IEICE TRANS. INF. & SYST., VOL.E94–D, NO.5 MAY 2011

PAPER Special Section on Formal Approach

Translation of State Machines from Equational Theories into Rewrite Theories with Tool Support*

Min ZHANG[†a)], ***Nonmember***, **Kazuhiro OGATA**[†], ***and*** **Masaki NAKAMURA**[††], ***Members***

SUMMARY This paper presents a strategy together with tool support for the translation of state machines from equational theories into rewrite theories, aiming at automatically generating rewrite theory specifications. Duplicate effort can be saved on specifying state machines both in equational theories and rewrite theories, when we incorporate the theorem proving facilities of CafeOBJ with the model checking facilities of Maude. Experimental results show that efficiencies of the generated specifications by the proposed strategy are significantly improved, compared with those that are generated by three other existing translation strategies.

key words: *specification translation, CafeOBJ, Maude, equational theory specification, rewrite theory specification*

1. Introduction

CafeOBJ [2] and Maude [3] are two up-to-date verification systems based on algebraic approaches. The former is equipped with theorem proving facilities, while the latter with model checking facilities. CafeOBJ can be used as an interactive proof assistant for verifying invariant properties of systems. Systems are first specified as equational theories, and then invariant properties to be verified are represented as inductive theorems, which need to be verified by some proof techniques like structural induction and case splitting [4], [5]. Maude can be used as a model checker to verify desirable properties of finite-state systems. Like traditional theorem proving and model checking techniques, both CafeOBJ and Maude have their own advantages and disadvantages in system verifications. For instance, CafeOBJ can deal with infinite-state systems, but needs humans' intervention during verifications, while Maude as a model checker is fully automatic, but is only able to deal with finite-state systems. Due to the complementarity of the two verification systems, it is desirable to incorporate them with each other to fully utilize their advantages and avoid disadvantages.

A way called induction-guided falsification (IGF) is proposed to collaborate CafeOBJ with Maude in [6]. The basic idea of IGF is to use Maude to search a bounded reachable state space of a transition system for counterexamples of an invariant property. If no counterexamples are found, it uses CafeOBJ to find all necessary lemmas that are used for the verification of the property. Maude is again used to find counterexamples of these lemmas. The iteration is repeated until counterexamples are found for some of these lemmas, or the property together with all necessary lemmas is proved. Because CafeOBJ and Maude take their own specifications in different formalisms (specifications used by CafeOBJ for theorem proving are based on equational logic, while those by Maude for model checking are based on rewriting logic), we have to prepare two specifications for a same system, in order to collaborate CafeOBJ with Maude by IGF. However, it is not only effort-consuming to manually specify a system separately in CafeOBJ and Maude, but at high risk of causing inconsistencies in specifications. Automatic translation is an effective way to solve these problems. However, although some strategies have been proposed for the translation from CafeOBJ to Maude [7], [8], they are sometimes hardly used for model checking, because the generated specifications cannot be efficiently model checked in Maude.

To improve efficiencies of the generated rewriting theory specifications, we propose a new style of rewriting theory specifications in Maude denoted by RWT Specs, and investigate a strategy for the translation from equational theory specifications (abbreviated by EQT Specs) in CafeOBJ into RWT Specs. We also implement a translator called YAST to automate the translation. Experimental results show that efficiencies of the generated specifications by the proposed strategy are indeed significantly improved, compared with those which are generated by three other existing strategies. The strategy can only translate a subclass of EQT Specs called EADS Specs (EADS is abbreviated for extended asynchronous distributed systems), based on the fact that not all EQT Specs have corresponding RWT Specs (the proof is given in Sect. 3). Nevertheless, the proposed strategy is still useful, because most of systems verified in CafeOBJ can be specified as EADS Specs. This article systematically describes above work by extending [1] with the provision of the translator and analysis on more experimental results.

Contributions of this work are manifold. From a foundational point of view, we show the inequality in the expressiveness of EQT Specs and RWT Specs, and the undecidability of checking whether an EQT Spec can be translated into an RWT Spec or not. Based on this foundational result, we investigate a specific class of EQT Specs called EADS Specs as a workaround without loss of practicability. By the

Manuscript received July 20, 2010.
Manuscript revised November 14, 2010.

[†]The authors are with Japan Advanced Institute of Science and Technology (JAIST), Nomi-shi, 923–1292 Japan.
[††]The author is with Kanazawa University, Kanazawa-shi, 920–1192 Japan.

*A preliminary version of this article appeared in [1].

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

DOI: 10.1587/transinf.E94.D.976

automatic translation, we save duplicate effort on manually specifying a state machine in both equational theories and rewrite theories, when we use the IGF approach to incorporate CafeOBJ and Maude for system verifications.

The rest of this paper is organized as follow. Section 2 introduces some preliminaries i.e., state machines, EQT Specs and RWT Specs. Section 3 investigates EADS Specs, and Sect. 4 describes the translation strategy. The translator YAST is introduced in Sect. 5, followed by a case study in Sect. 6. Experimental results is shown in Sect. 7. Section 8 mentions some related work and Sect. 9 concludes the paper.

2. Preliminaries

2.1 State Machines

State machines can be used to describe dynamic systems. A state machine consists of (1) a set $\mathcal{U}$ of states, (2) a set $\mathcal{I}$ of initial states such that $\mathcal{I} \subseteq \mathcal{U}$, and (3) a set $\mathcal{T}$ of transitions. Each $u \in \mathcal{U}$ is a record consisting of a (possibly infinite) number of data fields. A record is in the form of $\{l_1 = d_1, l_2 = d_2, \ldots\}$ of type $\{l_1 : D_1, l_2 : D_2, \ldots\}$, where D with a subscript e.g. D_i denotes a data type. For convenience, we let $l_i(u)$ denote l_i's corresponding value d_i in state u. Each transition $t \in \mathcal{T}$ is a binary relation over states, and $(u_1, u_2) \in t$ denotes a transition t from state u_1 to u_2.

Let us consider a mutual exclusion protocol as an example to show how to model dynamic systems as state machines. The pseudo code of the protocol is:

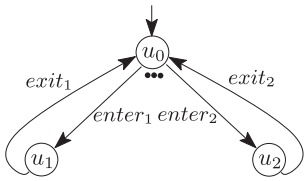
Loop remainder section
rs: **repeat while** *fetch&store(locked, true);*
critical section
cs: *locked := false;*

where, *fetch&store* is an atomic operation. It takes a variable x and a value d, and does the following two things atomically: setting x to d and returning the previous value of x. Whenever a process leaves the critical section, it sets *locked* false.

A state machine $\mathcal{M}_{\text{ME}}$ can be constructed as follow to model the protocol.

- $\mathcal{U}_{\text{ME}} \triangleq \{u | u : \{locked : Bool,\ pc_1 : Label, pc_2 : Label,\ \ldots\}\}$;
- $\mathcal{I}_{\text{ME}} \triangleq \{u_0 \in \mathcal{U}_{\text{ME}} | locked(u_0) = false, pc_i(u_0) = rs \text{ for each process } p_i\}$;
- $\mathcal{T}_{\text{ME}} \triangleq \{enter_1, enter_2, \ldots\} \cup \{exit_1, exit_2, \ldots\}$.
  For each process p_i, $(u, u') \in enter_i$ iff $pc_i(u) = rs$, $pc_i(u') = cs, locked(u) = false$, $locked(u') = true$ and $pc_j(u') = pc_j(u)$ for each $p_j \neq p_i$; and $(u, u') \in exit_i$ iff $pc_i(u) = cs$, $pc_i(u') = rs$, $locked(u') = false$ and $pc_j(u') = pc_j(u)$ for each $p_j \neq p_i$.

A part of state transitions in $\mathcal{M}_{\text{ME}}$ is shown in Fig. 1. An arrow labelled by a transition t from a state u to u' denotes $(u, u') \in t$. The triple black points denote the transitions from u_0 to other states, which are not depicted in the figure.



$u_0 = \{locked = false, pc_1 = rs, pc_2 = rs, \ldots\}$
$u_1 = \{locked = true, pc_1 = cs, pc_2 = rs, \ldots\}$
$u_2 = \{locked = true, pc_1 = rs, pc_2 = cs, \ldots\}$

Fig. 1 A part of state transitions in $\mathcal{M}_{\text{ME}}$.

2.2 EQT Specs

EQT Specs denote a class of equational theory specifications that are developed in OTS/CafeOBJ method in CafeOBJ. Transitions are defined with equations. Let Υ be a sort for states. An EQT Spec consists of (1) a finite set O of observers, (2) an arbitrary initial state *init*, (3) a finite set $\mathcal{A}$ of actions, and (4) a family $\mathcal{E}$ of sets of equations. Each observer o is an operator whose rank is $\Upsilon\ D_{o1} \ldots D_{om} \rightarrow D_o$. An observer corresponds to a (possibly infinite) set of data fields in a state u. Each action a is an operator with rank $\Upsilon\ D_{a1} \ldots D_{an} \rightarrow \Upsilon$. An action a represents a (possibly infinite) class of transitions in $\mathcal{T}$. Each action a is given an operator $c\text{-}a$ with rank $\Upsilon\ D_{a1} \ldots D_{an} \rightarrow$ Bool, representing the condition under which the transitions represented by a take place. $\mathcal{E}_{init} \in \mathcal{E}$ is a set of equations which *init* must satisfy, and for each action a, $\mathcal{E}_a \in \mathcal{E}$ is a set of equations which can be interpreted as the definition of a class of transitions represented by a.

