A Lightweight Integration of Theorem Proving and Model Checking for System Verification

Weiqiang Kong[1], Kazuhiro Ogata[1,2], Takahiro Seino[1], and Kokichi Futatsugi[1]

[1] Japan Advanced Institute of Science and Technology (JAIST)
1-1, Asahidai, Nomi, Ishikawa 923-1292, Japan
{weiqiang, t-seino, kokichi}@jaist.ac.jp, Phone: +81(90)3763-9683
[2] NEC Software Hokuriku, Ltd.
1 Anyoji, Tsurugi, Ishikawa 920-2141, Japan
ogatak@acm.org

Abstract

Theorem proving and model checking are known as two formal verification techniques that have complementary features. In this paper, we describe a lightweight integration of the two techniques by a translation from theorem proving formalism to model checking formalism, and then treating model checking as part of the decision procedure. In the translation, system and property specifications defined for a theorem prover can be automatically translated to specifications feedable to a model checker after a simple data abstraction. The main aim of this integration is to provide the theorem prover with automatic counter-example generating capability, thus to be able to find "bugs" in the early stage of theorem proving and ease the hard-work of doing theorem proving. A case study is used to demonstrate how this translation works and what the verification flow is when using this integration to do system verification.

1. Introduction

Theorem proving and model checking are known as two formal verification techniques that have complementary features. The main aspects considered in the comparisons of these two techniques include: (1) State space can be handled (infinite vs. finite), (2) Automation of verification procedure (limited automation vs. fully automation) and (3) Counter-example generating capability (no automatic counter-example vs. automatic counter-example). To take advantages of each of the two verification techniques, a number of researches have been reported, proposing different kinds of integration, such as [1, 3, 10, 11, 12].

In contrast to proposing a general-purpose integration as discussed in the above mentioned works, we consider in this paper a lightweight integration of the two formal verification techniques by a translation from theorem proving formalism to model checking formalism, and then treating model checking as part of the decision procedure. The main aim of this integration is to provide the theorem prover with automatic counter-example generating capability, thus to be able to find "bugs" in the early stage of theorem proving and ease the hard-work of doing theorem proving.

Theorem proving is a general formal verification technique that can be used to verify complex and infinite-state systems; furthermore, doing theorem proving may help users have more insight and understanding of the system to be verified[1]. However, if a property fails to hold, it is difficult for unexperienced users to extract enough valuable information from the verification result returned by a theorem prover. Therefore users must try to determine whether the fault lies with the system and property specifications or with the failed proof [12]. Besides, as discussed in [2] which gives an in-depth comparison of the two techniques in the hands of experienced users, considerable time is used during theorem proving to "discover, formalize and prove auxiliary system invariants, which are required to prove the property of interest". We believe that if a counter-example can be generated automatically, which shows a sequence of the system's behaviors that violates the property: (1) on one hand, by analyzing the counter-example, it will become easier to find out the reason for the failure, therefore pinpoint errors; and (2) on the other hand, before putting efforts trying to prove the newly founded invariant that is possibly correct/incorrect, we can benefit from firstly model checking the invariant and see whether a counter-example arises. In

[1]This is also the reason that we use theorem proving to do the main verification work rather than directly using abstraction and model checking.

case that a counter-example arises, this will serve as an indication of discarding the invariant and switching to finding alternative ones. As to the case that the verification result is *true*, this result can serve as a weak justification showing that there might exist a possible proof of this invariant.

In our specific lightweight integration, system specification and property specification defined using the OTS/CafeOBJ method [9] are automatically translated to corresponding parts in the OTS/Maude method [7] after a simple data abstraction. And then, the translated properties can be checked against the translated model using the Maude LTL model checker [4, 6]. The integration is considered to be "lightweight" because of the following reasons: Firstly (for something good), the formalisms of both the OTS/CafeOBJ method (for theorem proving) and the OTS/Maude method (for model checking) are quite similar (both based on equations). We think, on one hand, equations are easy to understand and use for unexperienced users; and on the other hand, similar formalisms will alleviate the burden for users to learn two different formalisms as discussed in some other integrations. Secondly (for something bad), the data abstraction method we used to link an infinite system with its model checkable finite version may not preserve soundness [10] (the abstracted version may have some property that does not hold in the original version). Instead of some non-trivial abstraction methods which are property-preserving, we employed a simple data abstraction by means of reducing the infinite domains of variables to some concrete values. For example, the number of processes in a mutual exclusion algorithm is reduced from infinite to 2. As discussed in [10], such simple data abstraction is effective when we aims to exposing bugs.

Here we use a mutual exclusion algorithm using a queue to demonstrate how the translation is done by a translator – Cafe2Maude. We also present what the verification flow is when using our integration to do system verification.

Organization. Section 2 describes the OTS/CafeOBJ method, focusing on how to write OTS in CafeOBJ for system specification and how to write invariants for property specification. Section 3 describes how the translator – Cafe2Maude works. In this section, system specification translation, and property translation together with the data abstraction are mainly introduced. Section 4 describes the case study to demonstrate how Cafe2Maude works and what the verification flow is. Section 5 concludes the paper.

