<table>
<thead>
<tr>
<th>Title</th>
<th>Modular Implementation of a Translator from Behavioral Specifications to Rewrite Theory Specifications (Extended Version)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Author(s)</td>
<td>Zhang, Min; Ogata, Kazuhiro</td>
</tr>
<tr>
<td>Citation</td>
<td>Research report (School of Information Science, Japan Advanced Institute of Science and Technology), IS-RR-2010-002: 1-16</td>
</tr>
<tr>
<td>Issue Date</td>
<td>2010-03-10</td>
</tr>
<tr>
<td>Type</td>
<td>Technical Report</td>
</tr>
<tr>
<td>URL</td>
<td><a href="http://hdl.handle.net/10119/8848">http://hdl.handle.net/10119/8848</a></td>
</tr>
<tr>
<td>Rights</td>
<td></td>
</tr>
<tr>
<td>Description</td>
<td>リサーチレポート（北陸先端科学技術大学院大学情報科学研究科）</td>
</tr>
</tbody>
</table>
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 analyze 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 ones1, 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/~s080005
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 $T$. 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 $T$ and how these quantities are changed by state transitions. An OTS $S$ is a triple $\langle \mathcal{O}, \mathcal{T}, \mathcal{T}^{\prime} \rangle$, where:

- $\mathcal{O}$: A finite set of observers. Each observer $o_{x_1:D_{x_1}, \ldots, x_m:D_{x_m}} : T \rightarrow D_o$ is an indexed function with $m$ indexes $x_1, \ldots, x_m$ of types $D_{x_1}, \ldots, D_{x_m}$ respectively. Given an OTS $S$ and two state $v_1, v_2 \in \mathcal{T}$, we say $v_1$ and $v_2$ are equivalent w.r.t $S$ denoted by $v_1 =_S v_2$, if and only if $\forall o_{x_1:D_{x_1}, \ldots, x_m:D_{x_m}} \in \mathcal{O}, \forall x_1 : D_{x_1}, \ldots, x_m : D_{x_m} \ni o_{x_1:D_{x_1}, \ldots, x_m:D_{x_m}} (v_1) = o_{x_1:D_{x_1}, \ldots, x_m:D_{x_m}} (v_2)$.
- $\mathcal{T}$: The set of initial states such that $I \subseteq \mathcal{T}$.
- $\mathcal{T}^{\prime}$: A finite set of transitions. Each transition $t_{y_1:D_{y_1}, \ldots, y_n:D_{y_n}} : T \rightarrow \mathcal{T}$ is an indexed function with $n$ indexes $y_1, \ldots, y_n$ of types $D_{y_1}, \ldots, D_{y_n}$ respectively, provided that $t_{y_1:D_{y_1}, \ldots, y_n:D_{y_n}} (v) =_S t_{\bar{y}_1:D_{\bar{y}_1}, \ldots, \bar{y}_n:D_{\bar{y}_n}} (v)$ for each $[v] \in \mathcal{T} / =_S$, each $y_1, y_2 \in [v]$ and each $y_k$ of type $D_{y_k}$ for $k = 1, \ldots, n$.

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 $T$. A hidden sort $H$ is declared in the form of $\ast [H] \ast$. In an OTS/CafeOBJ specification, an arbitrary initial state of the OTS $S$ is represented by a hidden constant $\text{init}$, each observer $o_{x_1:D_{x_1}, \ldots, x_m:D_{x_m}}$ by an observation operator $o$, each transition $t_{y_1:D_{y_1}, \ldots, y_n:D_{y_n}}$ by an action operator $t$ and an effective condition $c_{t_{y_1:D_{y_1}, \ldots, y_n:D_{y_n}}}$ by a conventional operator $c$-$t$. Hidden constant $\text{init}$, observation operator $o$, action operator $t$ and operator $c$-$t$ are declared as follows:

\begin{align*}
\text{op \ init} : & \rightarrow H \\
\text{bop \ o} : & H V_{o1} \ldots V_{om} \rightarrow V_o \\
\text{bop \ t} : & H V_{t1} \ldots V_{tn} \rightarrow H \\
\text{op \ c-t} : & H V_{t1} \ldots V_{tn} \rightarrow \text{Bool}
\end{align*}

Where, $H$ is a hidden sort denoting $T$; each $V_i$ is a visible sort denoting a data type $D_i$; $\text{Bool}$ is a visible sort denoting truth values and $X_k$ and $Y_k$ are CafeOBJ variables corresponding to indexes $x_k$ and $y_k$ respectively. Keyword $\text{bop}$ (or $\text{bops}$) is used to declare observation or action operator (or operators), while $\text{op}$ (or $\text{ops}$) for conventional operator (or operators) or hidden constant (or hidden constants).