For instance, an EQT Spec of $\mathcal{M}_{\text{ME}}$ is as follows:

$\mathcal{S}_{\text{ME}}$
$\mathcal{O}_{\text{ME}} \triangleq \{\texttt{pc} : \Upsilon\ \texttt{Pid} \rightarrow \texttt{Label}, \texttt{locked} : \Upsilon \rightarrow \texttt{Bool}\}$;
$\mathcal{A}_{\text{ME}} \triangleq \{\texttt{enter} : \Upsilon\ \texttt{Pid} \rightarrow \Upsilon, \texttt{exit} : \Upsilon\ \texttt{Pid} \rightarrow \Upsilon\}$
$\mathcal{E}_{\text{ME}} \triangleq \{\mathcal{E}_{\texttt{init}}, \mathcal{E}_{\texttt{enter}}, \mathcal{E}_{\texttt{exit}}\}$, where:
$\mathcal{E}_{\texttt{init}} \triangleq \{\texttt{pc}(\texttt{init}, x) = \texttt{rs}; \texttt{locked}(\texttt{init}) = \texttt{false}\}$
$\mathcal{E}_{\texttt{enter}} \triangleq \{\texttt{c-enter}(\upsilon, y) = \texttt{pc}(\upsilon, y) \doteq \texttt{rs} \wedge locked(\upsilon) \doteq false;$
$\texttt{pc}(\texttt{enter}(\upsilon, y), x) =$
$(\textbf{if } x \doteq y \textbf{ then } \texttt{cs} \textbf{ else } \texttt{pc}(\upsilon, x) \textbf{ fi}) \textbf{ if } \texttt{c-enter}(\upsilon, y);$
$\texttt{locked}(\texttt{enter}(\upsilon, y)) \doteq true \textbf{ if } \texttt{c-enter}(\upsilon, y);$
$\texttt{enter}(\upsilon, y) = \upsilon \textbf{ if } \neg\texttt{c-enter}(\upsilon, y)\}$
$\mathcal{E}_{\texttt{exit}} \triangleq \{\texttt{c-exit}(\upsilon, y) = \texttt{pc}(\upsilon, y) \doteq \texttt{cs};$
$\texttt{pc}(\texttt{exit}(\upsilon, y), x) =$
$(\textbf{if } x \doteq y \textbf{ then } \texttt{rs} \textbf{ else } \texttt{pc}(\upsilon, x) \textbf{ fi}) \textbf{ if } \texttt{c-exit}(\upsilon, y);$
$\texttt{locked}(\texttt{exit}(\upsilon, y)) \doteq false \textbf{ if } \texttt{c-exit}(\upsilon, y);$
$\texttt{exit}(\upsilon, y) = \upsilon \textbf{ if } \neg\texttt{c-exit}(\upsilon, y)\}$

`Pid`, and `Label` are sorts for process identifiers, queues and labels[†]. Constants `rs`, `ws` and `cs` are of sort `Label`, corresponding to labels rs, ws and cs, respectively. Variable υ is of Υ, and x, y are of `Pid`. Every variable in an equation (or a rewriting rule) is universally quantified and its scope is in the equation (or the rewriting rule). Symbol $\doteq$ denotes equiva-

[†]By convention, symbols like sorts, constants and function symbols at specification level are differentiated from those at mathematical level by using typewriter font.

lence relations over data types. The observer `locked` corresponds to the data field *locked* : *Bool* in states in $\mathcal{M}_{\mathrm{ME}}$, and `pc` to an infinite set of fields $\{pc_1 : Label, pc_2 : Label, \ldots\}$. Constant `init` together with $\mathcal{E}_{\mathtt{init}}$ specifies $\mathcal{I}_{\mathrm{ME}}$. Action `enter` corresponds to an infinite class $\{enter_1, enter_2, \ldots\}$ of transitions. The set $\mathcal{E}_{\mathtt{enter}}$ of equations can be interpreted as the definition of the class of transitions. Action `exit` together with $\mathcal{E}_{\mathtt{exit}}$ specifies the class of transitions $\{exit_1, exit_2, \ldots, \}$. Term $\mathtt{enter}(\upsilon, y)$ represents a successor of the state denoted by υ if $\mathtt{c\text{-}enter}(\upsilon, y)$ holds. Otherwise, $\mathtt{enter}(\upsilon, y)$ is equivalent to υ.

States in $\mathcal{M}_{\mathrm{ME}}$ are represented in $\mathcal{S}_{\mathrm{ME}}$ by terms of Υ, e.g. states u_0, u_1, and u_2 shown in Fig. 1 are represented by terms `init`, $\mathtt{enter}(\mathtt{init}, p_1)$, and $\mathtt{enter}(\mathtt{init}, p_2)$, respectively. For instance, $pc_1(u_1)$ is cs. Correspondingly, $\mathtt{pc}(\mathtt{enter}(\mathtt{init}, p_1), p_1)$ equals `cs`. The reduction of $\mathtt{pc}(\mathtt{enter}(\mathtt{init}, p_1), p_1)$ in $\mathcal{S}_{\mathrm{ME}}$ is as follow. According to the second equation in $\mathcal{E}_{\mathrm{enter}}$ with υ being `init`, x and y being p_1, we have

$$\mathtt{pc}(\mathtt{enter}(\mathtt{init}, p_1), p_1) = (\textbf{if } p_1 \doteq p_1 \textbf{ then } \mathtt{cs} \textbf{ else } \mathtt{pc}(\mathtt{init}, p_1) \textbf{ fi}) \textbf{ if } \mathtt{c\text{-}enter}(\mathtt{init}, p_1)$$

According to the first equation in $\mathcal{E}_{\mathtt{init}}$, $\mathtt{pc}(\mathtt{init}, p_1)$ equals `rs` and $\mathtt{locked}(\mathtt{init})$ equals false, indicating $\mathtt{c\text{-}enter}(\mathtt{init}, p_1)$ holds. Because $p_1 \doteq p_1$ is true, the right-hand side (RHS) of the equation above equals `cs`, namely that $\mathtt{pc}(\mathtt{enter}(\mathtt{init}, p_1), p_1)$ equals `cs`. Similarly, we have $\mathtt{pc}(\mathtt{enter}(\mathtt{init}, p_1), p_i)$ equals `rs` for $p_i (i > 1)$ and $\mathtt{locked}(\mathtt{enter}(\mathtt{init}, p_1))$ equals true.

2.3 RWT Specs

RWT Specs denotes a new style of rewrite theory specifications in Maude. Each state in RWT Specs is denoted by sort `State`, which is represented as a set of components. There are two kinds of components, i.e. *action components* and *observable components*. They are denoted by sorts `AComp` and `OComp` respectively, which are subsort of `State`. An action component corresponds to a class of transitions in state machines, and an observable component to a data field in a state.

An RWT Spec consists of (1) a finite set $\mathcal{OC}$ of observable component constructors, (2) a finite set $\mathcal{AC}$ of action component constructors, (3) a set $\mathcal{F}$ of function symbols for the representation of initial states, with a set $\mathcal{E}_{\mathcal{F}}$ of equations for the function symbols in $\mathcal{F}$, and (4) a finite set $\mathcal{R}$ of rewriting rules. An observable component constructor is declared in the form of $oc[\_, \ldots, \_]:\_$ which is a function symbol with rank $D_{o1} \ldots D_{om}\ D_o \rightarrow \mathtt{OComp}$[†]. An observable component constructor represents a (possibly infinite) set of data fields in a state. An action component constructor *ac* is a function symbol whose rank is $SetD_{t1}, \ldots, SetD_{tn} \rightarrow \mathtt{AComp}$, where $SetD_{ti}$ is a sort for sets of elements of D_{ti}. Each rewriting rule in $\mathcal{R}$ specifies a (possibly infinite) class of transitions.

An RWT Spec $\mathfrak{S}_{\mathrm{ME}}$ that specifies the state machine $\mathcal{M}_{\mathrm{ME}}$ is as follows:

$\mathfrak{S}_{\mathrm{ME}}$

$\mathcal{OC}_{\mathrm{ME}} \triangleq \{\mathtt{pc[\_]:\_}\colon \mathtt{Pid\ Label} \rightarrow \mathtt{OComp},$
$\mathtt{locked:\_}\colon \mathtt{Bool} \rightarrow \mathtt{OComp}\};$
$\mathcal{AC}_{\mathrm{ME}} \triangleq \{\mathtt{enter}\colon \mathtt{SetPid} \rightarrow \mathtt{AComp}, \mathtt{exit}\colon \mathtt{SetPid} \rightarrow \mathtt{AComp}\}$
$\mathcal{F}_{\mathrm{ME}} \triangleq \{\mathtt{init}\colon \mathtt{SetPid} \rightarrow \mathtt{State}, \mathtt{mk\text{-}pc}\colon \mathtt{SetPid} \rightarrow \mathtt{State}\};$
$\mathcal{E}_{\mathcal{F}_{\mathrm{ME}}} \triangleq \{\mathtt{init}(ys) = \mathtt{enter}(ys)\ \mathtt{exit}(ys)\ (\mathtt{locked}\colon \mathtt{false})$
$\mathtt{mk\text{-}pc}(ys),$
$\mathtt{mk\text{-}pc}(\mathtt{empty\text{-}set}) = \mathtt{empty\text{-}state},$
$\mathtt{mk\text{-}pc}(y\ ys) = (\mathtt{pc}[y]\colon \mathtt{rs})\ \mathtt{mk\text{-}pc}(ys)\}$
$\mathcal{R}_{\mathrm{ME}} \triangleq \{rw_{\mathtt{enter}}, rw_{\mathtt{exit}}\}$, where:
$rw_{\mathtt{enter}} \triangleq \mathtt{enter}((y\ ys))(\mathtt{pc}[y]\colon l)(\mathtt{locked}\colon b) \Rightarrow$
$\mathtt{enter}((y\ ys))(\mathtt{pc}[y]\colon \mathtt{cs})(\mathtt{locked}\colon \mathtt{true})$
$\textbf{if } l \doteq \mathtt{rs} \textbf{ and not } b,$
$rw_{\mathtt{exit}} \triangleq \mathtt{exit}((y\ ys))(\mathtt{pc}[y]\colon l)(\mathtt{locked}\colon b) \Rightarrow$
$\mathtt{exit}((y\ ys))(\mathtt{pc}[y]\colon \mathtt{rs})(\mathtt{locked}\colon \mathtt{false})\ \textbf{if } l \doteq \mathtt{cs}$

The sort `SetPid` denotes sets of process identifiers. Given a term *ps* which denotes a set of process identifiers, $\mathtt{init}(ps)$ represents the initial state when the processes participate in the mutual exclusion protocol. Rewriting rules $rw_{\mathtt{enter}}$ and $rw_{\mathtt{exit}}$ respectively specify the two classes of transitions $\{enter_1, enter_2, \ldots\}$ and $\{exit_1, exit_2, \ldots\}$ in $\mathcal{M}_{\mathrm{ME}}$.

