $t_{y_1,\ldots,y_n}$. A conditional equation is declared for it in the following form:

$$\begin{aligned}&\mathtt{bceq\ t(S,Y_1,\ldots,Y_n)\ =\ S}\\ &\quad \mathtt{if\ not\ c\text{-}t(S,Y_1,\ldots,Y_n)}\ .\end{aligned}$$

2.2. Rewrite Theory Specifications

Maude is also regarded as an algebraic specification language whose foundations are membership equational logic and rewriting logic. In Maude, basic units are modules. There are two types of modules, namely functional modules and system modules that represent membership equational theories and rewrite theories respectively. Functional modules admit equations, identifying data, and memberships, stating typing information for some data, while system modules also admit rewrite rules, describing transitions between states, besides equations and memberships [5]. Computationally, rewrite rules specify local concurrent transitions that can take place in a system if the pattern in the rule's left-hand side matches a fragment of the system state and the rule's condition is satisfied. In that case, the transition specified by the rule can take place and the matched fragment of the state is transformed into the corresponding instance of the right-hand side.

A specification of a dynamic system in Maude is a precise prototype to simulate its behaviors, which is called a *rewrite theory specification*. States can be represented both implicitly and explicitly. A state is called implicit in the sense that values of the state cannot be read directly, while an explicit state is a collection of values that can be read directly. Rewrite theory specifications with implicit states or explicit states are called *implicit-state rewrite theory specifications* and *explicit-state rewrite theory specifications* respectively[2].

2.2.1. Implicit-state rewrite theory specifications. In Maude implicit-state specifications, states are recursively specified like those in OTS/CafeOBJ specifications and transitions are also specified with equations. Definitions of states and transitions are essentially inherited from OTS/CafeOBJ specifications. In implicit-state specifications, there is also a sort declared to represent the state space. Since Maude has only one kind of sort that corresponds to visible sorts in CafeOBJ, the state space is denoted by a conventional sort in implicit-state specifications. A constant of the sort for state space is declared as the initial state. Some operators are defined over the sort for state space and some other parameters. Those operators take a state variable and related parameters to form new terms that represent new states. Some more operators need to be defined to retrieve values from a given state. The way of retrieving values from a state is defined recursively in the form of equations (or conditional equations) in Maude, by specifying how values are changed from an arbitrary state to its successor states. The declarations of operators in Section 2.1 in Maude are in the following forms:

```
op init : -> H .
op o : H Vo1 ... Vom -> Vo .
op t : H Vt1 ... Vtn -> H .
op c-t : H Vt1 ... Vtn -> Bool .
```

Where, `H` is the same kind of sort as others such as V_{o1}. Maude also does not differ behavioral operators from other conventional operators. All operators are declared with keyword `op` (or `ops`) as conventional operators.

For model checking purpose, transitions between a state and its successor states need to be declared explicitly by rewrite rules in the following form:

```
crl [t-tran] S => t(S, y1,...,yn)                    (I)
          if c-t(S, y1,...,yn) .
```

Where, `crl` is a keyword in Maude for the declaration of conditional rules; `t-tran` denotes the label of the rule and y_k is a constant of type D_{tk} for $k = 1,\ldots,n$. Maude basically requires each variable to appear in the left-hand side if it appears in the right-hand side or conditions to make rewrite rules executable. Constants are used in the successor state `t(S,y1,...,yn)` instead of variables in OTS/CafeOBJ specifications (see Section 2.1). An arbitrary state has $|D_{t1}| \times \ldots \times |D_{tn}|$ possible successor states w.r.t. the transition $t_{y_1,\ldots,y_n}$. Therefore, given a transition $t_{y_1,\ldots,y_n}$, we need to define $|D_{t1}| \times \ldots \times |D_{tn}|$ rewrite rules for all possible transitions between an arbitrary state and its successor states.

2.2.2. Explicit-state rewrite theory specifications. An explicit state is a collection of observable values that are declared in the form of "`Op Prams : Val`", where `Op` is an operator that corresponds to an observer; `Prams` is a list of parameters that correspond to those of the observer and `Val` represents the quantity of the observable value, which is called *observed value*. For instance, assume there is a state in which a Boolean variable `using` of process i is set true. We use the term "`using i : true`" to represent the value of the state. Similarly, we construct each term in this form for other values of the state. The collection of these terms identify the state explicitly from others. The changes of values between states are specified with rewrite rules in the following form:

```
crl [t-tran] : (Op Params : Val1)... =>
               (Op Params : Val2)...
  if c-t'(Val1,...) .
```

Where, `c-t'` is an auto-generated term corresponding to the effective condition $c\text{-}t_{y_1,\ldots,y_n}$ that denotes if the effective condition is satisfied in an explicit state. The rewrite rule says that if the effective condition holds, the value observed by observer `Op` w.r.t. parameters `Params` changes from `Val1` to `Val2`. Observable values that are changed by a transition with the same parameters should be specified together by one rule. Rewrite rules only need to specify the part of state that actually changes.

3. Modularization of translation strategies

Basic units of an OTS/CafeOBJ specification are modules. An OTS/CafeOBJ specification consists of exactly one module which specifies the OTS, and multiple modules which define necessary data types, operations and axioms that are used to specify the behaviors of the OTS.

2. In the rest of paper, we call them implicit-state specifications and explicit-state specifications in the case of not causing ambiguities.

As shown in Fig. 1, firstly, CafeOBJ modules are parsed according to the syntax of CafeOBJ. If these modules are parsed successfully, they are converted into Maude functional modules, with ignorance of the differences between hidden sorts and visible sorts, behavioral operators and other conventional operators. Besides, the module, in which the OTS is specified, is parsed to be a module of sort `OTSModule` which is predefined to represent the CafeOBJ module inside the translator. A module of sort `OTSModule` is called an *OTS module*, which is composed of six parts. `Header` is a sort for the name of an OTS module and `ImportList` is for a list of module names that denote their corresponding modules are imported to the current module. `HiddenSortDecl` denotes declarations of hidden sorts; `OpDeclSet` is a sort for declarations for conventional operators that are declared with keyword `op` (or `ops`); `OTSDeclSet` denotes a set of declarations for behavioral operators such as observers and action operators that are declared with keyword `bop` (or `bops`) and `EquationSet` is a set of equation definitions.

```
sort OTSDecl OTSDeclSet OTSModule .
subsort OTSDecl < OTSDeclSet .
op bop_:_->_. : Qid TypeList Type ->
  OTSDecl [ctor format(g o g o g o o o)] .
op bops_:_->_. : QidList TypeList Type ->
  OTSDecl [ctor format(g o g o g o o o)] .
op mod*_`{_____`} : Header ImportList
   HiddenSortDecl OpDeclSet OTSDeclSet
   EquationSet -> OTSModule
   [ctor gather(& & & & & &)
    format(d d s n++i ni d d ni n--i d)] .
op emptyOTSModule : -> OTSModule .
```

After the OTS/CafeOBJ specification is parsed successfully, translation part is called. In the translation part, there are two translation strategies modularized and embedded without changing other parts. They are called with different commands and work independently. Translation Strategy 1 produces a system module according to the functional modules which CafeOBJ modules are translated into. The system module specifies transitions of the OTS with states specified implicitly. Translation Strategy 2 generates a system module that specifies the OTS with states specified explicitly, according to the OTS module and the functional modules. Both of the rewrite specifications can be used by built-in model checking facilities of Maude for verification of the systems that they specify.