2. The OTS/CafeOBJ Method

We have been successfully applying the OTS/CafeOBJ method [9] to modeling, specification and verification of distributed systems such as security protocols [8]. In the OTS/CafeOBJ method, a system is modeled as an observational transition system (OTS), which is a transition system that can be straightforwardly written in terms of equations; and OTS is written in CafeOBJ[2] [5], an algebraic specification language. Desired properties of the OTS can then be verified by writing proofs (called proof scores) in CafeOBJ and executing the proof scores with the CafeOBJ system.

Assume that there exists a universal state space called Υ. We also assume that data types used, including the equivalence relation (denoted by =) for each data type, have been defined in advance. An OTS $\mathcal{S}$ consists of $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ where:

- $\mathcal{O}$: A set of observers. Each $o \in \mathcal{O}$ is a function $o : \Upsilon \rightarrow D$, where D is a data type and may differ from observer to observer. Given an OTS $\mathcal{S}$ and two states $\upsilon_1, \upsilon_2 \in \Upsilon$, the equivalence (denoted by $\upsilon_1 =_{\mathcal{S}} \upsilon_2$) between them wrt $\mathcal{S}$ is defined as $\forall o \in \mathcal{O}, o(\upsilon_1) = o(\upsilon_2)$.
- $\mathcal{I}$: The set of initial states such that $\mathcal{I} \subseteq \Upsilon$.
- $\mathcal{T}$: A set of conditional transitions. Each $\tau \in \mathcal{T}$ is a function $\tau : \Upsilon \rightarrow \Upsilon$, provided that $\tau(\upsilon_1) =_{\mathcal{S}} \tau(\upsilon_2)$ for each $[\upsilon] \in \Upsilon / =_{\mathcal{S}}$ and each $\upsilon_1, \upsilon_2 \in [\upsilon]$. $\tau(\upsilon)$ is called the successor state of $\upsilon \in \Upsilon$ wrt τ. The condition c_τ of τ is called the effective condition. For each $\upsilon \in \Upsilon$ such that $\neg c_\tau(\upsilon)$, $\upsilon =_{\mathcal{S}} \tau(\upsilon)$.

Observers and transitions may be parameterized. Generally, observers and transitions are denoted by $o_{i_1,\ldots,i_m}$ and $\tau_{j_1,\ldots,j_n}$, respectively, provided that $m, n \geq 0$ and there exist data types D_k such that $k \in D_k\,(k = i_1, \ldots, i_m, j_1, \ldots, j_n)$.

In the OTS/CafeOBJ method, an OTS is described in CafeOBJ which can be used to specify abstract machines as well as abstract data types. A visible sort denotes an abstract data type, and a hidden sort denotes the state space of an abstract machine. There are two kinds of operators in hidden sorts: action and observation operators. An action operator can change a state of an abstract machine; only observation operators can be used to observe the inside of an abstract machine. Declarations of observation and action operators start with `bop` or `bops`, and those of other operators with `op` or `ops`. Operators are defined in equations. Declarations of equations start with `eq`, and those of conditional equations with `ceq`. The CafeOBJ system rewrites a given term by regarding equations as left-to-right rewrite rules.

The universal state space Υ is denoted by a hidden sort, say *H*. An observer $o_{i_1,\ldots,i_m} \in \mathcal{O}$ is denoted by a CafeOBJ observation operator. We assume that there exist visible sorts V_k and *V* denoting D_k and *D*, where $k = i_1, \ldots, i_m$. The CafeOBJ observation operator is declared as `bop` $o : H\ V_{i_1} \ldots V_{i_m}$ `->` V .

A transition $\tau_{j_1,\ldots,j_n} \in \mathcal{T}$ is denoted by a CafeOBJ action operator. We assume that there exists a visible sort V_k denoting D_k, where $k = j_1, \ldots, j_n$. The CafeOBJ

[2]See www.ldl.jaist.ac.jp/cafeobj/.

action operator is declared as `bop` $\tau : H\ V_{j_1} \ldots V_{j_n}$ `->` H. $\tau_{j_1,\ldots,j_n}$ may change the value returned by $o_{i_1,\ldots,i_m}$ if it is applied in a state v such that $c_{\tau_{j_1,\ldots,j_n}}(v)$, which can be written generally as follows:

```
ceq $o(\tau(S, X_{j_1}, \ldots, X_{j_n}), X_{i_1}, \ldots, X_{i_m})$
        $= e\text{-}\tau(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})$
          if $c\text{-}\tau(S, X_{j_1}, \ldots, X_{j_n})$ .
```

S is a CafeOBJ variable for H and X_k is a CafeOBJ variable of V_k, where $k = j_1, \ldots, j_n$. $\tau(S, X_{j_1}, \ldots, X_{j_n})$ denotes the successor state of S wrt $\tau_{j_1,\ldots,j_n}$. $e\text{-}\tau(S, X_{j_1}, \ldots, X_{j_n}, X_{i_1}, \ldots, X_{i_m})$ denotes the value returned by $o_{i_1,\ldots,i_m}$ in the successor state. $c\text{-}\tau(S, X_{j_1}, \ldots, X_{j_n})$ denotes the effective condition $c_{\tau_{j_1,\ldots,j_n}}$. $\tau_{j_1,\ldots,j_n}$ changes nothing if it is applied in a state v such that $\neg c_{\tau_{j_1,\ldots,j_n}}(v)$.