3. EADS Specs

In RWT Specs, terms at the left-hand side (LHS) in rewriting rules represent segments of states. Rewriting rules denote these segments are correspondingly changed into those that are represented by terms at RHS in the rules, while the rest segments of states are kept unchanged. A segment at LHS in a rewriting rule consists of one action component, and a finite collection of observable components. Since one observable component only corresponds to one data field in a state in state machines, a rewriting rule can only specify the changes of a finite set of data fields in a state, namely that if there exists a rewriting rule for a transition $(u, u') \in t$, there must be a finite number of data fields in u' different from their corresponding data fields in u. Moreover, if the change of a data field depends on other data fields, the depended data fields must be finite. Therefore, if an EQT Spec can be translated into an RWT Spec, the state machine specified by the EQT Spec must satisfy the two conditions.

However, the forementioned conditions are not sufficient. Let us consider a state machine where states are of type $\{l_0 : \mathbb{N}, l_1 : \mathbb{N}, \ldots\}$. The initial state is $\{l_0 = 0, l_1 = 0, l_2 = 0, l_3 = 0, \ldots\}$. There is only one transition *inc* s.t. $(u, u') \in inc$ iff for each $i : \mathbb{N}$ if $i \leq l_0(u)$ then $l_i(u') = l_i(u)+1$, otherwise, $l_i(u') = l_i(u)$. The transition chain from the initial state is like:

$$\{l_0 = 0, l_1 = 0, l_2 = 0, l_3 = 0, \ldots\} \xrightarrow{inc} \{l_0 = 1, l_1 = 0, l_2 = 0, l_3 = 0, \ldots\} \xrightarrow{inc} \{l_0 = 2, l_1 = 1, l_2 = 0, l_3 = 0, \ldots\} \xrightarrow{inc} \{l_0 = 3, l_1 = 2, l_2 = 1, l_3 = 0, \ldots\} \xrightarrow{inc} \cdots$$

The state machine cannot be specified by any RWT Spec, because the number of changed natural numbers from u to u' varies for all $(u, u') \in inc$. After each transition, the

[†]CafeOBJ and Maude allows mixfix operators. An underscore indicates the place where an argument is put.

number of changed natural numbers is increased and unbounded. Hence, for each transition $t \in \mathcal{T}$ in a state machine which is specifiable in an RWT Spec, there must be only a bounded number of data fields changed from u to u' for each $(u, u') \in t$. Moreover, each changed data field in u' must depend upon a bounded number of data fields in u. We call the state machines that satisfy the conditions are *double bounded*, and corresponding EQT Specs *double bounded* EQT Specs. Any state machines that can be specified in RWT Specs are double bounded. Obviously, an EQT Spec that species the above state machine is not double bounded.

To automate the translation from double bounded EQT Specs into RWT Specs, we first need to check if the EQT Spec is double bounded. However, it is not decidable to check whether any given EQT Specs are double bounded. Let us consider Post's Correspondence Problem (PCP) [9] on alphabet $\{a, b\}$:

Input: a finite sequence $\{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$ of pairs of the alphabet $\{a, b\}$.
Output: the answer to the question whether there is a sequence (solution) $i_1, i_2, \ldots, i_k$ of natural numbers s.t. $a_{i_1} \cdots a_{i_k} = b_{i_1} \cdots b_{i_k}$

It is well-known that PCP is undecidable.

Let us consider an EQT Spec $\mathcal{S}_{\text{PCP}(\textit{pcp-instance})}$, where *pcp-instance* is an input sequence of PCP. Let `Seq` be a sort for sequences, `sq` be a variable of `Seq`, and `check` be a function symbol representing if a sequence is a solution to the input. $\mathcal{S}_{\text{PCP}(\textit{pcp-instance})}$ is as follow:

- $O_{\text{PCP}} \triangleq \{\texttt{isSolution} : \Upsilon\ \texttt{Seq} \to \texttt{Bool}\}$
- $\mathcal{A}_{\text{PCP}} \triangleq \{\texttt{solve} : \Upsilon \to \Upsilon\}$
- $\mathcal{E}_{\text{PCP}} \triangleq \{\mathcal{E}_{\texttt{init}}, \mathcal{E}_{\texttt{solve}}\}$
  - $\mathcal{E}_{\texttt{init}} \triangleq \{\texttt{isSolution}(\texttt{init}, \texttt{sq}) = \texttt{false}\}$
  - $\mathcal{E}_{\texttt{solve}} \triangleq \{\texttt{isSolution}(\texttt{solve}(\upsilon), \texttt{sq}) = \texttt{check}(\textit{pcp-instance}, \texttt{sq})\}$

Note that if *pcp-instance* has a solution *sq*, a repetition of *sq* is also a solution. Consequently, infinitely many observable values are changed by the transition from `init` to `solve(init)`. Thus, given an input sequence *pcp-instance* for PCP, there exists a solution to *pcp-instance* if and only if $\mathcal{S}_{\text{PCP}(\textit{pcp-instance})}$ is not double bounded. Therefore, given an any EQT Spec, it is undecidable to check if it is double bounded.

Hence, we need to impose some constraints to EQT Specs and identify a sub-class of EQT Specs which are double bounded, and decidable to check whether any given EQT Spec is in the sub-class. Such a sub-class of EQT Specs are called EADS Specs. Without loss of generality, we suppose that a special sort `Pid` is predefined for the processes (or principals). All EADS Specs must conform to the following syntax-level constraints:

1. Each $o \in O$ is declared in the form of $o : \Upsilon \to D_o$ or $o : \Upsilon\ \texttt{Pid} \to D_o$;
2. If there exists $o \in O$ s.t. $o : \Upsilon\ \texttt{Pid} \to D_o$, the declaration of each $a \in \mathcal{A}$ should be one of the following two forms:
   a. $a : \Upsilon\ \{D_{a1} \ldots D_{an}\} \to \Upsilon$, where each D_{ai} cannot be `Pid` [†];
   b. $a : \Upsilon\ \texttt{Pid}\ \{D_{a2} \ldots D_{an}\} \to \Upsilon$;

   Otherwise, there is no restriction on the declaration of each $a \in \mathcal{A}$;
3. If there exists $o \in O$ s.t. $o : \Upsilon\ \texttt{Pid} \to D_o$, equations must be in one of the following forms according the declaration of a and o:
   a. for $o : \Upsilon \to D_o$ and $a : \Upsilon\ \{D_{a1} \ldots D_{an}\} \to \Upsilon$ (D_{ai} cannot be `Pid`): $o(a(\upsilon\{, y_1, \ldots, y_n\})) = T_{oa}$ **if** $c\text{-}a(\upsilon\{, y_1, \ldots, y_n\})$;
   b. for $o : \Upsilon\ \texttt{Pid} \to D_o$ and $a : \Upsilon\ \{D_{a1} \ldots D_{an}\} \to \Upsilon$ (D_{ai} cannot be `Pid`): $o(a(\upsilon\{, y_1, \ldots, y_n\}), y) = o(\upsilon, y)$;
   c. for $o : \Upsilon \to D_o$ and $a : \Upsilon\ \texttt{Pid}\ \{D_{a2} \ldots D_{an}\} \to \Upsilon$: $o(a(\upsilon, y_1\{, y_2, \ldots, y_n\})) = T_{oa}$ **if** $c\text{-}a(\upsilon, y_1\{, y_2, \ldots, y_n\})$;
   d. for $o : \Upsilon\ \texttt{Pid} \to D_o$ and $a : \Upsilon\ \texttt{Pid}\ \{D_{a2} \ldots D_{an}\} \to \Upsilon$: $o(a(\upsilon, y_1\{, y_2, \ldots, y_n\}), y) =$ (**if** $y \doteq y_1$ **then** T_{oa} **else** $o(\upsilon, y)$ **fi**) **if** $c\text{-}a(\upsilon, y_1\{, y_2, \ldots, y_n\})$;

   where, T_{oa} is a term which represents the result into which the value observed by o is changed by action a. If all observers $o \in O$ are in the form of $o : \Upsilon \to D_o$, equations must be in form of (3a), but each D_{ai} can be any sort for data elements.
4. All observers $o' \in O$ (can be o) in T_{oa} and $c\text{-}a(\upsilon, y_1\{, y_2, \ldots, y_n\})$ must be used in the form of $o'(\upsilon\{, y_1\})$;
5. Only observers, actions and the function symbol $c\text{-}a$ associated to each action a can have Υ in their arity;
6. No actions are used in $oa(\upsilon, y_1, \{y_2, \ldots, y_n\})$ and $c\text{-}a(\upsilon, y_1\{, y_2, \ldots, y_n\})$.