3.1. Translation into Implicit-state Specifications

3.1.1. Extension of the structure of state. Maude implicit-state specifications share the same data types and structures as states that are defined in OTS/CafeOBJ specifications. CafeOBJ modules are converted into Maude functional modules at syntax level without differing hidden sorts from other visible sorts and behavioral operators from other conventional operators. Data types and structures that are defined in functional modules are used to represent implicit states. Transitions

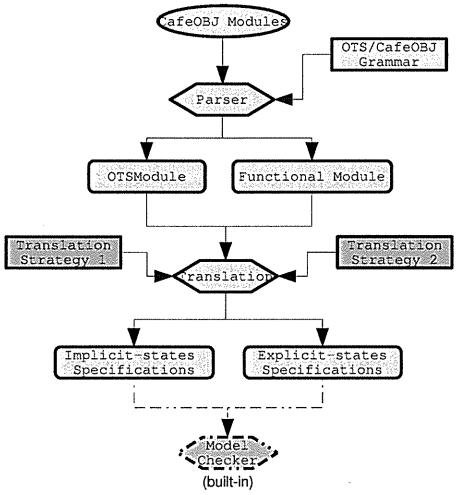

Fig. 1. Framework of the translator

between states are specified implicitly with equations like in OTS/CafeOBJ specifications.

However, Maude can not search a state space with transitions that are specified with equations. For model checking purpose, transitions between states need to be specified explicitly with rewrite rules like (I). Given an arbitrary state, we need to get its all possible successor states according to each transition $t_{y_1,\ldots,y_n}$. Successor states caused by a transition with different parameters may be different. To make successor states finite, we basically need to select a suitable finite set for each parameter sort $D_{t1},\ldots,D_{tn}$. Then, we get $|D_{t1}| \times \ldots \times |D_{tn}|$ possible successor states for an arbitrary state w.r.t. the transition $t_{y_1,\ldots,y_n}$. We need to define a rewrite rule to specify the transition from the arbitrary state to one of its possible successor states. There are $|D_{t1}| \times \ldots \times |D_{tn}|$ rewrite rules declared for the transition $t_{y_1,\ldots,y_n}$.

This approach is not only inflexible because we have to fix a finite set for each parameter sort before the translation, but inefficient with the increase of the number of rewrite rules. There is a way to overcome this inconvenience, in which a transition only needs a rewrite rule to be defined for it. Furthermore, we do not need to fix a finite set for each parameter sort before the translation. Given an arbitrary state `S`, we can get a successor state of `S` with a transition $t_{y_1,\ldots,y_n}$ and relevant parameters of sort $\mathtt{V}_{t1},\ldots,\mathtt{V}_{tn}$. We use variables $\mathtt{Y}_1,\ldots,\mathtt{Y}_n$ as parameters to get a general successor state of `S` with the transition $t_{y_1,\ldots,y_n}$, which is in the form of $\mathtt{t(S,Y}_1\mathtt{,\ldots,Y}_n\mathtt{)}$. Consequently, the rewrite rule for the transition is in the following form:

```
crl [t-tran] S => t(S, Y1,...,Yn)          (II)
          if c-t(S, Y1,...,Yn) .
```

However, the rewrite rule becomes inexecutable because

variables that appear at right-hand side and condition part do not appear at left-hand said. To make rewrite rules executable, we need to guarantee those variables appear at the left-hand side. In the left-hand side, there is supposed to be a term containing those variables as well as the state variable S that appear in the right-hand or condition part. Therefore, we extend the sort of state space to a new one that consists of two parts. One is a term that contains variables that appear in the right-hand side or condition part and the other part is a variable of sort of state. The new sort is called OTSState that is defined as follows:

```
sort OTSSate .
op [_;_,...,_] : H Type ... Type ->
                              OTSState [ctor] .
```

The constructor consists of two parts. The first part is a state of the sort H, which corresponds to the hidden sort in the OTS/CafeOBJ specification. The second part is a list of sorts which we call *common parameter list*. Elements in the list are separated by commas and each is a set of constants of the same sort. The common parameter list is determined by the parameters of action operators and it is automatically generated during the translation.

For convenience, the multiset $\{V_{t_1}, \ldots, V_{t_n}\}$ of parameter sorts of the transition $t_{y_1,\ldots,y_n}$ is denoted by P_t. The notation S_{V_*} stands for a sort that represents the set of constants of sort V_*. Assume that the set of action operators in the OTS/CafeOBJ specification is denoted by $\mathcal{A}$. For any action operator $t \in \mathcal{A}$, if there is a sort $V \in P_t$, we add the sort S_V to the common parameter list. Then, we get a common parameter list that contains all parameters for action operators in $\mathcal{A}$. There may be some redundant sorts in the list. For instance, assume there are two action operators $t_1, t_2 \in \mathcal{A}$ and sort $V \in P_{t_1}$, $V \in P_{t_2}$. We do not need to add twice the sort V to the list. If the sort V appears twice or more times in P_t for $t \in \mathcal{A}$, we have to add the sort S_V twice or more times respectively to the common parameter list. To make the target rewrite rule simpler, we reduce the common parameter list to the minimal one by removing those redundant sorts. A common parameter list is minimal in the sense that the length of the list is minimal.

We define some functions (see Appendix A) to get the minimal common parameter list for a collection of action operators. Function buildTypeList takes a collection of declarations of action operators and returns a list of sorts. The list is composed of sorts that there is at least one action operator taking as parameter sorts. Function combineTypeList combines two lists of sorts TyL and TyL' together with two auxiliary functions filter and cover. Function filter gets the common part of TyL and TyL' and cover gets those sorts that only appear either in TyL or in TyL'. The two parts are combined together as the returned value of combineTypeList.

For each sort V_* in the target list, if V_* does not have a corresponding sort declared to represent the set of items of V_*, we declare a new sort S_{V_*} and related operations on S_{V_*} for V_*. For example, action operators t_1 and t_2 are declared in the OTS/CafeOBJ specification as follows:

```
bop t1 : H V1 V2 -> H
bop t2 : H V1 V2 V3 V3 -> H
```

The minimal common parameter list for the action operator set $\{t_1, t_2\}$ is "$S_{V_1}, S_{V_2}, S_{V_3}, S_{V_3}$".

3.1.2. Construction of rewrite rules for transitions. This step is to build transitions between two states with rewrite rules according to the action operators that are declared in the OTS/CafeOBJ specification. Terms at left-hand side and right-hand side of rewrite rules are of the sort OTSState.