The properties considered in this paper are invariants. Since how to prove these properties in the OTS/CafeOBJ method is not related to the translation for our integration, we only describe here how to specify an invariant of S denoted by a predicate p in the OTS/CafeOBJ method. Let $x_1, \ldots, x_m$ whose types are $D_1, \ldots, D_m$ be all free variables in p except for v whose type is Υ. The operator denoting p and its defining equation in a module `INV` (`INV` imports the module where S is written, and the module writing S is called OTS module) are generally as follows:

```
op $inv : H\ V_1 \ldots V_m$ -> Bool
eq $inv(S, X_1, \ldots, X_m) = p(S, X_1, \ldots, X_m)$ .
```

where V_k ($k = 1, \ldots, m$) is a visible sort denoting D_k, and X_k is a CafeOBJ variable whose sort is V_k. $p(S, X_1, \ldots, X_m)$ is a CafeOBJ term denoting p.

3. Translation

We have designed and implemented a translator – Cafe2Maude, which can automatically translate the formalism of the OTS/CafeOBJ method to the formalism of the OTS/Maude method. Maude [4, 6] is a specification language which has model checking facilities whose performance is comparable to SPIN. The translation from OTS/CafeOBJ to OTS/Maude is mainly based on [7], in which we described the OTS/Maude method of specifying and model checking OTS using Maude.

Basic units of CafeOBJ (Maude) specifications are modules. An OTS/CafeOBJ (OTS/Maude) specification consists of a list of modules such that one module specifies an OTS, some specify data types used in the OTS module, and one (several in OTS/Maude) specifies the properties.

Let M be a CafeOBJ module and L be a list of CafeOBJ modules; $Cdmod$, $Comod$ and $Cinv$ be the types of CafeOBJ modules that specify data types, OTS and invariant properties, respectively; and $ListOfCmod$ and $ListOfMmod$ be the types of lists of CafeOBJ modules and Maude modules, respectively. The translator can be formalized by the function T, such that:

```
$T$ : $ListOfCmod$ -> $ListOfMmod$;
$T(nil) = nil$;
$T(M\ L) =$ if $M$ : $Cdmod$
            then $T_d(M)\ T(L)$
            else if $M$ : $Comod$ then $T_o(M)\ T(L)$
                 else $T_i(M)\ T(L)$;
```

where function T_d takes a CafeOBJ data type module and generates a corresponding Maude functional module that specifies the data type; function T_o takes a CafeOBJ OTS module and generates a corresponding Maude system module that specifies the OTS (therefore also called Maude OTS module); and function T_i takes a CafeOBJ invariant defining module and generates one Maude state predicate defining module and one Maude linear temporal logic (LTL) property defining modules. nil denotes an empty list of modules. Next, we introduce the three functions in turn.

3.1 Translation of CafeOBJ Data Type Modules

The translation from CafeOBJ data type modules to Maude functional modules is very straightforward, which involves only changes of the manner of expression. An example of such translation is given as follows, which translates the *module declaration* between two notations:

```
mod $module\_name$ "{"                      fmod $module\_name$ is
...                         $->_{T_d}$       ...
"}"                                         endfm
```

Other elements of CafeOBJ data type modules such as operator declarations are translated with syntactic changes in a similar way.

3.2 Translation of CafeOBJ OTS Module

Generally, a CafeOBJ OTS module consists of two parts: a signature and a set of equations. A signature consists of declarations of a hidden sort, observation and action operators. Equations can be classified into equations defining initial values of observation operators and equations defining action operators. Next, we describe the translations of these basic elements by the function T_o.

Signature: Declaration of a Hidden Sort

A hidden sort, say Sys, in the CafeOBJ OTS module denotes the universal state space Υ, which in practice denotes the state space of the system under consideration. The hidden sort Sys is considered as a normal sort of Maude OTS module (we use `Sys` to denote this sort). Besides, two additional sorts `OValue` and `TRule` are declared as subsorts of `Sys`, which denote the sorts of observation and action operators, respectively. By doing this, we can define that a snapshot (state) of an OTS $\mathcal{S}$ is a multi-set, or a bag of observers and transitions. The declaration of a hidden sort can be translated by function T_o as follows:

$T_o(*[Sys]*)$ =

```
subsort OValue TRule < Sys.
op none :  -> Sys .
op __ : Sys Sys -> Sys [assoc comm id: none] .
```

where $*[Sys]*$ is the declaration of Sys. The translation result is the actual Maude specification, and the last two formulas are declarations of constructors of bags. The three key flags `assoc, comm` and `id` denote the equational attributes of associativity, commutativity and identity. Generally, a snapshot (state) of $\mathcal{S}$ is in the following form:

ovalue-1 ... *ovalue-M* *trule*-1 ... *trule-N*

where *ovalue-i* $(i = 1, \ldots, M)$ is a term denoting an observer, and *trule-j* $(j = 1, \ldots, N)$ is a term denoting a transition.