We assume that a state machine $\mathcal{M}$ can be specified by an EADS Spec. Constraint 1 indicates that there are only two kinds of data fields in $\mathcal{M}$. Data fields represented by $o : \Upsilon \to D_o$ are called *system-level* data fields, which generally specify shared resources. Those represented by $o : \Upsilon\ \texttt{Pid} \to D_o$ are called *process-level* data fields, which generally represent resources that can only be accessed by a specific process. Constraint 2 indicates whenever there are process-level data fields in $\mathcal{M}$, only two kinds of transitions are allowed in $\mathcal{M}$. One is called *system-level* transition represented by $a : \Upsilon\ \{D_{a1} \ldots D_{an}\} \to \Upsilon$. A system-level transition refers to a transition that is caused by the system. The other is *process-level* transition represented by $a : \Upsilon\ \texttt{Pid}\ \{D_{a2} \ldots D_{an}\} \to \Upsilon$. A process-level transition refers to a transition that is caused by a process. Constraint 3 assures that only a bounded number of values in data fields in a state are changed by a transition. Equations (3a) and (3b) indicate that in the dynamic system only

[†]Contents in { and } may or may not occur.

system-level values can be changed by system-level transitions, and Eqs. (3c) and (3d) indicate a process-level transition can only change system-level values and the process's own values. Constraint 4 indicates a process-level transition executed by a process can only access the system-level data fields and the process-level data fields owned by the process. Constraint 5 guarantees that if a state machine can be specified by an EADS Spec, the changes of data fields in its states depend upon a bounded number of data fields, so do the conditions under which the transitions can take place in states. Constraint 6 assures that each action can be interpreted as a transition.

EADS Specs specify a class of asynchronous distributed systems such as communication protocols and distributed mutual exclusion protocols, and some class of asynchronous shared-memory systems such as the mutual exclusion protocol in this paper. These systems are characterized by the two main features: (1) A system consists of multiple processes (or principals, etc.) and some shared resources, and (2) Each process (or principal) has only bounded number of components, and each process is only allowed to access and modify its own components, besides shared resources.

4. Translation Strategy

The translation from an EADS Spec to an RWT Spec consists of two phases. The first phase is to generate observable component constructors $\mathcal{OC}$ and action component constructors $\mathcal{AC}$ from $\mathcal{O}$ and $\mathcal{A}$, and to generate rewriting rules $\mathcal{R}$ from $\mathcal{E}$. The second phase includes the optimization of translated RWT Specs and the construction of initial states for the optimized RWT Specs.

4.1 Generation of $\mathcal{OC}$ and $\mathcal{AC}$

According to the declaration of observers $\mathcal{O}$ and actions $\mathcal{A}$, we can generate observable component constructors $\mathcal{OC}$ and action component constructors $\mathcal{AC}$, as shown in Fig. 2.

4.2 Generation of $\mathcal{R}$

We need to construct a set $\mathcal{R}$ of rewriting rules to specify the actions in $\mathcal{A}$. There is a rewriting rule corresponding to each action.

If all observers in an EADS Spec are declared in the form of $o : \Upsilon \rightarrow D_o$, we consider all these observers to construct a rewriting rule for each action a s.t. $a : \Upsilon\ D_{a1}, \ldots, D_{an} \rightarrow \Upsilon$ ($n \geq 0$). According to Eq. (3a), $o(\upsilon)$ is changed into $T_{oa\downarrow\mathcal{S}}$ when $c\text{-}a(\upsilon)$ holds[†]. We introduce a fresh variable d_o of D_o denoting the value denoted by $o(\upsilon)$ in a data field. We assume there are m ($m \geq 1$) observers s.t. $\mathcal{O} = \{o_1, \ldots, o_m\}$. A rewriting rule that is constructed from a and $\mathcal{E}_a$ is as follow:

$a((y_1\ ys_1), \ldots, (y_n, ys_n))(o_1\colon o_1(\upsilon)) \ldots (o_m\colon o_m(\upsilon)) \Rightarrow$
$a((y_1\ ys_1), \ldots, (y_n, ys_n))(o_1\colon T_{o_1a\downarrow\mathcal{S}}) \ldots (o_m\colon T_{o_ma\downarrow\mathcal{S}})$
if $c\text{-}t(\upsilon)_{\downarrow\mathcal{S}}$,

with terms $o_1(\upsilon), \ldots, o_m(\upsilon)$ substituted by $d_{o_1}, \ldots, d_{o_m}$ respectively. In the rewriting rule, each component at LHS has a corresponding successor at RHS. The action component keeps unchanged. The change of observable components like from $(o : o(\upsilon))$ to $(o\colon T_{oa\downarrow\mathcal{S}})$ exactly denotes the one from $o(\upsilon)$ to $T_{oa\downarrow\mathcal{S}}$ in the original EQT Spec. Note that $T_{oa\downarrow\mathcal{S}}$ can be $o(\upsilon)$, which means that corresponding shared resource is not changed. If $o(\upsilon)$ is also not used by other observable component, it can be removed from the rewriting rule. This step is called *optimization* (see Sect. 4.3 for details).

If there exists an observer o in an EADS Spec s.t. $o : \Upsilon\ \texttt{Pid} \rightarrow D_o$, actions are declared in the form of either $a : \Upsilon\ D_{a1} \ldots D_{an} \rightarrow \Upsilon$ ($n \geq 0$) (D_{ai} cannot be `Pid`) or $a : \Upsilon\ \texttt{Pid}\ D_{a2} \ldots D_{an} \rightarrow \Upsilon$ ($n \geq 0$). In the first case, only system-level data fields can be changed by the transition represented by a. We only need to consider those observers that denote system-level data fields, i.e. the observers that are in form of $o : \Upsilon \rightarrow D_o, o(\upsilon)$. The way of constructing a rewriting rule for a is similar to the case when all observers are declared in the form of $o : \Upsilon \rightarrow D_o$.

In the second case, since an observer $a \in \mathcal{A}$ s.t. $a : \Upsilon\ \texttt{Pid}\ D_{a2} \ldots D_{an} \rightarrow \Upsilon$ ($n \geq 0$) denotes a process-level transition, both system-level and process-level data fields can be accessed. Let y_1 be a variable of `Pid` and $y_2, \ldots, y_n$ be variables of $D_{a2}, \ldots, D_{an}$ respectively. We consider a process-level transition denoted by a w.r.t. y_1 and parameters $y_2, \ldots, y_n$. For system-level data fields represented by $o \in \mathcal{O}$ s.t. $o : \Upsilon \rightarrow D_o$, we deal with it similarly like in the construction of rewriting rules for system-level transitions. For each $o \in \mathcal{O}$ s.t. $o : \Upsilon\ \texttt{Pid} \rightarrow D_o$, among process-level data fields denoted by o, only those owned by y_1 can be accessed. In the state denoted by υ, the value in a process-level data field of the process y_1 w.r.t. o is denoted by $o(\upsilon, y_1)$. According to Eq. (3d), it is changed into $T_{oa\downarrow\mathcal{S}}$ in the successor $a(\upsilon, y_1, y_2, \ldots, y_n)$ under the condition that $c\text{-}a(\upsilon, y_1, y_2, \ldots, y_n)$ holds. We introduce a fresh variable d_o of D_o corresponding to $o(\upsilon, y_1)$. We assume that the first k observers are in the form of $o : \Upsilon \rightarrow D_o$ and rest of $o : \Upsilon\ \texttt{Pid} \rightarrow D_o$ in m observers $\{o_1, \ldots, o_k, o_{k+1}, \ldots, o_m\}$. A rewriting rule specifying a set of process-level transitions denoted by action a and $\mathcal{E}_a$ w.r.t. $y_1, y_2, \ldots, y_n$ is as follow:

$a((y_1\ ys_1), (y_2\ ys_2), \ldots, (y_n\ ys_n))\ (o_1\colon o_1(\upsilon)) \ldots$
$(o_k\colon o_k(\upsilon))(o_{k+1}[y_1]\colon o_{k+1}(\upsilon, y_1)) \ldots (o_m[y_1]\colon o_m(\upsilon, y_1))$
$\Rightarrow a((y_1\ ps_1), (y_2\ ys_2), \ldots, (y_n\ ys_n))(o_1\colon T_{o_1a\downarrow\mathcal{S}}) \ldots$

$\mathcal{O}\&\mathcal{A}$		$\mathcal{OC}\&\mathcal{AC}$
$o : \Upsilon \rightarrow D_o$	⟶	$o\colon\_ : D_o \rightarrow$ `OComp`
$o : \Upsilon\ \texttt{Pid} \rightarrow D_o$	⟶	$o[\_]\colon\_ : \texttt{Pid}\ D_o \rightarrow$ `OComp`
$a : \Upsilon\ D_{a1} \ldots D_{an} \rightarrow \Upsilon$	⟶	$a : SetD_{a1} \ldots SetD_{an} \rightarrow$ `AComp`

Fig. 2 Translation from $\mathcal{O}$ and $\mathcal{A}$ into $\mathcal{OC}$ and $\mathcal{AC}$.

[†]$T_{oa\downarrow\mathcal{S}}$ represents the canonical form of T_{oa} in context $\mathcal{S}$. We assume all EADS Specs are confluent and terminating.

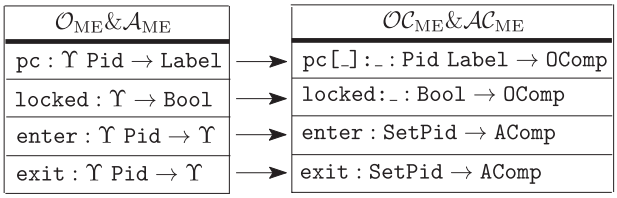


Fig. 3 Translation from $\mathcal{O}_{\mathrm{ME}}$ and $\mathcal{A}_{\mathrm{ME}}$ into $\mathcal{OC}_{\mathrm{ME}}$ and $\mathcal{AC}_{\mathrm{ME}}$.

$(o_k\colon T_{o_k a\downarrow\mathcal{S}})(o_{k+1}[y_1]\colon T_{o_{k+1}a\downarrow\mathcal{S}})\ldots(o_m[y_1]\colon T_{o_m a\downarrow\mathcal{S}})$
if $c\text{-}t(\upsilon, y_1, y_2, \ldots, y_n)_{\downarrow\mathcal{S}}$,

with $o_1(\upsilon), \ldots, o_k(\upsilon), o_{k+1}(\upsilon, y_1), \ldots, o_m(\upsilon, y_1)$ substituted by $d_{o_1}, \ldots, d_{o_k}, d_{o_{k+1}}, \ldots, d_{o_m}$, respectively.