The function buildTransitionRules (see Appendix A) generates rewrite rules for all action operations. It takes two parameters and returns a set of rewrite rules. One parameter is the declaration of the constructor of sort OTSState and the other is a collection of declarations of action operators that are defined in OTS modules. The declaration of the constructor contains the name of the constructor OP and a list of sorts of common parameter list TyL. A declaration of an action operator contains the name of the action operator OP' and a list of sorts of parameters TyL' as well as a hidden sort Ty'. Functions buildLHS and buildRHS generate terms for left-hand side and right-hand side respectively. Given the constructor of sort OTSState, the list of sorts of parameters, an action operator OP', the list of sorts of its parameters and the hidden sort Ty', buildLHS returns a term that represents an arbitrary state of OTSState. A typical term at the left-hand side term is in the following form:

```
[S ; V1@t11:V1 V1Set@t11:V1Set,
     V2@t11:V2 V2Set@t11:V2Set,
     V3@t11:V3 V3Set@t11:V3Set,
     V3@t12:V3 V3Set@t12:V3Set ] .
```

A variable X of sort S can be declared on the fly like X:S. Note that V_1Set is the sort of sets consisting of terms of sort V_1. Variable names are generated automatically with a mechanism for avoiding name conflict. The number following "t1" in the variable name is used to avoid name conflict of variables in the case that two or more variables of the same sort appear in the term like V3@t11 and V3@t12.

Function buildRHS takes the same parameters as function buildLHS. It returns a state of OTSState that represents all possible successor states of the state at left-hand side w.r.t. the given action operator. The first part of the term at right-hand side is a term that represents a successor state of S and the other part is the minimal common parameter list without any changes with the one at left-hand side. A term that represents a successor states is composed of three parts: an action operator, the state S and parameters except S that are available in the minimal common parameter list.

Take t_1 for example. Term t1(S,V1@t11,V2@t11) is a successor state of S w.r.t. t_1. Because variables V1@t11 and V2@t11 can be replaced with all possible constants of sort V_1 and V_2, t1(S,V1@t11,V2@t11) represents all possible

successor states of S w.r.t. t_1. The right-hand side term for t_1 is as follows:

```
[t1(S, V1@t11:V1, V2@t11:V2) ;
     V1@t11:V1 V1Set@t11:V1Set,
     V2@t11:V2 V2Set@t11:V2Set,
     V3@t11:V3 V3Set@t11:V3Set,
     V3@t12:V3 V3Set@t12:V3Set ] .
```

Function buildCondition builds a condition part for a conditional rewrite rule. The condition part of the target rewrite rule can be constructed by replacing the action operator by its corresponding effective condition operator in the successor state. For example, we get the condition part c-t1(S,V1@T11,V2@T11) for t_1 by replacing t1 with c-t1 in term t1(S,V1@T11,V2@T11), where $c\text{-}t_1$ is the effective condition of t_1.

With terms of left-hand side, right-hand side and condition part, a conditional rewrite rule can be constructed in a system module. As an example, the rewrite rule with respect to t_1 is declared as follows:

```
crl [t1] :
    [S ; V1@t11:V1 V1Set@t11:V1Set,
         V2@t11:V2 V2Set@t11:V2Set,
         V3@t11:V3 V3Set@t11:V3Set,
         V3@t12:V3 V3Set@t12:V3Set] =>
    [t1(S, V1@t11:V1, V2@t11:V2) ;
         V1@t11:V1 V1Set@t11:V1Set,
         V2@t11:V2 V2Set@t11:V2Set,
         V3@t11:V3 V3Set@t11:V3Set,
         V3@t12:V3 V3Set@t12:V3Set]
    if c-t1(S, V1@T1, V2@T1) .
```

Function buildTransitionRules is defined recursively. It generates a rewrite rule for each action operator and returns none for an empty set of declarations of action operators.

3.1.3. Generation of a system module. With rewrite rules generated for all action operators, a system module is built by function buildTransitionSystem (see Appendix A). It generates a system module according to the OTS/CafeOBJ module that specifies the OTS. The predefined functional module STATE is imported to the target system module. The auto-generated functional module ME which the OTS module is translated into, is also imported to the system module. Function buildSortListSet is used to declare sort S_{V_*} for each sort V_* in TyL in the case that S_{V_*} is not defined in the input OTS/CafeOBJ modules. The returned value of buildSortListSet is assigned to variable SS and added to the target system module. Function buildListConnector declares an operator "_,_" for each S_{V_*} which denotes concatenation operation with identity the empty item. The declaration of the constructor of OTSState is returned by the function buildStateConstructor and added to the module with the predefined function addOps. Variable OPDS′ denotes a set of declarations of constants. These constants are empty identities of sort S_{V_*}. Rewrite rules that are generated by function buildTransitionRules are added by function addRls. The generated system module is added to the database which is initiated in Full Maude.

When using an implicit-state specifications for the purpose of model checking in Maude, we need to give an initial state of sort OTSState, from which Maude starts searching the whole state space. Because the initial state of an OTS has been declared in the OTS/CafeOBJ specification and translated into an equivalent term in functional modules in Maude, we only need to initiate the minimal common parameter list, like $((v_{11}\ v_{12}),(v_{21}\ v_{22}),(v_{31}\ v_{32}),(v_{31}\ v_{32}))$, where v_{n1} and v_{n2} $(n = 1, 2, 3)$ are constants of sort V_n, which are declared beforehand. It is worth pointing out that set $(v_{31}\ v_{32})$ has to be declared twice in the parameter list, because the sort V_3 appears twice in the transition t_2. We get an initial state like [init ; $((v_{11}\ v_{12}),(v_{21}\ v_{22}),(v_{31}\ v_{32}),(v_{31}\ v_{32}))$] of sort OTSState, from which Maude LTL model checker can build a search graph with rewrite rules that are declared before, to check if there is a state violating the property to be verified.

3.2. Translation into Explicit-state Specifications

3.2.1. Construction of Explicit-state Structure. States are specified implicitly in OTS/CafeOBJ specifications. To build an explicit-state specification in Maude, implicit states need to be translated into explicit states first. An explicit state is a collection of observable values. These observable values are regarded as basic units of an explicit state. A sort OValue is defined to represent observable values. A collection of terms of sort OValue form a state. To differ from the state sort in the OTS/CafeOBJ specification, a new sort OTSState for state is declared beforehand in functional module STATE-ES as follows:

```
fmod STATE-ES is
  sorts OTSState OValue .
  subsort OValue < OTSState .
  op nil : -> OTSState .
  op _ _ : OTSState OTSState -> OTSState
                [assoc comm id: nil] .
endfm
```

Sort OValue is a subsort of OTSState. The constructor of sort OValue consists of three parts: an observation operator, parameters of the observation operator and the value that it observes. An instance of sort OValue is called an observable value. A state consists of one or more observable values. Declarations of constructors for OValue are in the form of "op o_:_ : ParameterList V -> OValue" in Maude, where o is an observation operator that is declared in CafeOBJ; ParameterList denotes parameters of o except for the hidden sort that is declared for state space and V is the sort of the value that o observes. For instance, assume there is an observation operator o declared for observer $o_{x_1,\ldots,x_m}$

in CafeOBJ like "bop o : H $V_{o1} \ldots V_{om}$ -> V". Its corresponding declaration as an observable value in Maude is "op o_..._:_ : $V_{o1} \ldots V_{om}$ V -> OValue". In case of $m = 0$, ParameterList is null. The declaration is written like "op o`:_ : V -> OValue". Assume the value that $o_{x_1,\ldots,x_m}$ observes in a state S with parameters $x_1, \ldots, x_m$ is C_V, where parameters $x_1, \ldots, x_m$ are constants of sorts $V_{o1}, \ldots, V_{om}$ and C_V is a constant of sort V. Its corresponding observable value is "o x_1 ... x_m:C_V" which is a part of state S.