Signature: Declarations of Observation and Action Operators

We assume that all required data types are predefined and there exist sorts corresponding to these data types. The declaration of a CafeOBJ observation operator can be translated by function T_o to Maude one as follows:

T_o(`bop` $o : Sys\ V_{i_1} \ldots V_{i_m}$ `->` V) =

`if` $m > 0$ `then`
`op` $(o$`[_,...,_]` `:` `_`$)$ `:` $V_{i_1} \ldots V_{i_m}\ V$ `-> OValue .`
`else`
`op` $(o$ `: _`$)$ `:` V `-> OValue .`

The declaration of a CafeOBJ action operator can be translated by function T_o to Maude one as follows:

T_o(`bop` $\tau : Sys\ V_{j_1} \ldots V_{j_n}$ `->` Sys) =

`op` τ `:` $V_{j_1} \ldots V_{j_n}$ `-> TRule .`

Equations Defining Initial Values of Observation Operators

Actually, the equations defining initial values of observation operators, together with the declarations of transitions and data abstraction, are used to define the initial state of an OTS S in another Maude module instead of the Maude system (OTS) module. So we suspend the description of T_o for initial state translation until subsection 3.3.2.

Equations Defining Action Operators

Equations defining action operators describe the state changes of an OTS $\mathcal{S}$. We assume that observers needed and affected by the execution of the transition $\tau_{j_1,\ldots,j_n}$ are $o^1_{i^1_1,\ldots,i^1_{m_1}}, \ldots, o^l_{i^l_1,\ldots,i^l_{m_l}}$. The CafeOBJ equations defining the action operators are $ceq_1, \ldots, ceq_l$, where ceq_t $(t = 1, \ldots, l)$ is generally in the following form:

`ceq` $o^t\ (\tau\ (Sys, X_{j_1}, \ldots, X_{j_n}), X_{i^t_1}, \ldots, X_{i^t_{m_t}})$ = X_t
  `if` $c\text{-}\tau\ (Sys, X_{j_1}, \ldots, X_{j_n})$.

$c\text{-}\tau(Sys, X_{j_1}, \ldots, X_{j_n})$ denotes the effective condition of transition $\tau_{j_1,\ldots,j_n}$. X_k $(k = j_1, \ldots, j_n, i^t_1, \ldots, i^t_{m_t}, t)$ is a variable or a term for the intended sort. The set of equations $ceq_1, \ldots, ceq_l$ is translated to a Maude rewrite rule by function T_o as follows:

$T_o(ceq_1, \ldots, ceq_l)$ =
`crl[rule-`τ`] :`
  $\tau(X_{j_1}, \ldots, X_{j_n})$
  $(o^1[X_{i^1_1}, \ldots, X_{i^1_{m_1}}] : X_1) \ldots (o^l[X_{i^l_1}, \ldots, X_{i^l_{m_l}}] : X_l)$
  `=>`
  $\tau(X_{j_1}, \ldots, X_{j_n})$
  $(o^1[X_{i^1_1}, \ldots, X_{i^1_{m_1}}] : X_1^{'}) \ldots (o^l[X_{i^l_1}, \ldots, X_{i^l_{m_l}}] : X_l^{'})$
  `if` $c\text{-}\tau(X_{j_1}, \ldots, X_{j_n}, X_{i^1_1}, \ldots, X_{i^1_{m_1}}, X_1, \ldots, X_{i^l_1},$
    $\ldots, X_{i^l_{m_l}}, X_l)$.

`rule-`τ is the label of the rewrite rule. In the translation, the name of the action operator τ is used to denote this label. $X_k^{'}(k = 1, \ldots, l)$ denotes the value returned by observer $o^k_{i^k_1,\ldots,i^k_{m_k}}$ in the successor state with respect to $\tau_{j_1,\ldots,j_n}$. Note that for the situation that the value returned by $o^k_{i^k_1,\ldots,i^k_{m_k}}$ is not affected by transition $\tau_{j_1,\ldots,j_n}$, $X_k^{'}$ equals X_k.

3.3 Translation of CafeOBJ Property Module

To make the description of the property translation by function T_i more clear, we firstly introduce how to model check OTS using Maude [7], by which we set up the context of the property translation.

3.3.1 How to Model Check OTS using Maude

We assume that an OTS is written in Maude as a system module whose name is `SYSTEM`. We first define state predicates with which propositional LTL formulas denoting properties are described. Such state predicates are declared in a module, say `SYSTEM-PREDS`, which looks like:

```
mod SYSTEM-PREDS is
  protecting SYSTEM .
  including SATISFACTION .
  subsort Sys < State .
  ...
endm
```

where the dots $\cdots$ indicate the part in which the syntax and semantics of state predicates are specified.