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

\begin{equation}
\text{eq o init}(X_1, \ldots, X_m) = f(X_1, \ldots, X_m). \tag{1}
\end{equation}

Where, $f(X_1, \ldots, X_m)$ 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 $X_1, \ldots, X_m$ is the one that $f(X_1, \ldots, X_m)$ returns.

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

\begin{equation}
\text{eq c-t}(S, Y_1, \ldots, Y_n, X_1, \ldots, X_m) = \begin{cases} 
\text{e-t}(S, Y_1, \ldots, Y_n, X_1, \ldots, X_m) & \text{if } c-t(S, Y_1, \ldots, Y_n) \\
\text{t-t}(S', Y_1, \ldots, Y_n, X_1, \ldots, X_m) & \text{if } \neg c-t(S, Y_1, \ldots, Y_n)
\end{cases} \tag{2}
\end{equation}

Where, $S$ is a variable of $H$ that represents the state $v$; term $\text{t-t}(S, Y_1, \ldots, Y_n, X_1, \ldots, X_m)$ represents the successor state of $S$; $\text{e-t}(S, Y_1, \ldots, Y_n, X_1, \ldots, X_m)$ is a term that represents the value returned by $o_{x_1:D_{x_1}, \ldots, x_m:D_{x_m}}$ in the successor state and the condition part $c-t(S, Y_1, \ldots, Y_n)$ 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 $c_{t_{y_1:D_{y_1}, \ldots, y_n:D_{y_n}}}(v)$ always holds in any state $v$ or the value returned by $o_{x_1:D_{x_1}, \ldots, x_m:D_{x_m}}$ 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{equation}
\text{eq o t}(S, Y_1, \ldots, Y_n, X_1, \ldots, X_m) = \begin{cases} 
\text{e-t}(S, Y_1, \ldots, Y_n, X_1, \ldots, X_m) & \text{if } c-t(S, Y_1, \ldots, Y_n) \\
\text{t-t}(S', Y_1, \ldots, Y_n, X_1, \ldots, X_m) & \text{if } \neg c-t(S, Y_1, \ldots, Y_n)
\end{cases}
\end{equation}

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

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

\begin{equation}
\text{bceq t}(S, Y_1, \ldots, Y_n) = S \begin{cases} 
\text{if } c-t(S, Y_1, \ldots, Y_n) = S & \text{if not } c-t(S, Y_1, \ldots, Y_n)
\end{cases}
\end{equation}
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.

### 2.2.1. 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 this 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 and conditional rules; t-tran denotes the labels of rules and t_Y1,...,Y_n_ is a constant of type D_1...D_n for k = 1,...,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 instead of variables in OTS/CafeOBJ specifications (see Section 2.1). An arbitrary state has [D_1] x ... x [D_n] possible successor states w.r.t. the transition t_Y1,...,Y_n_. Therefore, given a transition t_Y1,...,Y_n_, we need to define [D_1] x ... x [D_n] 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 Pram : Val”, where Op is an operator that corresponds to an observer; Pram 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:

\[
\text{crl [t-tran]} : (Op \text{ Prams} : \text{Val1}) \ldots \Rightarrow \text{(Op \text{ Prams} : \text{Val2})} \ldots \\
\text{if } \text{c-t'(Val1,...) .}
\]

Where, c-t' is an auto-generated term corresponding to the effective condition c-t(Y1,...,Yn), 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 Pram changes from Val1 to Val2. Observable values that are changed by a transition and 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.
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.

$$\text{sort OTSDecl OTSDeclSet OTSModule}.$$
$$\text{subsort OTSDecl < OTSDeclSet}.$$
$$\text{op bop: _\rightarrow_ : Qid TypeList Type \rightarrow OTSDecl [ctor format(g o g o g o o o o)]}.$$
$$\text{op bops: _\rightarrow_ : QidList TypeList Type \rightarrow OTSDecl [ctor format(g o g o o o o)]}.$$
$$\text{op mod*: _\rightarrow_ : Header ImportList HiddenSortDecl OpDeclSet OTSDeclSet EquationSet \rightarrow OTSModule [ctor gather(4 4 4 4 4 4 4)]}.$$
$$\text{op emptyOTSModule: \rightarrow OTSModule}.$$

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

However, Maude cannot 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 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 \( V_{i_1}, \ldots, V_{i_n} \). We use variables \( Y_1, \ldots, Y_n \) 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 \( t(S, Y_1, \ldots, Y_n) \). Consequently, the rewrite rule for the transition is in the following form:

\[
\text{cr1 [t-tran] S} \Rightarrow t(S, Y_1, \ldots, Y_n) \quad \text{(II)}
\]

if \( c-t(S, Y_1, \ldots, Y_n) \).