The operator "_ _" is used to combine two states of OTSState to form a new state that is still of sort OTSState. The commutativity and associativity of the operator means that there is no particular order for values in an explicit state.

In a state, an observer $o_{x_1,\ldots,x_m}$ returns different values with different parameters. To make returned values finite, we need to select a suitable finite set for each parameter sort [8]. For each sort $V_k (k = o1, \ldots, om)$ of $o_{x_1,\ldots,x_m}$, we give a finite set of V_k denoted by FT_{V_k} Maude can not select a finite sort for each parameter sort automatically. We need to manually fix finite sets for those parameter sorts in OTS/CafeOBJ specifications by declaring finite constants for each finite sort[3], before loading them into the translator.

Then, we can get $|FT_{V_{o1}}| \times \ldots \times |FT_{V_{om}}|$ observable values w.r.t. $o_{x_1,\ldots,x_m}$ in each state S. Observable values are in form "o x_1 ... x_m : T_V", where T_V is a term of sort V that depends on $x_1, \ldots, x_m$. These observable values are parts of state S w.r.t. $o_{x_1,\ldots,x_m}$. Similarly, observable values of other observers are constructed in state S. The collection of all these observable values is regarded as an explicit state of state S.

3.2.2. Construction of Initial States. In OTS/CafeOBJ specifications, initial states are implicitly specified with equations (see Section 2.1). In the initial state, an observer may return different values with different parameters. We need to get all observable values of all observers to construct an initial explicit state that is behaviorally equivalent to the implicit one. For the observer $o_{x_1,\ldots,x_m}$, we get the term $f(x_1, \ldots, x_m)$ as the value that $o_{x_1,\ldots,x_m}$ returns with parameters $x_1, \ldots, x_m$, by replacing variables $X_1, \ldots, X_m$ with constants $x_1, \ldots, x_m$ in (1). An observable value can be constructed as "o x_1 ... x_m:$f(x_1, \ldots, x_m)$". Similarly, we can get other observable values for $o_{x_1,\ldots,x_m}$ with other different parameters. Furthermore, we get observable values with other observer for the initial state. The collection of all these observable values is of sort OTSState and forms the initial explicit state that satisfies the following equation:

```
eq init' = (o x1 ... xm:f(x1,...,xm)) ...
```

Where, init' is a constant of sort OTSState that represents the initial explicit state.

A function buildInitialState (see Appendix B) is defined to construct an initial explicit state and function buildInitialEquation1 declares an equation that says a constant is equivalent to the initial explicit state. Function buildInitialEquation calls the two functions above to get the equation and adds it to the target system module. Function buildInitialEquation takes five parameters. Sort Bop2opList denotes a list of mapping from a declaration of an action operator in OTS/CafeOBJ modules to its corresponding declaration of observable value in Maude; sort TypeConstantsList denotes a list of items that record all constants of parameter sorts; OTSModule is the input OTS module; SModule in parameters is an auto-generated system module to which the initial state will be added and FModule is a flattened functional module that is translated from the input OTS/CafeOBJ modules. A flattened module in Maude is a module with its sub-modules flattened to it [16]. First, the buildInitialEquation function calls the function unfoldEquationSet to flatten the equations in FM by replacing variable parameters of observers with constants, like "eq o(init, $x_1, \ldots, x_m$) = f($x_1, \ldots, x_m$)". The buildInitialState function is used to build a list of observable values to represent the initial state with these flattened equations and the corresponding constructors of observers. Observable values are like "o x_1 ... x_m:f($x_1, \ldots, x_m$)" and the collection of them is the initial explicit state that is behaviorally equivalent to the one defined in the OTS module. Function buildInitialEquation1 returns an equation that says the constant init of sort OTSState is equivalent to the initial state so that we can use the constant to denote the initial explicit state. Function buildInitialEquation calls addEqs to add the equation to the target rewrite system.

3.2.3. Construction of transitions between explicit states. Like in the implicit-state specification, the transition between a state and each of its successor states needs to be specified explicitly in the explicit-state specification. For a dynamic system, given an arbitrary state, it is sufficient to specify all possible transitions from an arbitrary state to its successor states.

Given an action operator $t_{y_1,\ldots,y_n}$ with constants $y_1, \ldots, y_n$ of sorts $V_{t1}, \ldots, V_{tn}$ as its parameter, first, we need to construct an arbitrary explicit state. An arbitrary explicit state is composed of a collection of observable values with variables as their corresponding observed values. For example, we take variable X_V as the observed value of observer $o_{x_1,\ldots,x_m}$ w.r.t. parameters $x_1, \ldots, x_m$. The corresponding observable value is "o x_1 ... x_m : X_V". Because rewrite rules only need to specify the part of a state that actually changes, observable values that are not changed by a transition with some parameters do not need to be considered into the target rewrite rule if its observed value is not used by other observable values. For instance, if the observed value of $o_{x_1,\ldots,x_m}$ w.r.t. parameters $x_1, \ldots, x_m$ is not changed by the transition $t_{y_1,\ldots,y_n}$ with some parameters and the observed values is also not used by other observable values, the observable value "o x_1 ... x_m : X_V" can be removed from the target rewrite rule that specifies the transition $t_{y_1,\ldots,y_n}$ with the related parameters.

3. We do not consider the finite sort with constructors in the current implementation of the translator.

To get those observable values that are changed by $t_{y_1,\ldots,y_n}$ with constants $y_1,\ldots,y_n$, we get the relevant equations that specify the transition $t_{y_1,\ldots,y_n}$ in form of (2) and replace the parameters of $t_{y_1,\ldots,y_n}$ with constants $y_1,\ldots,y_n$. We get a collection of equations in the following form:

```
ceq o(t(S,y1,...,yn),X1,...,Xm) =                    (3)
    e-t(S,y1,...,yn,X1,...,Xm)
    if c-t(S,y1,...,yn) .
```

In the case that a transition does not have effective condition, the equation is in the form like:

```
eq o(t(S,y1,...,yn),X1,...,Xm) =                     (4)
    e-t(S,y1,...,yn,X1,...,Xm) .
```

We only consider those observers that equations corresponding to the action operator $t_{y_1,\ldots,y_n}$ specify.

Then, we flatten the equations by replacing the variables that are used as parameters of observers with all possible values that the variables can be, and get equations in the following forms:

```
ceq o(t(S,y1,...,yn),x1,...,xm) =                    (5)
    e-t(S,y1,...,yn,x1,...,xm)
    if c-t(S,y1,...,yn) .
```

```
eq o(t(S,y1,...,yn),x1,...,xm) =                     (6)
    e-t(S,y1,...,yn,x1,...,xm) .
```

For each equation, we can build an observable value by introducing a new variable like "o x_1 ... x_m : X_V". The observable value is a part of the arbitrary state. The collection of these observable values compose the arbitrary state which is the term at left-hand side of the target rewrite rule.