In the Maude module `SATISFACTION` (included in a Maude file `model-checker.maude`), the module `LTL` (also included in the file `model-checker.maude`) which describes propositional linear temporal logic (`LTL`) is imported, the sort `State` that denotes states of a system under consideration is declared and the following satisfaction operator is declared:

```
op _|=_ : State Formula ~> Bool .
```

The sort `Formula` is declared in the module `LTL`, denoting propositional LTL formulas. The operator is used to define state predicates. A state predicate denoted by a term *pred* holds in a state denoted by *state* is defined as follows:

eq *state* |= *pred* = true .

Generally, *state* is in the following form:

ovalue-1 … ovalue-M S

where $ovalue\text{-}i$ $(i = 1, \ldots, M)$ is a term for `OValue` and S is a variable for `Sys`.

We next define propositional LTL formulas denoting properties to be checked for the OTS and also initial states of the OTS. Such formulas and initial states are described in a module, say `SYSTEM-CHECK`, which looks like:

```
mod SYSTEM-CHECK is
    including SYSTEM-PREDS    .
    including MODEL-CHECKER   .
    ...
endm
```

where the dots $\cdots$ indicate the part in which operators denoting propositional LTL formulas to be checked for the OTS and initial states of the OTS, and the corresponding equations are declared.

In the module `MODEL-CHECKER` (included in the file `model-checker.maude`), the operator `modelCheck` is declared, which takes two arguments denoting an initial state and a propositional LTL formula, and returns the result of the model checking.

Propositional LTL formulas are constructed of state predicates declared in `SYSTEM-PREDS`, Boolean connectives and temporal operators declared in `LTL`. Among temporal operators are "Eventually" denoted by `<>`, "Henceforth" (or "Always") denoted by `[]` and "Leads-to" denoted by `|->`.

The term denoting an initial state is generally in the following form:

ovalue-1 … ovalue-M trule-1 … trule-N

where $ovalue\text{-}i$ $(i = 1, \ldots, M)$ is a term for `OValue`, and $trule\text{-}i$ $(i = 1, \ldots, N)$ is a term for `TRule`.

Let $init$ be a term denoting an initial state and $property$ be a term denoting a propositional LTL formula. We model check, in the Maude environment, that if $property$ holds at state $init$ as follows:

red modelCheck$(init, property)$.

3.3.2 Invariant Property Translation

We are now ready to describe property translation based on previous preliminary knowledge. Given a CafeOBJ invariant defining module, what we need to do is to firstly construct a Maude module that defines state predicates, and secondly construct another Maude module that defines propositional LTL formulas denoting properties using these state predicates. Note that a CafeOBJ invariant for property consists of a set of predicates and logical connectives, and each predicate of the set can be represented as a state predicate of the system. So our strategy to property translation is: (1) classify predicates in a given invariant; (2) declare state predicate for each of the predicates according to its kind; (3) replace the predicates in the invariant with corresponding declared state predicates, and also replace logical connectives with corresponding Maude ones; (4) to simulate the semantics of the invariant that a property holds in any reachable state, we add a temporal operator "Always" `[]` in front of the replaced invariant. Thus we get a Maude propositional LTL formula denoting property corresponding to the invariant.

In the following, we classify the predicates in a given invariant into several kinds, and for each kind of predicates we declare corresponding state predicates in the Maude `SYSTEM-PREDS` module (suppose the name of the Maude module that specifies the OTS is `SYSTEM`). The classification method is based on an assumption: each predicate has at most one observation operator, while predicates with two or more observation operators should be written separately. Such as $o_1(S, \ldots) = o_2(S, \ldots)$ should be written as $o_1(S, \ldots) = value$ and $o_2(S, \ldots) = value$ separately.

The first kind of predicates are those *without observation operator*. Such predicates can be generally formalized as $bool(V_1, \ldots, V_m)$, where $V_1, \ldots, V_m$ are variables occurring in this predicate. The function T_i for this kind of predicates is shown as follows:

$T_i(bool(V_1, \ldots, V_m)) =$

$S \mid= prop(V_1, \ldots, V_m) \ = \ true \ if \ bool(V_1, \ldots, V_m)$.

the generated formula says that: the state predicate $prop(V_1, \ldots, V_m)$ holds at arbitrary state S as long as the condition $bool(V_1, \ldots, V_m)$ is satisfied.