However, the rewrite rule becomes inexecutable because
variables that appear at right-hand side and condition part do not appear at left-hand side. To make rewrite rules executable, we need to guarantee that 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 \( \text{OTSState} \) that is defined as follows:

\[
\text{sort OTSSState}.
\]

\[
\text{op } \{ \ldots \} : \text{H Type ... Type } \rightarrow \text{OTSState [ctor]}.
\]

The constructor consists of two parts. The first part is a state of the sort \( \text{t} \), which corresponds to the hidden sort in the \( \text{OTS/CafeOBJ} \) 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 \( S_{V_y} \) stands for a sort that represents the set of constants of sort \( V_y \). Assume that the set of action operators in the \( \text{OTS/CafeOBJ} \) specification is denoted by \( A \). For any action operator \( t \in A \), 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 \( A \). There may be some redundant sorts in the list. For instance, assume there are two action operators \( t_1, t_2 \) and sort \( V \in P_1 \), \( V \in P_2 \). 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 \( t \in A \), 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 \( \text{buildTypeList} \) 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 \( \text{combineTypeList} \) combines two lists of sorts \( \text{TyL} \) and \( \text{TyL'} \) together with two auxiliary functions \( \text{filter} \) and \( \text{cover} \). Function \( \text{filter} \) gets the common part of \( \text{TyL} \) and \( \text{TyL'} \) and \( \text{cover} \) gets those sorts that only appear either in \( \text{TyL} \) or in \( \text{TyL'} \). The two parts are combined together as the returned value of \( \text{combineTypeList} \).

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

\[
\text{bop } t_1 : H V_1 V_2 \rightarrow H
\]

\[
\text{bop } t_2 : H V_1 V_2 V_3 \rightarrow H
\]

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 \( \text{OTS/CafeOBJ} \) specification. Terms at left-hand side and right-hand side of rewrite rules are of the sort \( \text{OTSState} \).

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

\[
[S; V_1 \oplus t_{11}; V_1 \oplus \text{set} \oplus t_{11}; V_1 \oplus \text{set}, V_2 \oplus t_{11}; V_2 \oplus \text{set} \oplus t_{11}; V_2 \oplus \text{set}, V_3 \oplus t_{11}; V_3 \oplus \text{set} \oplus t_{11}; V_3 \oplus \text{set}, V_3 \oplus t_{12}; V_3 \oplus \text{set} \oplus t_{12}; V_3 \oplus \text{set} ]
\]

A variable \( X \) of sort \( S \) can be declared on the fly like \( X: S \). Note that \( V_1 \text{set} \) 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 "\( t \)" 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 \( V_3 \oplus t_{11} \) and \( V_3 \oplus t_{12} \).

Function \( \text{buildRHS} \) takes the same parameters as function \( \text{buildLHS} \). It returns a state of \( \text{OTSState} \) that represents all possible successor states of the state at the action operator 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 \( t_1 \{ S, V_1 \oplus t_{11}, V_2 \oplus t_{11} \} \) is a successor state of \( S \) w.r.t. \( t_1 \). Because variables \( V_1 \oplus t_{11} \) and \( V_2 \oplus t_{11} \) can be replaced with all possible constants \( o \)-sort \( V_1 \) and \( V_2 \), \( t_1 \{ S, V_1 \oplus t_{11}, V_2 \oplus t_{11} \} \) represents all possible
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 $\[ \text{init} ; ((v_{11}, v_{12}), (v_{21}, v_{22}), (v_{31}, v_{32}), (v_{31}, v_{32})) \]$ 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 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@t1:V1 V1Set@t1:V1Set, V2@t1:V2 V2Set@t1:V2Set, V3@t1:V3 V3Set@t1:V3Set, V3@t1:V3 V3Set@t1: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] 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. The returned value of buildSortListSet is assigned to variable SS and added to the target system module. Function buildListConnector declares an operator "\_\_" for each sort $V$ 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
in CafeOBJ like \("bop \circ : \text{Ho} \rightarrow \text{Vo} \rightarrow \text{V}\). Its corresponding declaration as an observable value in Maude is \("\text{op o...o}: \text{V}_{a1}\ldots \text{V}_{an} \rightarrow \text{V}\". Its commutativity and associativity of the operator means that there is no particular order for values in an explicit state.