We take the construction of the rule $rw_{\mathtt{enter}}$ in $\mathfrak{S}_{\mathrm{ME}}$ from $\mathcal{S}_{\mathrm{ME}}$ as an example. First, observers in $\mathcal{O}_{\mathrm{ME}}$ and actions in $\mathcal{A}_{\mathrm{ME}}$ are translated into $\mathcal{OC}_{\mathrm{ME}}$ and $\mathcal{AC}_{\mathrm{ME}}$, as shown in Fig. 3. Since the transitions represented by `enter` are process-level transitions. We first construct an intermediate rewriting rule to specify the transition:

$\mathtt{enter}((y\ ys))(\mathtt{pc}[y]\colon \mathtt{pc}(\upsilon, y))(\mathtt{locked}\colon \mathrm{locked}(\upsilon)) \Rightarrow$
$\mathtt{enter}((y\ ys))(\mathtt{pc}[y]\colon \mathtt{pc}(\mathtt{enter}(\upsilon, y), y)_{\downarrow\mathcal{S}_{\mathrm{ME}}})(\mathtt{locked}\colon$
$\mathtt{locked}(\mathtt{enter}(\upsilon, y))_{\downarrow\mathcal{S}_{\mathrm{ME}}})$ **if** $\mathtt{c\text{-}enter}(\upsilon, y)_{\downarrow\mathcal{S}_{\mathrm{ME}}}$.

According to $\mathcal{S}_{\mathrm{ME}}$, $\mathtt{pc}(\mathtt{enter}(\upsilon, y), y)$ is reduced to `cs` and $\mathtt{locked}(\mathtt{enter}(\upsilon, y))$ to `true`, under the condition that $\mathtt{c\text{-}enter}(\upsilon, y)$ is true, namely that $\mathtt{pc}(\upsilon, y) \doteq \mathtt{rs}$ **and** $\mathtt{locked}(\upsilon) \doteq \mathtt{false}$ is true. The above rule can be transformed to:

$\mathtt{enter}((y\ ys))(\mathtt{pc}[y]\colon \mathtt{pc}(\upsilon, y))(\mathtt{locked}\colon \mathtt{locked}(\upsilon)) \Rightarrow$
$\mathtt{enter}((y\ ys))(\mathtt{pc}[y]\colon \mathtt{cs})(\mathtt{locked}\colon \mathtt{true})$
if $\mathtt{pc}(\upsilon, y) \doteq \mathtt{rs}$ **and not** $\mathtt{locked}(\upsilon)$.

We introduce fresh variables l of `Label` and b of `Bool`, respectively for $\mathtt{pc}(\upsilon, y)$ and $\mathtt{locked}(\upsilon)$. Then, we obtain the following rule:

$\mathtt{enter}((y\ ys))(\mathtt{pc}[y]\colon l)(\mathtt{locked}\colon b) \Rightarrow \mathtt{enter}((y\ ys))$
$(\mathtt{pc}[y]\colon \mathtt{cs})(\mathtt{locked}\colon \mathtt{true})$ **if** $l \doteq \mathtt{rs}$ **and not** b,

which is exactly the one in $\mathcal{S}_{\mathrm{ME}}$.

4.3 Optimization of RWT Specs

Generated RWT Specs need to be optimized so that they can be efficiently model checked in Maude. In Maude, rewriting with both equations and rules takes place by matching an LHS against a subject term and evaluating the corresponding condition [3, chap. 1]. Hence, the less complex the LHS and the condition of a rewriting rule are, the less time it takes to match a term to the LHS and to evaluate the condition, respectively.

A general way of optimizing rewriting rules is deleting redundant terms. In an RWT Spec, action components are not changed in rewriting rules. They are used to provide necessary variables to guarantee rewriting rules are executable, because Maude generally requires variables that occur in the RHS or condition must occur in the LHS to make rewriting rules executable [3, chap. 6]. However, some variables in an action component may be also used by some observable components at the LHS in rewriting rules. In this situation, these variables in the action component become redundant. For instance, in the rule $rw_{\mathtt{enter}}$, y in the action component $\mathtt{enter}((y\ ys))$ is also used in the observable component $(\mathtt{pc}[y]\colon l)$. We can remove the action component from the rule, without affecting the executability of the rule. We obtain an optimized rule:

$(\mathtt{pc}[y]\colon l)(\mathtt{locked}\colon b) \Rightarrow (\mathtt{pc}[y]\colon \mathtt{cs})(\mathtt{locked}\colon \mathtt{true})$
if $l \doteq \mathtt{rs}$ **and not** b

Another case is that when a parameter $y_k, k \in \{1, \ldots, n\}$ in an action component of a occurs in some observable components at LHS of a rewriting rule or y_k occurs neither in any observable components at RHS nor in condition, we can remove the k^{th} parameter of a, and consequently revise the declaration of a in $\mathcal{AC}$. If all parameters of a are removed, the action component can be removed.

Redundant observable components in rewriting rules can also be deleted. An observable component is redundant when the value in it is neither changed by the transition, nor used by other components or in conditions. A redundant observable component can be deleted directly from both the sides of rewriting rules.

Another optimization is to simplify or delete the condition of a rewriting rule. The optimization is achieved by *equivalent replacement*. We assume that the condition is a conjunction. If a conjunct in the condition is an equivalence relation in the form of $x \doteq T$ and x occurs in neither T nor the other part of the condition, where x is a variable and T is a term, we can replace x that occurs in the both sides of the rewriting rule with T and delete the conjunct from the condition. Hence, the rule $rw_{\mathtt{enter}}$ can be further optimized to be:

$(\mathtt{pc}[y]\colon rs)(\mathtt{locked}\colon b) \Rightarrow (\mathtt{pc}[y]\colon \mathtt{cs})(\mathtt{locked}\colon \mathtt{true})$
if not b.

An optimized RWT Spec of mutual exclusion protocol is as follow:

$\mathfrak{S}'_{\mathrm{ME}}$

$\mathcal{OC}'_{\mathrm{ME}} \triangleq \{\mathtt{pc[\_]:\_}\colon \mathtt{Pid\ Label} \to \mathtt{OComp},$
$\mathtt{locked:\_}\colon \mathtt{Bool} \to \mathtt{OComp}\};$
$\mathcal{AC}'_{\mathrm{ME}} \triangleq \emptyset;$
$\mathcal{F}'_{\mathrm{ME}} \triangleq \{\mathtt{init}\colon \mathtt{SetPid} \to \mathtt{State}, \mathtt{mk\text{-}pc}\colon \mathtt{SetPid} \to \mathtt{State}\};$
$\mathcal{E}'_{\mathcal{F}'_{\mathrm{ME}}} \triangleq \{\mathtt{init}(ys) = (\mathtt{queue}\colon\ \mathtt{empty})\ \mathtt{mk\text{-}pc}(ys),$
$\mathtt{mk\text{-}pc}(\mathtt{empty\text{-}set}) = \mathtt{empty\text{-}state},$
$\mathtt{mk\text{-}pc}(y\ ys) = (\mathtt{pc}[y]\colon\ \mathtt{rs})\ \mathtt{mk\text{-}pc}(ys).\}$
$\mathcal{R}'_{\mathrm{ME}} \triangleq \{rw_{\mathtt{enter}}, rw_{\mathtt{exit}}\}$, where:
$rw_{\mathtt{want}} \triangleq (\mathtt{pc}[y]\colon \mathtt{rs})(\mathtt{locked}\colon b) \Rightarrow$
$(\mathtt{pc}[y]\colon \mathtt{cs})(\mathtt{locked}\colon \mathtt{true})$ **if not** b;
$rw_{\mathtt{exit}} \triangleq (\mathtt{pc}[y]\colon \mathtt{cs})(\mathtt{locked}\colon \mathtt{true}) \Rightarrow$
$(\mathtt{pc}[y]\colon \mathtt{rs})(\mathtt{locked}\colon \mathtt{false}).$

4.4 Generation of $\mathcal{F}$ and $\mathcal{E}_{\mathcal{F}}$

The last step is to construct an initial state in the generated

RWT Spec to correspond to *init* in the source EADS Spec. The initial state is specified by a set $\mathcal{F}$ of function symbols and a set $\mathcal{E}_{\mathcal{F}}$ of equations for $\mathcal{F}$.

In the initial states represented by *init*, the value of the data field corresponding to an observer o s.t. $o : \Upsilon \to D_o$ is $o(init)_{\downarrow\mathcal{S}}$. Consequently, an observable component $(o : o(init)_{\downarrow\mathcal{S}})$ in the generated RWT Spec can be constructed to correspond to the data field. For each observer $o \in \mathcal{O}$ s.t. $o : \Upsilon\ \texttt{Pid} \to D_o$, the value of the data field corresponding to o with a process y in initial states is denoted by $o(init, y)_{\downarrow\mathcal{S}}$. Hence, the data field can be denoted by the observable component $(o[y] : o(init, y)_{\downarrow\mathcal{S}})$. Given a set of processes, we need to construct an observable component for each process to represent the initial state. We declare an auxiliary function denoted by *mk-o* : `PidSet` → `State` to achieve this. The following two equations are declared for *mk-o*:

mk-o(`empty-set`) = `empty-state`,
mk-o(*y ys*) = $(o[p] : o(init, y)_{\downarrow\mathcal{S}})$ *mk-o*(*ys*),

Let sa_i $(1 \leq i \leq n)$ denote an instance i.e. a set of $SetD_{ai}$ that are used in the specified system. For each $a : SetD_{a1} \ldots SetD_{an} \to$ `AComp`, the action component of a in the initial states can be constructed as $a(sa_1, \ldots, sa_n)$.