The predicates *with observation operator* can be further classified into two kinds: (1) predicates that are in the form of normal observation equations, which can be generally formalized as $o(S, V_1, \ldots, V_m) \ = \ term$; and (2) other non-normal ones, which can be generally formalized as $pred(\ldots, o(S, V_1, \ldots, V_m), \ldots)$. The function T_i for "normal form" predicates is shown as follows:

$T_i(o(S, V_1, \ldots, V_m) \ = \ term) =$

$(o[V_1, \ldots, V_m] \ : \ term) \ S \mid=$
$\quad prop(V_1, \ldots, V_m, X_1, \ldots, X_n) \ = \ true$.

where $X_1, \ldots, X_n$ are variables possibly contained by $term$. The generated formula says that: the state predicate $prop(V_1, \ldots, V_m, X_1, \ldots, X_n)$ holds at arbitrary state as long as this state contains $(o[V_1, \ldots, V_m]\ :\ term)$ as a fragment of it. The function T_i for "non-normal form" predicates is shown as follows:

$T_i(pred(\ldots, o(S, V_1, \ldots, V_m), \ldots)) =$

$(o[V_1, \ldots, V_m]\ :\ VAR)\ S \mid =$
$\quad prop(V_1, \ldots, V_m, X_1, \ldots, X_n)\ =\ true$
$\quad if\ pred(\ldots, VAR, \ldots)$.

where $X_1, \ldots, X_n$ are variables possibly contained by the omitted part of this predicate, and VAR is a newly generated variable denoting the return value of the observation formula $o(S, V_1, \ldots, V_m)$. As in the condition, we just rewrite the original predicate but replace the observation formula with the variable VAR. The generated formula says that: if the condition $pred(\ldots, VAR, \ldots)$ is satisfied, the state predicate $prop(V_1, \ldots, V_m, X_1, \ldots, X_n)$ holds at arbitrary state as long as this state contains $(o[V_1, \ldots, V_m]\ :\ VAR)$ as a fragment of it.

After constructing the module `SYSTEM-PREDS` that defines state predicates, we can now construct the module `SYSTEM-CHECK` that defines LTL formulas and initial state. A CafeOBJ invariant for property can be generally formalized as a tuple $inv = (PRED, \circ)$, where $PRED$ is a set of predicates, and $\circ$ is a set of logical connectives. Assume that the set of state predicates $PROP$, each element of which corresponds to a predicate in the set $PRED$, has been declared in the `SYSTEM-PREDS` module. The function T_i for the invariant translation is shown as follows:

$T_i(inv)$ = `[]`$(PROP, \bullet)$.

where the mapping between logical connectives $\circ$ and $\bullet$ is, for example, `and` to `/\`, `or` to `\/` and `implies` to `->`. `[]` is the Maude notation for LTL operator "Always".

To make the generated LTL formula model checkable, we need to instantiate the variables occurring in the LTL formula. We use a simple data abstraction method by means of reducing the infinite domain of each sort to some concrete values, where the sort is the sort (or constructive sort) of the variable occurs in the formula. For example, assume that a variables X is contained by a LTL formula, whose sort is D and the sort D is not constructed by other sorts. We make the infinite domain of D finite by selecting some concrete values, such as $d1$ and $d2$ from D. And then we instantiate the variable using these values. Although such simple data abstraction might not preserve soundness, it is effective when we aims to finding bugs [10] as in our integration.

The last work left for us is to define initial state in the Maude module `SYSTEM-CHECK`. Recall that an initial state of system *S* is represented as a bag of observers and transitions. Given a CafeOBJ OTS module that has x observers and y transitions, the equations in the CafeOBJ OTS module defining initial state, say *init*, are as follows:

`eq` o_t (*init*, $X_{i_1^t}, \ldots, X_{i_{m_t}^t}$) $= X_t$.

where $t = 1, \ldots, x$. Also assume that the transitions in the CafeOBJ OTS module are in the form of $\tau_c(S, X_{j_1^c}, \ldots, X_{j_{n_c}^c})$, where $c = 1, \ldots, y$. The initial state generated in the Maude module `SYSTEM-CHECK` by function T_i is shown as follows:

$\tau_1(X_{j_1^1}, \ldots, X_{j_{n_1}^1}) \ldots \tau_y(X_{j_1^y}, \ldots, X_{j_{n_y}^y})$
$(o_1\ [X_{i_1^1}, \ldots, X_{i_{m_1}^1}]\ :\ X_1) \ldots\ (o_x\ [X_{i_1^x}, \ldots, X_{i_{m_x}^x}]\ :\ X_x)$

where the first y terms denote all possible transitions that may change the state of *S*, and the last x terms denote the initial values of *S* returned by all observers.

To instantiate the variables occurring in the formula for initial state, we employ the same data abstraction method as used in generating LTL formula denoting property. Thus, we can get the instantiated version of the initial state.

As a summary, the translator Cafe2Maude is implemented in Java using Java DOM API for XML type[3]. Current version of the translator consists of about 3000 lines.

4. Case Study: a Mutual Exclusion Algorithm

In this section, we describe a case study on a mutual exclusion algorithm using a queue. The pseudo-code executed by each process i repeatedly can be described as follows:

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

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

4.1. OTS/CafeOBJ Specification of the Algorithm

The algorithm is modeled as an OTS with two observers and three transitions. Observer $queue$ returns the queue of process IDs shared by all processes. It initially returns the empty queue; observer $pc_i(i \in Pid)$ returns the label of a command that process i will execute next, where Pid is the sort of process IDs. Each pc_i initially returns label *l1*. Transition $wait_i(i \in Pid)$ denotes that process i executes the command at label *l1*; transition $try_i(i \in Pid)$ denotes

[3]Actually the OTS/CafeOBJ specification is firstly represented as a XML version of it for the purpose to parse the specification.

that process i executes one iteration of the loop at label *l2*; transition $exit_i(i \in Pid)$ denotes that process *i* executes the command at label *cs*. Besides the observers and transitions of the OTS, some data types used in the OTS, such as labels of commands, process IDs and queues, are also defined.