The next step is to get a successor state of the arbitrary explicit state. For each observable value in the arbitrary state, we need to know how the observable value is changed by an action operator and construct its corresponding observed value in the successor state. For observer $o_{x_1,\ldots,x_m}$ with parameters $x_1,\ldots,x_m$, according to (5), the observed value is changed into "e-t(S,$y_1,\ldots,y_n$,$x_1,\ldots,x_m$)" by the transition $t_{y_1,\ldots,y_n}$ with constants $y_1,\ldots,y_n$. We replace the term "o(S,$x_1,\ldots,x_m$)" with the variable X_V in term "e-t(S,$y_1,\ldots,y_n$,$x_1,\ldots,x_m$)" and get a term like "e-t'(X_V,$y_1,\ldots,y_n$,$x_1,\ldots,x_m$)" which is the observed value of $o_{x_1,\ldots,x_m}$ with parameters $x_1,\ldots,x_m$ in the successor state. The observable value "o x_1 ... x_m:e-t'(X_V,$y_1,\ldots,y_n$,$x_1,\ldots,x_m$)" is a part of the successor state.

If there is no such an equation declared in the OTS/CafeOBJ specification, it means the transition instance $t_{y_1,\ldots,y_n}$ with parameters $y_1,\ldots,y_n$ does not change values that are observed by $o_{x_1,\ldots,x_m}$. In this case, the observable value stays the same without any change. Successor observable values of other observable values are constructed similarly. The collection of all successor observable values composes the successor state of the arbitrary state w.r.t. the transition $t_{y_1,\ldots,y_n}$ with parameters $y_1,\ldots,y_n$. Similarly, successor states of the arbitrary state with the transition $t_{y_1,\ldots,y_n}$ and other different parameters

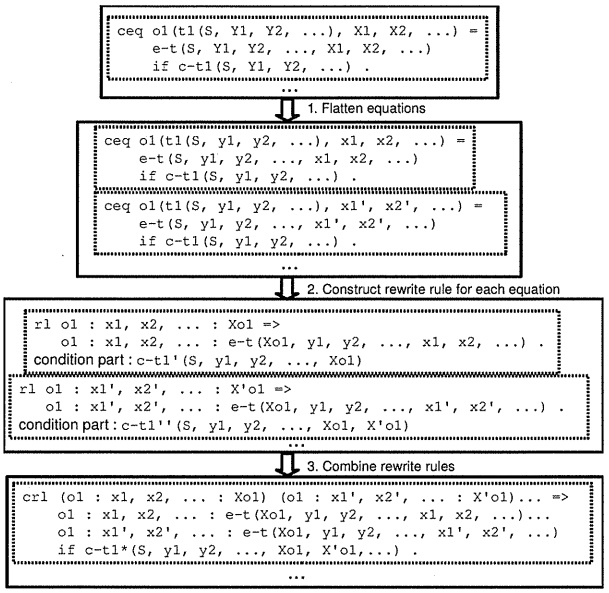

Fig. 2. Construction of rewrite rules

can be constructed. Furthermore, successor states with other transitions can be built in the same way.

Generally, a transition instance takes place under the condition that its corresponding effective condition holds in the state. In the case that an action operator has an effective condition, we need to build a conditional rewrite rule to specify this transition. Besides the arbitrary state and its successor state w.r.t. the transition $t_{y_1,\ldots,y_n}$ and parameters $y_1,\ldots,y_n$, a condition part is also needed for the target conditional rewrite rule.

In the OTS/CafeOBJ specification, there is an effective condition $c\text{-}t_{y_1,\ldots,y_n}$ declared and a related equation defined for the term c-t(S,$Y_1,\ldots,Y_n$). The right-hand side of the equation is a term that returns a Boolean value. The term contains values that are returned by observers in the state S. By replacing these values with related variables that are introduced to represent an explicit state for S, we get a new Boolean expression as the effective condition of the transition $t_{y_1,\ldots,y_n}$ with parameters $y_1,\ldots,y_n$. The expression is represented by the term c-t'($X_1,\ldots,X_{n'}$), where $X_1,\ldots,X_{n'}$ are variables introduced for observable values and $n' = |FT_{V_{o1}}| \times \ldots \times |FT_{V_{om}}| \times |FT_{V_{t1}}| \times \ldots \times |FT_{V_{tn}}|$.

With the arbitrary explicit state, its successor state and the condition term, a conditional rewrite rewrite is defined for the transition $t_{y_1,\ldots,y_n}$ with parameters $y_1,\ldots,y_n$. Function convertEq2Rl (see Appendix C) converts equations that are defined in OTS/CafeOBJ modules to specify behaviors, into rewrite rules in Maude which specify the equivalent behaviors of dynamic systems. It takes a set of declarations of action operators, a set of equation and two other related parameters. The function returns a set of rewrite rules for the target system module. Equations first are flattened by

replacing variables in parameters with constants, as shown in Fig. 2. For each declaration of an action operator, function `convertEq2Rl` calls function `buildAllSuccessors` to get all possible successors that are in OTS/CafeOBJ syntax like `OP(S, p1,...,p2)` according to the given action operator and different parameters. A transition from the arbitrary state `S` to one of its successor states is called an *transition instance*. Function `convertEq2RlAux` is defined to get a set of rewrite rules for all transition instances. For each transition instance, `convertEq2RlAux` calls function `convertEq2RlAux1` to get a set of rewrite rules that specify the changes of all observed values caused by the transition instance. Function `convertEq2RlAux1` selects the equations that are related to the transition instance and call function `convertEq2RlAux2` with these equations. Function `convertEq2RlAux2` builds a rewrite rule for each related equation and simplifies the condition part with the new introduced variable by calling function `buildSingleRule` (see the second step in Fig. 2). The set of rewrite rules and the condition part are passed to the function `combineRules` which constructs a new rewrite rule by combining all the left-hand side terms of the rewrite rules as the left-hand side term of the new rule and all the right-hand side terms as the right-hand side term (see the third step in Fig. 2). If the condition part is not null, then `combineRules` returns a condition rewrite rule. The new generated rewrite rule is passed to function `eliminateObservers` which replaces the observers in the rewrite rule with their corresponding operators that are auto-generated in Maude. Then, we get a rewrite rule for a concrete transition.

A system module can be constructed with the three steps above for an OTS. The system module mainly consists of the following four parts with states specified explicitly:

1) functional modules that are defined for data types;
2) operators that are declared for observers of OTSs;
3) an equation that is defined for the initial state;
4) rewrite rules that are built for transitions.

3.3. Correctness and Comparison

From the practical viewpoint, the first translation strategy generates a system module that explicitly specifies transitions between states, without changing behaviors that are defined in its corresponding OTS/CafeOBJ specification. If a counter-example is found in translated implicit-state specifications, there must be a path from the initial state to the state that violates the desired property. In the original OTS/CafeOBJ specification, that state is also reachable. Consequently, the counter-example is also a counter-example of the original OTS/CafeOBJ specification. The correctness of the second translation strategy is proved at theoretical level in [8].

Both of the two translation strategies have strengths and weaknesses. Although state space is required to be finite or bounded for model checking, the first translation strategy does not require related parameter sorts to be finite before translation. We only need to provide finite sets for those sorts when we construct initial states for model checking. The second translation strategy enumerates all possible rewrite rules between explicit states. We need to give an appropriate finite set for each parameter sort before the translation. Any changes to a parameter sort cause some rewrite rules to be removed or added. We have to re-translate the specification once we remove or add some elements to parameter sorts.

However, to evaluate a value of an implicit state, it needs to backtrack to the initial state and evaluate the value of each state in the path from the initial state to that state. The implicit-state specification is not so efficient as the explicit-state one in which we can get values of explicit states directly.