We suppose $\mathcal{OC}$ consists of m observable component constructors, where the first k constructors are declared as o_i:_ : $D_{oi} \to$ `OComp` for $i = 1, \ldots, k$, and the rest as o_i[_]:_ : `Pid` $D_{oi} \to$ `OComp` for $i = k+1, \ldots, m$. We further suppose $\mathcal{OA}$ consists of n action component constructors $a_1, \ldots, a_n$ and each a_j takes l_j parameters for $j = 1, \ldots, n$. Let $\{SetD_1, \ldots, SetD_{n'}\}$ be the set of all sorts that are taken by at least one component constructor in $\mathcal{AC}$. We declare a function symbol `init` s.t. `init` : `SetPid` $SetD_1 \ldots SetD_{n'} \to$ `State`, and declare the following equation for `init`:

`init`$(ys, sd_1, \ldots, sd_{n'}) = (o_1 : o_1(init)_{\downarrow\mathcal{S}}) \ldots (o_k :$
$o_k(init)_{\downarrow\mathcal{S}})$ *mk-o*$_{k+1}(ys) \ldots$ *mk-o*$_m(ys)$
$a_1(sd1_1, \ldots, sd1_{l_1}) \ldots a_n(sdn_1, \ldots, sdn_{l_n})$,

where each sd_i $(1 \leq i \leq n')$ is a variable of $SetD_i$, and each sdj_w $(1 \leq j \leq n, 1 \leq w \leq l_j)$ is one of $sd_1, \ldots, sd_{n'}$. Consequently, we obtain a set of function symbols $\mathcal{F} = \{$`init`, *mk-o*$_{k+1}, \ldots,$ *mk-o*$_m\}$, and a set $\mathcal{E}_{\mathcal{F}}$ of equations that are declared for `init` and *mk-o*$_{k+1}, \ldots,$ *mk-o*$_n$.

For instance, we show how $\mathcal{F}'_{\mathrm{ME}}$ and $\mathcal{E}'_{\mathcal{F}'_{\mathrm{ME}}}$ in Ξ'_{ME} are constructed to represent initial states that correspond to `init` in $\mathcal{S}_{\mathrm{ME}}$. Because $\mathcal{AC}'_{\mathrm{ME}}$ is empty, we only need to consider $\mathcal{OC}'_{\mathrm{ME}}$. Since only sort `Pid` is taken as a parameter sort, `init` is declared as `init` : `PidSet` → `State`, and we have `init`(*ys*) = (`locked`: `locked`$(init)_{\downarrow\mathcal{S}_{\mathrm{ME}}}$) `mk-pc`(*ys*). That is `init`(*ys*) = (`locked`: `false`) `mk-pc`(*ys*). Because `pc`(*init*, *y*) is `rs` for each process *y*, `mk-pc` : `PidSet` → `State` can be declared as follows:

`mk-pc`(`empty-set`) = `empty-state`;
`mk-pc`(*y ys*) = (`pc`[*y*]: `rs`) `mk-pc`(*ys*).

4.5 Principles of Defining EADS Specs

There may be one or more EADS Specs for a given extended asynchronous distributed system which consequently corresponds to different RWT Specs generated by the proposed strategy. The efficiency of RWT Specs varies according to the complexity of rewriting rules. Some principles should be followed to develop EADS Specs from which efficient RWT Specs can be generated: (1) condition should be in conjunction form if possible, and (2) each conjunct should be in the form of $o(\upsilon\{, x_1\}) \doteq T$ if possible, where T is a term.

For instance, the effective condition of action `enter` is specified by `pc`$(\upsilon, p) \doteq$ `rs` **and not** `locked`(υ). The rewriting rule $rw_{\texttt{enter}}$ can be optimized with the first conjunct in its condition, as shown in Sect. 4.3. The rule can be further optimized if the effective condition can be specified by an equivalent expression `pc`$(\upsilon, p) \doteq$ `rs` **and** `locked`$(\upsilon) \doteq$ `false`. We have

(`pc`[*y*]: *rs*)(`locked`: *b*) ⇒ (`pc`[*y*]: `cs`)(`locked`: `true`)
if $b \doteq$ `false`.

Then, we replace *b* with `false` in the rule and obtain:

(`pc`[*y*]: `rs`)(`locked`: `false`) ⇒
(`pc`[*y*]: `cs`)(`locked`: `true`)

which is much simpler than the previous optimized one.

Let us consider another example. We assume an observer `queue` corresponds to a shared queue in a system, and an action *a* takes place under the condition that a process *p* is at the top of the queue. The condition can be specified by either `top`(`queue`$(\upsilon)) \doteq p$ or `queue`$(\upsilon) \doteq (q|p)$, where `top` is a function symbol for returning the top element of a queue, *q* is a variable of sort `queue` and the rightmost element is the top one. The related rewriting rules can be optimized with the second declaration, but not with the first one.

5. Tool Support: YAST

The proposed translation strategy in Sect. 4 is automated by a translator called YAST. YAST takes an EQT Spec and first checks if it is an EADS Spec. If an input EQT Spec is an EADS Spec, YAST automatically generates a corresponding RWT Spec, otherwise, it terminates immediately with warning messages.

Different from other existing translators from CafeOBJ specifications of state machines into other specifications e.g. Cafe2Maude [7], Chocolat/SMV [10], YAST is implemented in Maude meta-programming [11], [12]†. Due to the modularity of Maude meta-programming, YAST has a clear three-level architecture of the design, as depicted in Fig. 4. The bottom level is called *object-level*, where EADS Specs $\mathcal{S}$ and RWT Specs Ξ are interpreted as mathematical theories. The middle level is called *meta-level* where EADS Specs and RWT Specs are treated as data structures, on which we can perform some kind of analysis or computations with Maude's meta-level facilities. YAST admits object-level EADS Specs $\mathcal{S}$, and then generates corresponding

†YAST together with some examples is now available at http://www.jaist.ac.jp/~s0820005.

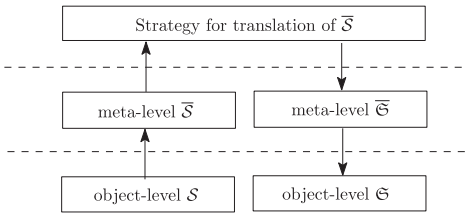


Fig. 4 Three-level architecture of YAST.

meta-level representations $\overline{\mathcal{S}}$. $\overline{\mathcal{S}}$ are then passed as data to the upper level, which is called *meta meta-level*. Specifications of translation strategies are located at the meta meta-level. They take $\overline{\mathcal{S}}$ as input data, and generate corresponding meta-level RWT Specs $\overline{\mathfrak{S}}$. Object-level RWT Specs $\mathfrak{S}$ are recovered by Maude built-in function `downTerm` from their meta-level representations $\overline{\mathfrak{S}}$. The arrows in Fig. 4 show the data flow inside YAST.

Advantages of implementing YAST in Maude metaprogramming are as follows. (1) Compared to using conventional programming languages, less effort is needed. For instance, Cafe2Maude is implemented in Java with 3,000 lines of code, while it only needs 2,000 lines of Maude code to implement YAST. Moreover, YAST is an extension of Full Maude [3, char. 18]. Full Maude provides an ample of meta-level functions, which can be reused in the implementation of YAST. (2) Maude's meta-level facilities are naturally suited for the translation and optimization. The implementation of YAST is essentially an executable algebraic specification of the proposed strategy in Maude. It takes fully advantage of Maude's reasoning facilities i.e. reducing to interpret the input EQT Specs and generate optimal RWT Specs correspondingly. (3) our tool-building experience has been positive, both in terms of how quickly we were able to develop the tool, and how easily we could extend and maintain it.

Furthermore, thanks to the efficient implementation of Maude (that can reach 1,300,000 rewrites per second on a 450 MHz Pentinum II for some applications), the performance of YAST is also competitive in that most of translations are finished in seconds. Due to the proposed strategy, translated RWT Specs are also efficient enough to be model checked in reasonable time. The efficiency of both YAST and generated specifications guarantees the usefulness of YAST in practice.

6. A Case Study: NSPK

NSPK is a security protocol to achieve mutual authentication between two principals over network [13]. An EADS Spec $\mathcal{S}_{\text{NSPK}}$ of NSPK has been developed in order to try proving the protocol enjoys Secrecy Property. However, the property fails to be proved in CafeOBJ. A counterexample is desirable to show NSPK does not satisfy Secrecy Property. In this section, we show how to use YAST to generate a corresponding RWT Spec $\mathfrak{S}_{\text{NSPK}}$ of NSPK from $\mathcal{S}_{\text{NSPK}}$.

NSPK can be described as three message exchanges:

$$\begin{aligned}
&\text{Msg1 } p \longrightarrow q : \mathcal{E}_q(n_p, p)\\
&\text{Msg2 } q \longrightarrow p : \mathcal{E}_p(n_p, n_q)\\
&\text{Msg3 } p \longrightarrow q : \mathcal{E}_q(n_q)
\end{aligned}$$

Each principal is given a pair of keys: public and private keys. $\mathcal{E}_p(m)$ denotes an encrypted message of a plain message m with the principal p's public key, and n_p is nonce (represented by a random number) generated by principal p. Besides normal principals, intruders are taken into account to verify Secrecy Property. Intruders are able to (1) wiretap any messages in the network, (2) collect nonces used in messages if ciphertexts in messages can be decrypted by the intruders (i.e. when ciphertexts are encrypted with the intruder's public key), (3) fake messages based on wiretapped messages and collected nonces, and (4) put the faked messages into the network.