The OTS/CafeOBJ specification of the mutual exclusion algorithm consists of three data type modules (with the names `LABEL, PID` and `QUEUE`), one OTS module (with the name `QLOCK`) and one invariant property defining module (with the name `INV`). The three data type modules define sorts `Label`, `Pid` and `Queue`, respectively. We show the data type module `LABEL` as an example, and the other two data type modules are defined similarly.

The data type module defining sort `Label` is written in CafeOBJ as follows:

```
mod! LABEL {
[Label]
ops l1 l2 cs : -> Label
op _=_ : Label Label -> Bool comm
var L : Label
eq (L = L) = true .
eq (l1 = l2) = false .
eq (l1 = cs) = false .
eq (l2 = cs) = false . }
```

The OTS module specifies behaviors (state transitions) of the algorithm. The hidden sort denoting the states of the OTS is declared as `Sys`. The operators denoting the observers and transitions are declared as follows (where ‘`--`’ marks the rest of the line as a comment):

```
-- observers
bop pc    : Sys Pid -> Label
bop queue : Sys -> Queue
-- transitions
bop want : Sys Pid -> Sys
bop try  : Sys Pid -> Sys
bop exit : Sys Pid -> Sys
```

In the following, let `I`, `J` be CafeOBJ variables for `Pid`, and `S` be a CafeOBJ variable for the hidden sort `Sys` of the OTS. Operator `want` is defined with these equations:

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

The other two operators `try` and `exit` are defined with CafeOBJ equations in a similar way, which are omitted here due to space limitation.

The specification for a desired property – mutual exclusion property, is shown as follows as an example, which is defined in the CafeOBJ invariant defining module `INV`:

```
eq inv(S,I,J) = (pc(S,I)=cs and pc(S,J)=cs
                implies I=J) .
```

The formula says that if two processes `I` and `J` are both in the critical section `cs`, then the two processes are same.

4.2. Verification of the Algorithm

Let us firstly see the verification flow of our integration using Cafe2Maude. Before proving the desired property by writing proof scores in CafeOBJ, we would like to firstly check whether there exists a possible proof for this property using model checking technique. This can be done by translating the OTS/CafeOBJ specification to the OTS/Maude specification, which can then be model checked. If a counter-example arises, we should analyze the counter-example, and pinpoint the errors and revise the system (or property) specification; otherwise, if the verification result is *true*, we make a weak justification that there might exist a possible proof for this property. We can then start writing proof scores to give a full-scale proof on infinite state space. Also, during writing proof scores of the desired property, we may need to discover auxiliary invariants to support the main proof. During this course, the “translation/model checking” process is employed iteratively.

Back to the case study. Firstly the data type modules are translated by the function T_d with only changes of the manner of expression. The data type module `LABEL` as an example is translated as follows, where the CafeOBJ declaration of equivalence relation for data type and the equations defining such equivalence relation are ignored because they are not necessary in Maude functional module:

```
fmod LABEL is
sort Label .
ops l1 l2 cs : -> Label .
endfm
```

The hidden sort `Sys` is translated by function T_o exactly as described in subsection 3.2.

The operator declarations of observers and transitions are translated as follows (where ‘`***`’ marks the rest of the line as a comment) :

```
*** Observers
op pc[_] : _ : Pid Label -> OValue .
op queue : _ : Queue -> OValue .
*** transitions
op want : Pid -> TRule .
op try  : Pid -> TRule .
op exit : Pid -> TRule .
```

Transition `want` as an example is translated as follows, where `LABEL` and `QUEUE` are variables of sorts `Label` and `Queue`, respectively, which are generated by the translator denoting the values returned by observation operators before the transition `want` happens.

```
crl[want] :
   want(J)(pc[J] : LABEL)(queue : QUEUE)
   =>
   want(J)(pc[J] : l2)(queue : put(QUEUE,J))
   if LABEL == l1 .
```

The translation of the desired property involves the construction of two modules – QLOCK-PREDS and QLOCK-CHECK. The module QLOCK-PREDS is constructed as follows, where state predicates prop1(I), prop2(J) and prop3(I,J) correspond to the predicates pc(S,I)=CS, pc(S,J)=CS and I=J in the CafeOBJ invariant, respectively.

```
mod QLOCK-PREDS is
  ...
  op prop1 prop2 prop3 : Pid -> Prop .
  vars I J : Pid .
  var S : Sys .
  eq (pc[I] : cs) S |= prop1(I) = true .
  eq (pc[J] : cs) S |= prop2(J) = true .
  eq S |= prop3(I,J) = true if (I = J) .
endm
```