In a state, an observer \(\sigma_{a2\ldots am}\) 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 \(\text{V}_k (k = 01, \ldots, om)\) of \(\sigma_{a2\ldots am}\), we give a finite set of \(\text{V}_k\) that \(\text{FTV}_k\) in 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 \(\text{V}_k\), before loading them into the translator.

Then, we can get \(\text{FTV}_{a1}[\text{x}_{1} \ldots \text{x}_{m}]\) observable values w.r.t. \(\sigma_{a2\ldots am}\) in each state \(S\). Observable values are in form \("\text{o x}_{1} \ldots \text{x}_{m} : \text{V}\"\), where \(\text{V}\) is a term of sort \(\text{V}\) that depends on \(\text{x}_{1}, \ldots, \text{x}_{m}\). These observable values are parts of state \(S\) w.r.t. \(\sigma_{a2\ldots am}\). 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 \(\sigma_{a2\ldots am}\), we get the term \(f(x_{1}, \ldots, x_{m})\) as the value that \(\sigma_{a2\ldots am}\) returns with parameters \(x_{1}, \ldots, x_{m}\), by replacing variables \(x_{1}, \ldots, x_{m}\) with constants \(a_{1}, \ldots, a_{m}\) in (I). An observable value can be constructed as \("\text{o x}_{1} \ldots \text{x}_{m} : f(x_{1}, \ldots, x_{m})\"\). Similarly, we can get other observable values for \(\sigma_{a2\ldots am}\) 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:

\[\text{eq init'} = \text{o x}_{1} \ldots \text{x}_{m} : f(x_{1}, \ldots, x_{m}) \ldots\]

Where, \(\text{init'}\) 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 buildInitialState 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 the constants of parameter sorts; OTSModule is the input OTS module; SModule in parameters is a auto-generated system module to which the initial state will be added and Module 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 \("\text{eq o(init, x}_{1}, \ldots, x_{m}) = f(x_{1}, \ldots, x_{m})\"\). 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 \("\text{o x}_{1} \ldots \text{x}_{m} : f(x_{1}, \ldots, x_{m})\"\) 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 OTSSate 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 \(\text{V}_{o1}, \ldots, \text{V}_{on}\) 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 \(\sigma_{a2\ldots am}\) w.r.t. parameters \(x_{1}, \ldots, x_{m}\). The corresponding observable value is \("\text{o x}_{1} \ldots \text{x}_{m} : x_{V}\"\). 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 \(\sigma_{a2\ldots am}\) 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 \("\text{o x}_{1} \ldots \text{x}_{m} : x_{V}\"\) can be removed from the target rewrite rule that specifies the transition \(t_{y_{1}, \ldots, y_{n}}\) with the related parameters.
To get those observable values that are changed by $t_{y_1,...,y_n}$ with constants $Y_1,...,Y_n$, we get the relevant equations that specify the transition $t_{y_1,...,y_n}$ in form of (2) and replace the parameters of $t_{y_1,...,y_n}$ with constants $Y_1,...,Y_n$. We get a collection of equations in the following form:

$$
\text{ceq } c(t(Y_1, ..., Y_n), X_1, ..., X_m) = e(t(Y_1, ..., Y_n, X_1, ..., X_m) \quad \text{if } c(t(Y_1, ..., Y_n)) = 1
$$

(3)

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

$$
\text{eq } o(t(Y_1, ..., Y_n), X_1, ..., X_m) = e(t(Y_1, ..., Y_n, X_1, ..., X_m) \quad \text{if } c(t(Y_1, ..., Y_n)) = 1
$$

(4)

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

$$
\text{ceq } c(t(Y_1, ..., Y_n), X_1, ..., X_m) = e(t(Y_1, ..., Y_n, X_1, ..., X_m) \quad \text{if } c(t(Y_1, ..., Y_n)) = 1
$$