4. Execution Environment and an Example

4.1. Execution Environment

The translator runs on the top of Full Maude. In the environment of Full Maude [15], we can launch the translator with the command `load ots`. Besides the commands provided by Full Maude, the translator supports it own two commands, namely `conv2exps` and `conv2imps` that are used to generate explicit-state rewrite specifications and implicit-state rewrite specifications respectively. Like Full Maude, the translator needs commands to be enclosed in parentheses to differ from commands supported by Core Maude. Besides, each modules in OTS/CafeOBJ specifications has to be enclosed in parentheses to differ from the input module for Core Maude [12], [16].

4.2. An Example of Semaphore Mechanism

Semaphore is a mechanism for restricting access to shared resources to a fixed number of processes at a time. The place where at most one process is allowed to enter is called the critical section. In a system with semaphore, there is a public integer variable x that all processes can access. Each process has a private Boolean variable to denote whether it is in the critical section. When a process p wants to enter the critical section, it first checks if $x > 0$. In the case that $x > 0$, p decreases x by 1 and sets its private Boolean variable true. The should be done atomically. If $x \leq 0$, process p has to wait until $x > 0$. If process p wants to leave the critical section, p has to increase x by 1 and sets its private Boolean variable false, which is also an atomic operation.

4.2.1. OTS/CafeOBJ specification for semaphore. To specify the semaphore, we need to declare a new sort called `Pid` in a loose module `PROCESS` to identify each process. The module `PROCESS` is defined as shown in Fig. 3.

With the module `PROCESS` and the built-in module `INT`, we can define the following module `SEMAPHORE` to specify the semaphore mechanism. Modules `PROCESS` and `INT` are imported. A hidden sort `Sys` is declared to represent state space. There is a observation `using` taking `Pid` as one of parameter and returning a Boolean value. It denotes whether

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

Fig. 3. System module: SEMAPHORE-EXPS

a process is using the shared resources or not. Observation `semaphore` only takes `Sys` as its parameter and returns an integer that stands for the value of the semaphore in a state. The first two equations in the module say that every process is not in the critical section and the value of the semaphore is 1. There are two action operators `down` and `up` specifying the *enter* and *leave* operations of each process respectively.

The third conditional equation says that if process `Y11` satisfies the effective condition of the action operator `down` in state `S`, namely the value of `c-down(S, Y11)` is true, we get a new state `down(S, Y11)` that differs from the state `S`. In the state `down(S, Y11)`, the value of the observation `using` is *true* in terms of process `Y11`. The transition by action operator `down` with process `Y11` only changes the value for `Y11`. Other processes denoted by `X11` returns values in the predecessor state `S`, that is `using(S, X11)`. In the case that the effective condition is not satisfied, the state `S` and its successor state `down(S, Y11)` can be regarded to be behaviorally equivalent. Similarly, the last four equations in the module are defined for the action operator `up`.

4.2.2. Translation results. As mentioned in Section 4.1, the input specification should be enclosed in parentheses before being loaded into the translator. Besides, a dot "." has to be added manually to the end of each declarations of operations and variables, because Maude can not recognize the invisible character of line feed. By convention, we use the character dot "." as the end of a line.

If the modules `PROCESS` and `SEMAPHORE` are correct in terms of syntax, they can be parsed and loaded successfully by the translator. Otherwise, the translator will output an error message pointing out the place that violates the CafeOBJ grammar. After loading the modules into the translator, we can use the command `(conv2imps .)` to get an implicit-state

```
mod SEMAPHORE-IMPS is
  including STATE . including SEMAPHORE .
  sorts PidSet .    subsort Pid < PidSet .
  op __ : PidSet PidSet -> PidSet [assoc comm id: nil] .
  op `[_;_`] : Sys PidSet -> OTSState [ctor] .
  op nil : -> PidSet  [ctor] .
  crl [S:Sys ; Pid@down1:Pid PidSet@down1:PidSet] =>
      [down(S:Sys,Pid@down1:Pid); Pid@down1:Pid
      PidSet@down1:PidSet] if c-down(S:Sys,Pid@down1:Pid)
      = true [label down] .
  crl [S:Sys ; Pid@up1:Pid PidSet@up1:PidSet] =>
      [up(S:Sys,Pid@up1:Pid); Pid@up1:Pid
      PidSet@up1:PidSet] if c-up(S:Sys,Pid@up1:Pid)
      = true [label up] .
endm
```

Fig. 4. System module: SEMAPHORE-IMPS

rewrite specification named `SEMAPHORE-IMPS`. The translator will output a message saying the translation is successfully done. The `show module` command asks the translator print out the content of the module `SEMAPHORE-IMPS` as show in Fig. 4.

Where, the module `STATE` is a built-in module in the translator with the sort `OTSState` declared in it. The module `SEMAPHORE` is a function module that is translated from the loose module `SEMAPHORE` in CafeOBJ. New sort `PidSet` is declared to be a set sort of sort `Pid`. The first rewrite rule says that if a process `Pid@down1` satisfies the effective condition in the state `S`, it can reach its successor state `down(S, Pid@down1)`. Similarly, the second conditional rule specifies the action operator `up`.

With the system module, Maude can search the whole reachable state space to check whether a property holds in every reachable state from the initial state. For instance, there are two processes in the system, namely `p` and `q`. The initial state should be `[init ; p q]`. We feed the following command into Maude to check whether there is a state `S` in which both `p` and `q` are in the critical section:

```
search [1, 100] : [init ; p:Pid q:Pid]
    =>* [S:Sys ; p:Pid q:Pid] such that
          using(S:Sys, p:Pid) and
          using(S:Sys, q:Pid) .
```

Maude returns "No solution", which means that there is no such kind of state in the first 100 depth from the initial state. The fist number in square brackets denotes the desired number of solutions that we hope Maude to return and the second means the maximal depth of the search.

In the OTS of semaphore mechanism, besides the sort `Sys`, there is only one additional parameter `Pid` for observers and transitions. To get an explicit-state rewrite specification, we first choose a non-empty and finite subset for sort `Pid`. We declare two constants `p` and `q` for `Pid` in the module `PROCESS` by adding the statement "`ops p q : -> Pid`". Then, we feed the two CafeOBJ modules into the translator. With the command `(conv2exps .)`, we have a system module named `SEMAPHORE-EXPS` generated as shown in Fig. 5.