To make the infinite domain of the variables finite, we assign the sort Pid with two concrete values p1 and p2, for simplicity. And then we use these two values to instantiate the variables I and J. The module QLOCK-CHECK is constructed as follows:

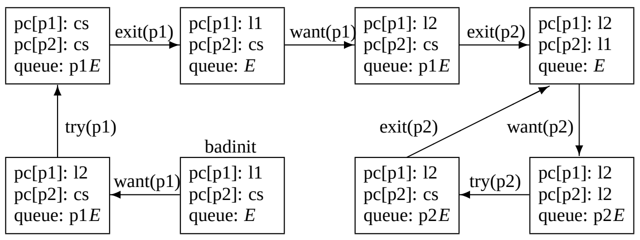
```
mod QLOCK-CHECK is
  ...
  ops p1 p2 : -> Pid .
  op init : Sys .
  op inv : -> formula .
  eq init = want(p1)try(p1)exit(p1)
            want(p2)try(p2)exit(p2)
            (pc[p1] : l1)(pc[p2] : l1)
            (queue : empty) .
  eq inv  = []((prop1(p1) /\ prop2(p2))
               -> prop3(p1,p2)) .
endm
```

By model checking the property in Maude environment, we get the verification result *true*, which convinces us to start writing proof scores of the property in CafeOBJ.

Let us consider now another example where there exist errors in the definition of initial state. We name this initial state as badinit, which is got by partly changing init from (pc[p2] : l1) to (pc[p2] : cs). We get a counter-example by checking the same mutual exclusion property in Maude environment, which can be represented as shown in Figure 1 (where *E* denotes the *empty* element of a queue).

5. Conclusion and Future Work

We described a lightweight integration of theorem proving and model checking by a translation of the formalisms



pc[p1]: cs pc[p2]: cs queue: p1*E*
exit(p1)
pc[p1]: l1 pc[p2]: cs queue: *E*
want(p1)
pc[p1]: l2 pc[p2]: cs queue: p1*E*
exit(p2)
pc[p1]: l2 pc[p2]: l1 queue: *E*
try(p1)
exit(p2)
want(p2)
badinit
pc[p1]: l2 pc[p2]: cs queue: p1*E*
want(p1)
pc[p1]: l1 pc[p2]: cs queue: *E*
pc[p1]: l2 pc[p2]: cs queue: p2*E*
try(p2)
pc[p1]: l2 pc[p2]: l2 queue: p2*E*

Figure 1. Counter-example

from the former to the latter, which is done by the translator Cafe2Maude. In the future, we plan to introduce other abstraction methods, such as *predicate abstraction* into our translator, which can provide users a *strong* justification when model checking the abstracted model returns *true*. Besides, although we have successfully employed our integration on several case studies, such as formal verification of secure workflow system, a formal correctness proof of the translation is needed and will be more convincing.

References

[1] H. Amjad. Combining model checking and theorem proving. Technical Report, Number 601, University of Cambridge, 2004.

[2] D. Basin, H. Kuruma, K. Takaragi, and B. Wolff. Verification of a signature architecture with HOL-Z. In *FM05*, LNCS, pages 269–285, 2005.

[3] S. Berezin. Model checking and theorem proving: a unified framework. PhD thesis, Carnegie Mellon University, 2002.

[4] M. Clavel, F. Durán, S. Eker, P. Lincoln, N. Martí-Oliet, J. Meseguer, and C. Talcott. *Maude 2.0 manual: Version 2.1*. http://maude.cs.uiuc.edu/maude2-manual/, March 2004.

[5] R. Diaconescu and K. Futatsugi. *CafeOBJ report*. Number 6 in AMAST Series in Computing. World Scientific, 1998.

[6] S. Eker, J. Meseguer, and A. Sridharanarayanan. The Maude LTL model checker. In *WRLA 2002*, volume 71 of *ENTCS*. Elsevier Science Publishers, 2002.

[7] W. Kong, K. Ogata, and K. Futatsugi. Model-checking observational transition system with maude. In *ITC-CSCC 2005*, pages 5–6, 2005.

[8] K. Ogata and K. Futatsugi. Rewriting-based verification of authentication protocols. In *WRLA 2002*, volume 71 of *ENTCS*. Elsevier Science Publisher, 2002.

[9] K. Ogata and K. Futatsugi. Proof scores in the OTS/CafeOBJ method. In *FMOODS '03*, volume 2884 of *LNCS*, pages 170–184. Springer, 2003.

[10] J. Rushby. Integrated formal verification: Using model checking with automated abstraction, invariant generation and theorem proving. In *SPIN 1999*, volume 1680 of *LNCS*, pages 1–11. Springer, 1999.

[11] N. Shankar. Combining theorem proving and model checking through symbolic analysis. In *CONCUR 2000*, volume 1877 of *LNCS*, pages 1–16. Springer, 2000.

[12] T. E. Uribe. Combinations of model checking and theorem proving. In *FroCoS 2000*, volume 1794 of *LNCS*, pages 151–170. Springer, 2000.