(5)

$$
\text{eq } o(t(Y_1, ..., Y_n), X_1, ..., X_m) = e(t(Y_1, ..., Y_n, X_1, ..., X_m) \quad \text{if } c(t(Y_1, ..., Y_n)) = 1
$$

(6)

For each equation, we can build an observable value by introducing a new variable like $o(X_1, ..., X_m)$. 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 $o_{x_1,...,x_m}$ with parameters $x_1,...,x_m$, according to (5), the observable value is changed into $e(t(Y_1, ..., Y_n, X_1, ..., X_m)$ by the transition $t_{y_1,...,y_n}$ with constants $Y_1,...,Y_n$. We replace the term $o(S(Y_1, ..., Y_n, X_1, ..., X_m)$ with the variable $X_i'$ in term $e(t(Y_1, ..., Y_n, X_1, ..., X_m)$ and get a term like $e(t(Y_1, ..., Y_n, X_1', ..., X_m))$ which is the observed value of $o_{x_1,...,x_m}$ with parameters $x_1,...,x_m$ in the successor state. The observable value $o(X_1, ..., X_m)$ 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 $t_{y_1,...,y_n}$ with parameters $Y_1,...,Y_n$ does not change values that are observed by $o_{x_1,...,x_m}$. 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 $t_{y_1,...,y_n}$ with parameters $Y_1,...,Y_n$. Similarly, successor states of the arbitrary state with the transition $t_{y_1,...,y_n}$ and other different parameters 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 $t_{y_1,...,y_n}$ and parameters $Y_1,...,Y_n$, a condition part is also needed for the target conditional rewrite rule.

In the OTS/CafeOBJ specification, there is an effective condition $c(t_{y_1,...,y_n}$ declared and a related equation defined for the term $c(t(Y_1, ..., Y_n))$. 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 $t_{y_1,...,y_n}$ with parameters $Y_1,...,Y_n$. The expression is represented by the term $c'(X_1, ..., X_n)$, where $X_1,...,X_n$ are variables introduced for observable values and $n' = |FT_{V_{out}|} 	imes |FT_{V_{out}|} 	imes |FT_{V_{out}|} 	imes ...$.

With the arbitrary explicit state, its successor state and the condition term, a conditional rewrite rewrite is defined for the transition $t_{y_1,...,y_n}$ with parameters $Y_1,...,Y_n$. 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 its own two commands, namely conv2exp and conv2imp 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 Fid 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 Fid 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(INF) pr(PROCESS)
* [Sys]
op init : -> Sys
bop c-down : Sys Pid -> Bool
bop semaphore : Sys -> Int
ops c-down up : Sys Pid -> Sys
eq using(init, Xll) = false
ceq semaphore(init) = 1
ceq using(down(S:Sys, Xll), Xll) = if Xll = Xll then true else using(S, Xll) fi
bceq up(S, Y2l) = S if not c-up(S, Y2l)
ceq semaphore(up(S, Y2l)) = semaphore(S) + 1 if c-up(S, Y2l)
bceq down(S, Yll) = S if not c-down(S, Yll)

cq using(S, Xll) = if Xll = Xll then false else using(S, Xll) fi
ceq semaphore(S) = 0 .
bceq down(S, Yll) = S if not c-down(S, Yll)

cq c-down(S, Yll) = false
ceq semaphore(S) = 0 .
bceq down(S, Yll) = S if not c-down(S, Yll)

cq using(S, Xll) = if Xll = Xll then false else using(S, Xll) fi
ceq semaphore(S) = 0 .
bceq down(S, Yll) = S if not c-down(S, Yll)

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 declaration 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 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 shown 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 PidSet@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 first 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
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 Francisco Durán for his patient instruction on the implementation of Full Maude via mails.

References


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

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

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')].)


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.

c eq 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, EqS) =
buildInitialStateAux1(BOPL, OP, TL, T')
if BOP2OP := getBOP2OP(BOPL, OP) \ BOP2OP =/= nil.

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

Appendix C
Key functions to construct transitions between explicit states

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

c eq convertEq2Rl(BOPL, TCL, (OTS OTS), EqS) =
convertEq2RlAux(BOPL, TCL, OTS, EqS) convertEq2RlAux(BOPL, TCL, OTS, OTS, EqS)
if OTS =/= none.

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

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

eq convertEq2RlAux1(BOPL, OP[TL], (eq OP'[OP[T, TL], TL'] = T' if EqC [Attr] .) 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].