To specify NSPK in an EADS Spec $\mathcal{S}_{\text{NSPK}}$, some basic data types and related operations should be first defined, e.g. `Prin` for principals, `Nonce` for nonces, and `Msg` for messages, etc. We use one action in NSPK to partially show how to specify NSPK in an EADS Spec. The action is that Msg1 is sent from one principal to another, which can be specified by the following code snippet:

```
bop send1 :  Sys Prin Prin -> Sys
op  c-send1 :  Sys Prin Prin -> Bool .
eq  c-send1(S,P1,P2) = not (P1 = P2) .
ceq rand(send1(S,P1,P2)) =
    next(rand(S)) if c-send1(S,P1,P2) .
ceq nw(send1(S,P1,P2)) =
    m(P1,P1,P2,enc1(P2,n(P1,P2,rand(S)),P1)) nw(S)
    if c-send1(S,P1,P2) .
ceq nonces(send1(S,P1,P2)) =
    (if P2 = intr then n(P1,P2,rand(S)) nonces(S)
     else nonces(S) fi)
   if c-send1(S,P1,P2) .
ceq send1(S,P1,P2) = S if not c-send1(S,P1,P2) .
```

Keyword `bop` is used to declare actions[†], and `eq` (or `ceq`) equations (or conditional equations). Sort `Sys` corresponds to system states. `S` is a variable of `Sys` and `P1` and `P2` of `Prin`. Operators `rand`, `nw` and `nonces` are observers. which correspond to values in a state, e.g. `rand(S)` denotes a random number in a state corresponding to `S`. The transition from `S` to `send1(S,P1,P2)` takes places under the condition that `c-send1(S,P1,P2)` holds, namely that `P1` and `P2` are not equal. After the transition takes place, the values are changed. A new random number replaces the older one; an encrypted message is added to network and a nonce is wiretapped if `P2` denotes an intruder. These changes are specified by the first three conditional equations, respectively. The last equation says if `c-send1(S,P1,P2)` does not hold, `S` and `send1(S,P1,P2)` are treated as equal.

YAST generates the following rewriting rule $rw_{\texttt{send1}}$ corresponding to action `send1` in $\mathcal{S}_{\text{NSPK}}$.

[†]We use `bop` to differentiate actions and observers from other operators. In the future, all operators will declared by `op`, instead of `bop`.

```
crl send1((V-Prin1:Prin ; V-Prin1set:SetPrin),
          (V-Prin2:Prin ; V-Prin2set:SetPrin))
    (nonces: V-nonces:NonceSet)(nw: V-nw:MsgSet)
    (rand: V-rand:Rand) =>
    send1((V-Prin1:Prin ; V-Prin1set:SetPrin),
          (V-Prin2:Prin ; V-Prin2set:SetPrin))
    (nonces: if intr = V-Prin2:Prin then
                V-nonces:NonceSet n(V-Prin1:Prin,
                V-Prin2:Prin,V-rand:Rand)
             else V-nonces:NonceSet fi)
    (nw:(V-nw:MsgSet m(V-Prin1:Prin,V-Prin1:Prin,
   V-Prin2:Prin,enc1(V-Prin2:Prin,n(V-Prin1:Prin,
      V-Prin2:Prin,V-rand:Rand),V-Prin1:Prin))))
    (rand:  next(V-rand:Rand))
if true xor V-Prin1:Prin = V-Prin2:Prin = true
[label send1] .
```

A variable *Var* of sort Sort is declared on-the-fly in the form of *Var*:Sort. Sort SetPrin is automatically generated, denoting sets of principals. Union operator _;_ is associative and commutative over sets of principals, so that (*V-Prin1*:Prin ; *V-Prin1set*:SetPrin) denotes a non-empty set of principals and *V-Prin1set* can be any principal in a set. The rewriting rule specifies principal *V-Prin1* sends Msg1 to principal *V-Prin2*. Term (rand: *V-rand*:Rand) denotes that the random number is *V-rand* in a state. It is changed into next(*V-Prin1*:Rand) after the action. Consequently, we obtain (rand: next(*V-Prin1*:Rand)) as shown at RHS in the rule. Similarly, the changes of nonces and messages by the action are also reflected in the rule. The condition part says *V-Prin1* does not equal *V-Prin2*. The rewriting rule faithfully specifies the same action that is specified by equations in $\mathcal{S}_{\text{NSPK}}$.

Initially, there are no nonces and messages. Let constant empty denote empty set of nonces and empty set of messages[†], and seed denote an initial random number. An constant init is declared in $\mathcal{S}_{\text{NSPK}}$ with the following three equations, for the representation of initial state in NSPK.

```
eq rand(init)   = seed .
eq nw(init)     = empty .
eq nonces(init) = empty .
```

YAST constructs an initial state generator with the following equation in the RWT Spec $\mathfrak{S}_{\text{NSPK}}$, which corresponds to init in $\mathcal{S}_{\text{NSPK}}$.

```
eq init(V-SetPrin:SetPrin)=
   fake1(V-SetPrin:SetPrin,V-SetPrin:SetPrin)
   fake2(V-SetPrin:SetPrin,V-SetPrin:SetPrin)
   fake3(V-SetPrin:SetPrin,V-SetPrin:SetPrin)
   fake4(V-SetPrin:SetPrin,V-SetPrin:SetPrin)
   send1(V-SetPrin:SetPrin,V-SetPrin:SetPrin)
   (nonces: empty)(nw: empty)(rand: seed) .
```

Note that there are five action components beside the observable components. This is because these action components cannot be removed after optimization, as shown in the rule $rw_{\texttt{send1}}$. To make sure the generated rewriting rules executable, these action components should be initially provided.

The initial state generator represents arbitrary initial states. In order to do model checking with $\mathfrak{S}_{\text{NSPK}}$, we need to provide a concrete initial state by fixing a bounded number of principals and assigning them to *V-SetPrin*. For instance, we assume there are three principals in NSPK denoted by p1, p2 and intr, where intr stands for an intruder. The initial state specified in $\mathfrak{S}_{\text{NSPK}}$ is init(p1 ; p2 ; intr), namely that:

```
fake1(p1 ; p2 ; intr, p1 ; p2 ; intr)
fake2(p1 ; p2 ; intr, p1 ; p2 ; intr)
fake3(p1 ; p2 ; intr, p1 ; p2 ; intr)
fake4(p1 ; p2 ; intr, p1 ; p2 ; intr)
send1(p1 ; p2 ; intr, p1 ; p2 ; intr)
(nonces: empty) (nw: empty) (rand: seed).
```

7. Experimental Results

In this section, we conduct four experiments to show the superiority of the proposed strategy to other existing strategies. To the best of our knowledge, there are three existing strategies for the translation from CafeOBJ equational theory specifications into Maude rewriting theory specifications. One straightforward translation strategy (TS1) is described in [14]. The basic idea is to construct a rewriting rule for the transition from an arbitrary state υ to each of its successors. For instance, the following rule specifies a class of transitions denoted by enter:

$\upsilon \Rightarrow \texttt{enter}(\upsilon, y)$ **if** $\texttt{c-enter}(\upsilon, y)$.

To make the rule executable, we add to the both sides of the rule an additional term pid(y ys) which serves as action components in our strategy. Hence, we obtain an executable rule:

$\texttt{pid}(y\ ys)\ \upsilon \Rightarrow \texttt{pid}(y\ ys)\ \texttt{enter}(\upsilon, y)$ **if** $\texttt{c-enter}(\upsilon, y)$.

Another translation strategy (TS2) is proposed in [7]. States in generated specifications by TS2 are explicitly represented, and only one rewriting rule needs to be constructed for a class of transitions. However, additional components are introduced into states to represent transitions, which drastically increases the scale of the terms representing states, and hence leads to the inefficiency of translated specifications. Another improved strategy (TS3) is proposed in [8]. All possible transitions are first enumerated. For each transition, a rewriting rule is constructed. Hence, it is not necessary to introduce additional components to represent transitions. The scale of the terms representing states is reduced, and hence the efficiency of translated specifications is improved to some extent. However, because the number of rewriting rules is increased, the efficiency is still beyond practical use.

We show the superiority of our proposed strategy (denoted by TS4 for convenience) to the other three existing

[†]This is possible because CafeOBJ allows operator overloading.

strategies in terms of the efficiency of generated specifications, and the scale of EQT Specs that can be dealt with by the proposed strategy. We also compare the efficiency of generated specifications by TS4 with those that are manually developed. The efficiency is measured by the time that Maude takes to search a bounded depth of state spaces with corresponding specifications. We choose a mutual exclusion protocol called *Tlock*, NSPK, Alternating Bit Protocol (ABP) [15] and Suzuki-Kasami Protocol (SKP) [16] as benchmarks.

Tlock is a mutual exclusion protocol using *atomicInc*, which atomically increases the number stored in a variable and returns the old number. We use Maude to model check Mutual Exclusion Property of Tlock (there is always at most one process in a critical section at any moment), with the specifications generated by TS2, TS3, TS4 together with a manually developed one. Although Tlock is a simple protocol, it can be used as a benchmark because all the four translation strategies can be applied to the EQT Spec of Tlock, and generated specifications by TS2, TS3, and TS4 can be model checked by Maude in reasonable time. The specification generated by TS1 is not used, because Maude cannot finish searching in reasonable time with it. We neglect the comparison with TS1 in the following cases for the same reason. Figure 5 shows the time that Maude spends on searching bounded depths of state spaces with the generated specifications by TS2, TS3, TS4, and the manually developed one. With the increase of depth, the time spent on searching also increases. The figure shows that the time taken with specifications generated by TS2 and TS3 increases more drastically than the one with the generated specification by TS4 or the manually developed one. It also shows the generated specification by TS4 is almost as efficient as the manually developed one.

The second experiment is about ABP, which is a simplified communication protocol of Transmission Control Protocol (TCP), aiming at providing a reliable communication channel over an unreliable channel (for details about ABP refer to [15]). We use the generated specifications of ABP together with the manually developed one to model check Reliable Communication Property of ABP (whenever the n^{th} packet is delivered to the receiver, all packets up to the n^{th} one has been delivered without any duplication nor reordering). Figure 6 shows efficiencies of the generated specifications by TS2, TS3, and TS4, as well as the manually developed one. The efficiencies are measured by the time which Maude spends on searching bounded depths of state space to model check Reliable Communication Property with these specifications. The efficiencies of the specifications generated by TS2 and TS3 are almost the same, while the generated specification by TS4 is as efficient as the manually developed one. However, the efficiency of generated specification by TS4 is just slightly improved than other two generated specifications. That is because in the EQT Spec of ABP, all observers and actions take the system state as their only parameter. No matter with which strategy, only one rewriting rule needs to be constructed, and no additional components need to be introduced into states. Therefore, the numbers of rewriting rules in all generated specifications are the same, and the scales of terms representing states are similar. The only difference is some redundant observable components are removed in the optimization phase in TS4, which makes the generated specification be model checked faster.