The following command asks Maude to search the whole

```
mod SEMAPHORE-EXPS is
  including STATE-ES .  protecting PROCESS .
  protecting INT .      sorts Sys .
  op c-down : Sys Pid -> Bool .     op c-up : Sys Pid -> Bool .
  op down : Sys Pid -> Sys .        op init : -> OTSState .
  op semaphore`:_ : Int -> OValue . op up : Sys Pid -> Sys .
  op using_:_ : Pid Bool -> OValue .
  eq init = semaphore : 1 (using p : false) using q : false .
  crl semaphore : semaphore@:Int  (using p : usingp:Bool)
      using q : usingq:Bool => semaphore :(semaphore@:Int + 1)
      (using p : if p == p then false else usingp:Bool fi)
      using q : if q == p then false  else usingq:Bool fi
    if usingp:Bool = true [label upp] .
  crl semaphore : semaphore@:Int (using p : usingp:Bool)
      using q : usingq:Bool => semaphore :(semaphore@:Int + 1)
      (using p : if p == q then false else usingp:Bool fi)
      using q : if q == q then false else usingq:Bool fi
    if usingq:Bool = true [label upq] .
  crl semaphore : semaphore@:Int using p : usingp:Bool)
      using q : usingq:Bool => semaphore :(semaphore@:Int - 1)
      (using p : if p == p then true else usingp:Bool fi)
      using q : if q == p then true  else usingq:Bool fi
    if not usingp:Bool and semaphore@:Int > 0 = true [label downp] .
  crl semaphore : semaphore@:Int (using p : usingp:Bool)
      using q : usingq:Bool => semaphore :(semaphore@:Int - 1)
      (using p : if p == q then true else usingp:Bool  fi)
      using q : if q == q then true else usingq:Bool fi
    if not usingq:Bool and semaphore@:Int > 0 = true [label downq] .
endm
```

Fig. 5. System module: SEMAPHORE-EXPS

reachable state space, restricting to two processes `p` and `q`, to check whether there is a state that violates the mutual exclusion property.

```
search [1] init =>* (using p : true)
       (using q : true) S:OTSState .
```

Maude also returns "No solution".

5. Related work

There have been many formal specification languages and verification tools proposed. A combination of some of those languages and tools enables us to take advantages of them and get rid of their weaknesses. One promising way of combing two verification tools is to translate specifications written in one language into the ones written in the other language. Some studies have been conducted on the topic of specification translation.

Attiogbe has proposed a way to model check state machines written in Event B by translating state machines in Event B into those in PROMELA that is the specification language for SPIN [17], with the purpose of model checking liveness properties for state machines written in Event B. George and Haxthausen have formally analyzed the Mondex electronic purse protocol [2]. They have specified Mondex in the RAISE specification language (RSL) and analyzed Mondex by translating the specifications in RSL into those for the PVS theorem prover and the SAL model checkers. The reason why they select both PVS and SAL as their verification tools is that they are able to make use of advantages of PVS for interactive theorem proving and SAL for model checking.

Their translators are implemented in conventional programming languages, while our translator described in the paper has been implemented in Maude. Based on our experiences, it shows that a translator can be implemented with less effort and be more extensible thanks to Maude meta-programming facilities.

Much effort has been made on model transformation in Model-Driven (Software) Engineering (MDE) [18]. In MDE, high-level models are transformed towards more implementation-oriented models. Our technique described in the paper may also be contributed to model transformation, which is one piece of our future work.

6. Conclusion and future work

We introduce a modular implementation of a translator in Maude, which needs less effort than in other conventional programming languages. Translation strategies are modularized and embedded in the implementation. The modularization of the implementation allows us embedded different translation strategies in the translator. It is one piece of our future work to come up with other translation strategies for the translation from OTS/CafeOBJ specifications into Maude rewrite theory specifications.

Furthermore, we only focus ourselves on the introduction to the translator in this paper, instead of presenting a detailed description of its implementation. We are going to propose a methodology for developing formal tools with meta-programming facilities of Maude with the implementation of the translator as a concrete example. On the other hand, we would like to consider the translation from rewrite specification to behavioral specification, which seems more sensible in terms of the integration of model checking and theorem proving.

Acknowledgment

The authors would like to thank Prof. Jose Meseguer and Prof. Kokichi Futatsugi for their comments and advices on the translation strategies and the implementation of the translator. They also would like to thank Prof. Francisco Durán for his patient instruction on the implementation of Full Maude via mails.

References

[1] W. Kong, K. Ogata, T. Seino, and K. Futatsugi, "A lightweight integration of theorem proving and model checking for system verification," in *12th APSEC*. IEEE CS Press, 2005, pp. 59–66.

[2] C. George and A. Haxthausen, "Specification, proof, and model checking of the Mondex electronic purse using RAISE," *Formal Aspects of Computing*, vol. 20, no. 1, pp. 101–116, 2008.

[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 Brithday, LNCS*, vol. 4060, p. 596, 2006.

[4] R. Diaconescu and K. Futatsugi, *CafeOBJ Report: The Language, Proof Techniques, and Methodologies for Object-Oriented Algebraic Specification*. World Scientific, 1998.

[5] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott, *All about Maude: a high-performance logical framework: how to specify, program and verify systems in rewriting logic*. Springer, 2007, vol. 4350.

[6] S. Eker, J. Meseguer, and A. Sridharanarayanan, "The Maude LTL model checker," in *4th WRLA, ENTCS*, vol. 71. Elsevier, 2004, pp. 162–187.
[7] W. Kong, K. Ogata, and K. Futatsugi, "Algebraic approaches to formal analysis of the mondex electronic purse system," *6th IFM, LNCS*, vol. 4591, pp. 393–412, 2007.
[8] M. Nakamura, W. Kong, K. Ogata, and K. Futatsugi, "A specification translation from behavioral specifications to rewrite specifications," *IEICE Transactions*, vol. 91-D, no. 5, pp. 1492–1503, 2008.
[9] R. Diaconescu and K. Futatsugi, "Behavioural coherence in object-oriented algebraic specification," *J. UCS*, vol. 6, no. 1, pp. 74–96, 2000.
[10] J. A. Goguen and G. Malcolm, "A hidden agenda," *Theor. Comput. Sci.*, vol. 245, no. 1, pp. 55–101, 2000.
[11] M. Clavel, F. Duran, S. Eker, J. Meseguer, and M. Stehr, "Maude as a formal meta-tool," in *FM'99, LNCS*, vol. 1709. Springer, 1999, pp. 1684–1703.
[12] F. Duran and P. Olveczky, "A Guide to Extending Full Maude Illustrated with the Implementation of Real-Time Maude," in *7th WRLA, ENTCS*. Elsevier, 2008, (to appear).
[13] P. Olveczky and J. Meseguer, "Specification and analysis of real-time systems using Real-Time Maude," *7th FASE, LNCS*, vol. 2984, pp. 354–358, 2004.
[14] M. Clavel, F. Duran, J. Hendrix, S. Lucas, J. Meseguer, and P. Olveczky, "The Maude formal tool environment," *2nd CALCO, LNCS*, vol. 4624, pp. 173–178, 2007.
[15] F. Duran and J. Meseguer, "The Maude specification of Full Maude," *Computer Science Laboratory, SRI International, http://maude.csl.sri.com/papers*, 1999, (Manuscript).
[16] F. Duran, "A reflective module algebra with applications to the Maude language," Ph.D. dissertation, University of Malaga, 1999.
[17] C. Attiogbé, "A mechanically proved development combining b abstract systems and spin," in *4th QSIC*. IEEE CS Press, 2004, pp. 42–49.
[18] K. Lano and D. Clark, "Model transformation specification and verification," in *8th QSIC*. IEEE CS Press, 2008, pp. 45–54.

Appendix A
Key functions to build an implicit-state rewrite system

```
op buildTransitionSystem : OTSModule -> SModule .
op buildTypeList    : OTSDeclSet -> TypeList .
op combineTypeList : TypeList TypeList -> TypeList .
op buildStateConstructor : Type TypeList -> OpDecl .
op buildTransitionRules : OpDecl OTSDeclSet -> RuleSet .

ceq buildTransitionSystem(
    mod* ME { IL HSD OPDS OTSDS EqS }) =
    addImports((including 'STATE .) (including ME .),
     addSorts(SS,
      addSubsorts(declareSubsort(TyL),
       addOps(buildListConnector(SS),
        addOps(OPD,
         addOps(OPDS',
          addRls(buildTransitionRules(
            OPD, getTransitions(OTSDS)),
             setName(emptySModule, qid(string(ME) + "-IMPS")))))))))
   if TyL   := buildTypeList(
               getTransitions(OTSDS)) /\
      SS    := buildSortListSet(TyL) /\
      TyL'  := buildTypeListList(TyL) /\
      OPD   := buildStateConstructor(
               getSort(HSD), TyL')  /\
      OPDS' := addIdentityOp(TyL) .

eq buildTypeList((bop OP : Ty TyL -> Ty' .) OTSDS) =
   combineTypeList(TyL, buildTypeList(OTSDS)) .

eq combineTypeList(TyL, TyL') = (filter(TyL, TyL')) (cover(TyL, TyL')) .
eq buildStateConstructor(Ty, TyL) =
   (op buildOperator(size(TyL)) : Ty TyL -> 'OTSState [ctor] .) .

eq buildTransitionRules((op OP : Ty TyL -> 'OTSState [AttS] .),
                        (bop OP' : Ty' TyL' -> Ty' .) OTSDS) =
   (crl buildLHS(OP, OP', Ty', TyL, TyL') => buildRHS(OP, OP', Ty', TyL, TyL')
    if buildCondition(OP', Ty', TyL') [label(OP')] .)
   buildTransitionRules((op OP : Ty TyL -> 'OTSState [AttS] .), OTSDS) .

eq buildTransitionRules((op OP : Ty TyL -> 'OTSState [AttS] .), none) = none .

eq buildLHS(OP, OP', Ty', TyL, TyL') =
   OP[qid("S:" + string(Ty')), buildParameters(OP', TyL, nil)] .

eq buildRHS(OP, OP', Ty', TyL, TyL') =
   OP[OP'[qid("S:" + string(Ty')),
      buildSinglePara(OP', TyL', nil)], buildParameters(OP', TyL, nil)] .

eq buildCondition(OP, Ty', TyL') =
   (qid("c-" + string(OP))[qid("S:" + string(Ty')),
    buildSinglePara(OP, TyL', nil)] = 'true.Bool) .
```

Appendix B
Key functions to construct an initial explicit state

```
op buildInitialEquation :
   Bop2opList TypeConstantsList OTSModule SModule FModule -> SModule .
op buildInitialEquation1 : TermList -> Equation .
op buildInitialState : Bop2opList Constant EquationSet -> TermList .

ceq buildInitialEquation(BOPL, TCL, OTSM, SM, FM) =
    addEqs(buildInitialEquation1(TL), SM)
    if EqS := unfoldEquationSet(TCL, getEqs(FM)) /\
        TL := buildInitialState(BOPL, getInitialState(OTSM), EqS) .

eq buildInitialEquation1((T, TL)) =
   if TL =/= (empty).EmptyCommaList then (eq 'init.OTSState = '__[T,TL] [none].)
   else (eq 'init.OTSState = T [none] .) fi .

eq buildInitialState(BOPL, CON, EqS) =
   buildInitialStateAux(BOPL, CON, getEquations(CON, EqS)) .
eq buildInitialState(BOPL, CON, none) = empty .

eq buildInitialStateAux(BOPL, CON, (eq OP[TL] = T' [Attr] .) EqS ) =
   (buildInitialStateAux1(BOPL, OP, TL, T'), buildInitialStateAux(BOPL, CON, EqS)) .
eq buildInitialStateAux(BOPL, CON, none) = empty .

ceq buildInitialStateAux1(BOPL, OP, TL, T') =
    buildInitialStateAux2(BOP2OP, OP, TL, T')
    if BOP2OP := getBOP2OP(BOPL, OP) /\ BOP2OP =/= nil .

eq buildInitialStateAux2([(bop OP : TyL -> Ty .) ;
   (op OP' : TyL' -> Ty' [Attr] .)], OP, (T, TL), T') = (OP'[TL, T']) .
eq buildInitialStateAux2(BOP2OP, OP, TL, T') = empty [owise] .
```

Appendix C
Key functions to construct transitions between explicit states

```
op convertEq2Rl : Bop2opList TypeConstantsList OTSDeclSet EquationSet -> RuleSet .
op convertEq2RlAux : Bop2opList TermList EquationSet -> RuleSet .
op convertEq2RlAux1 : Bop2opList Term EquationSet -> RuleConditionPair .
op convertEq2RlAux2 : Bop2opList Term EquationSet EqCondition -> RuleConditionPair .
op buildSingleRule :
   Bop2opList Qid TermList Equation EqCondition -> RuleConditionPair .

ceq convertEq2Rl(BOPL, TCL, (OTSD OTSDS), EqS) =
       convertEq2Rl(BOPL, TCL, OTSD, EqS) convertEq2Rl(BOPL, TCL, OTSDS, EqS)
   if OTSDS =/= none .

eq convertEq2Rl(BOPL, TCL, (bop OP : Ty TyL -> Ty .), EqS) =
   convertEq2RlAux(BOPL, buildAllSuccessors(TCL, OP, TyL), EqS) .

ceq convertEq2RlAux(BOPL, (T, TL), EqS ) = Rl convertEq2RlAux(BOPL, TL, EqS)
    if TL =/= empty /\
   Rl := eliminateObservers(BOPL, combineRules(convertEq2RlAux1(BOPL, T, EqS))) .

eq convertEq2RlAux1(BOPL, OP[TL], (ceq OP'[OP[T, TL], TL'] = T' if EqC [AttrS] .) EqS)
```

```
   = convertEq2RlAux2(BOPL, OP[TL],
     (ceq OP'[OP[T, TL], TL'] = T' if EqC [AttrS] .) EqS, EqC) .

ceq convertEq2RlAux2(BOPL, OP[TL], EqS, EqC') =
    < EqC3 ; (Rl Rls) >
   if ((ceq OP'[OP[T, TL], TL'] = T' if EqC [AttrS] .) EqS') := EqS /\
      < EqC'' ; Rl > :=
      buildSingleRule(BOPL, OP, TL, (eq OP'[OP[T, TL], TL'] = T' [AttrS] .), EqC') /\
      < EqC3 ; Rls > :=  convertEq2RlAux2(BOPL, OP[TL], EqS', EqC'') .

ceq buildSingleRule(BOPL, OP, TL, (eq OP'[OP[T, TL], TL'] = T' [AttrS] .), EqC) =
     < replaceCondition(EqC, OP'[T, TL'], VAR) ;
     (rl  T'' =>  OP''[TL', replaceTerm(T', OP'[T, TL'], VAR)]
     [label(buildVar(OP, TL))] .) >
    if ([(bop OP' : Ty TyL -> Ty' .) ;
       (op OP'' : TyL' -> 'OValue [none] .) ] BOPL') := BOPL /\
       VAR := qid(string(buildVar(OP', TL')) + ":" + string(Ty')) /\
        T'' := OP''[TL', VAR] .
```