We model check Secrecy Property of NSPK (nonces cannot be leaked to intruders) with the specifications generated by TS4 and the manually developed one. Maude fails to admit the specifications generated by TS2, because of the huge scale of the terms representing states. For instance, 3 principals and 2 random numbers lead to 18 nonces and 32,706 messages, which drastically increase the number of transitions. Consequently, it leads to the huge scale of terms to represent these transitions in states. Similarly, it leads to the huge number of rewriting rules in the specification generated by TS3, which cannot be admitted by Maude either. Hence, Fig. 7 only shows the efficiencies of the generated specification by TS4 and the manually developed one for NSPK. The efficiencies are measured by the time which Maude spends on model checking Secrecy Property with

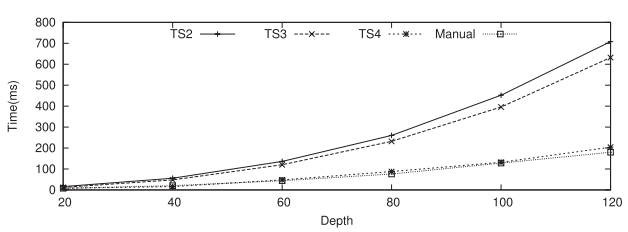


Fig. 5 Time on model checking Mutual Exclusion Property of Tlock with RWT Specs of Tlock generated by different strategies and the manually developed one.

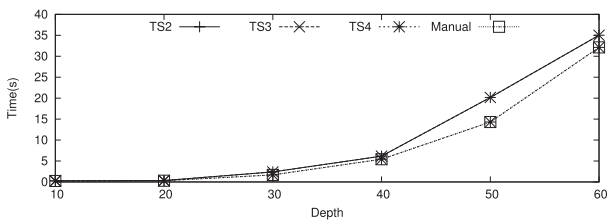


Fig. 6 Time on model checking Reliable Communication Property of ABP with RWT Specs of ABP generated by different strategies and the manually developed one.

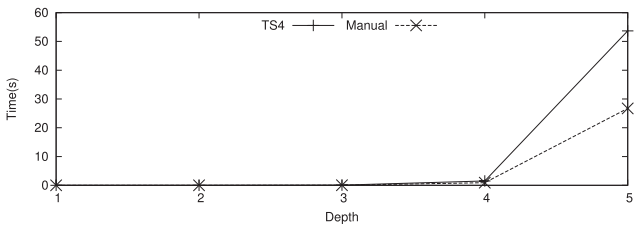


Fig. 7 Time on model checking Secrecy Property of NSPK with the RWT Spec of NSPK generated by TS4 and the one manually developed.

different depths. In the first four depths, the time spent on the two specifications is almost the same. However, when the depth is set 5, the time increase drastically, especially with the generated specification by TS4. That is because the number of searched states drastically increases from 33,091 to 710,899 when the depth is increased from 4 to 5 both with the two specifications. Maude fails to finish model checking in reasonable time when the depth is set 6 with both of the two specifications due to the huge scale of state space. The figure also shows that Maude spends more time on model checking with the generated specification by TS4 than with the manually developed one. The reason is that some optimizations cannot be automatically done in the translation, but can be manually done. For example, if the condition of a rewriting rule is $\neg(x = u)$, the rule cannot be automatically transformed into an unconditional rule. However, it is worthwhile to sacrifice the efficiency to some extent for the sake of automaticity.

Another experiment is about SKP, which is a distributed algorithm to achieve mutual exclusion in a computer network, where a fixed number of nodes communicate with each other only by exchanging messages. The mutual exclusion means that at most one node is allowed to stay in its critical section at any moment. The basic idea of SKP is to transfer the privilege for entering critical sections (for details refer to [16]). We model check Mutual Exclusion Property of SKP with the generated specifications. However, TS2 and TS3 fail to generate Maude specifications of SKP from the corresponding EQT Spec. That is because in the EQT Spec of SKP, there are two parameters which are taken by two actions, corresponding to infinite sets of elements. Consequently, it is not possible to enumerate all transitions and to represent them in states or with rewriting rules. TS4 avoids initializing the two parameters thanks to optimization. Figure 8 only shows the efficiencies of the generated specification by TS4 and the manually developed one. We can tell that the efficiencies of them are very close.

The example of SKP shows that TS4 is superior to other existing strategies when some parameters that are taken by actions correspond to infinite set of elements and cannot be initialized. For instance, assuming that an action takes a multiset of processes as its parameter, we cannot enumerate all mulitsets of processes, even if the number of processes are fixed[†].

To sum up, efficiencies of generated specifications by TS4 are indeed significantly improved, compared with those

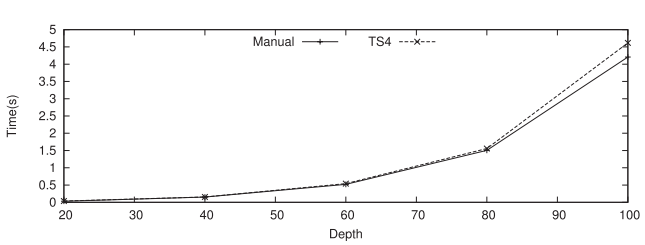


Fig. 8 Time on model checking Mutual Exclusion Property of SKP with the RWT Spec of SKP generated by TS4 and the manually developed one.

by TS1, TS2, and TS4 such as in Tlock, ABP and NSPK. TS4 also can be applied to some specifications to which TS2 and TS3 cannot be applied, such as in SKP. Unlike manual translation, the efficiency of generated specifications by TS4 also depends upon the form of original EADS Specs. Some advices have been proposed on how to develop EADS Specs for the generation of more efficient RWT Specs [1], such as specifying the conditions in conjunction forms with each conjunct represented with equivalence relations so that optimizations can be fully applied, etc. Then, we can obtain an automatically generated RWT Spec which is competitively as efficient as the manually developed one. Although in some situations the generated specifications by TS4 may be slightly less efficient than those manually developed, it is worthwhile to sacrifice a little efficiency for the sake of automaticity.

8. Related Work

Several alternative approaches to the integration of model checking and theorem proving have emerged in recent years. Among them, many verification systems are loosely or tightly integrated based on specification translation. For instance, SAL can be used as a framework for integrating different symbolic analysis techniques including theorem proving and model checking [17]. The basic idea of SAL is to provide a small intermediate language for describing transition systems, and to achieve integrations by specification translations among different formalisms via the intermediate language.

Another application of specification translation is the integration of the proof assistant PVS with a BDD-based model checker, which is achieved by translating PVS into μ-calculus [18]. However, only finite-state fragments of the PVS language are translated into μ-calculus. Whenever model checking a property, we need to translate the related fragment for the property. In our approach, the specification of state machines (possibly with infinite state space) is entirely translated from CafeOBJ equational theories into Maude rewriting theories, which allows us to model check different properties with the translated specifications. Recently, the ProZ tool that is used to validate high-level Z specifications is integrated into ProB animator, by translating Z into B [19]. Raise Specification Language (RSL) is translated into CSPM, in order to use the model checker FDR to model check the LTL formulae that are expressed in RAISE [20].

In a narrow sense, although several strategies have been proposed to translate specifications from CafeOBJ into Maude for the collaboration of the two verification systems [7], [8], the inefficiency problem of translated specifications is always the bottleneck in practical verifications. Our methodology is to investigate a specific class of specifications in CafeOBJ as a workaround, without the loss of

[†]We may prove that all such multisets that are actually needed in the specifications are enumerable.

practicability. After such a class of specifications is found, the following work such as the translation strategy and tool support becomes easier.

9. Conclusion and Future Work

We have shown that not all EQT Specs can be translated into RWT Specs, and hence investigated a specific class of EQT Specs called EADS Specs which can be translated into RWT Specs. We have proposed a strategy for the translation of EADS Specs into RWT Specs, and implemented a translator YAST to automate the translation. A case study has been presented to demonstrate how YAST works. We have also conducted four experiments. The analysis of the experimental result shows the superiority of the proposed strategy to other three existing strategies, in terms of the efficiency of generated specifications and the scale of EQT Specs which can be dealt with by the proposed strategy. The experimental results also show for which kind of EADS Specs the proposed strategy is superior to others. The superiority of the proposed strategy together with the tool support YAST can save us duplicate effort on specifying state machines in both equational theories and rewrite rules when we need to incorporate CafeOBJ and Maude for system verifications.

Since the proposed translation strategy is straightforward thanks to its domain, i.e. EADS Specs, there is a clear correspondence between an input EADS Spec and the generated RWT Spec by the strategy. However, it is preferable to prove the correspondence in theory. One possible correspondence for falsification such as IGF is that whenever a state transition chain can be represented in the generated RWT Spec, the state transition chain can also be represented in the original EADS Spec. For instance, we assume there are two processes represented by constants `p1` and `p2` of `Pid` in $\mathfrak{S}'_{\mathrm{ME}}$. We have a state transition chain as follow:

```
init(p1;p2) ↪
(locked:true)(pc[p1]:cs)(pc[p2]:rs) ↪
(locked:false)(pc[p1]:rs)(pc[p2]:rs)↪
(locked:true)(pc[p1]:rs)(pc[p2]:cs),
```

where, term `init(p1;p2)` can be reduced to its canonical form `(locked:false)(pc[p1]:rs)(pc[p2]:rs)`, representing the state $\{locked = false,\ pc_1 = rs,\ pc_2 = rs\}$. The state satisfies the equations in $\mathcal{S}_{\mathrm{ME}}$ declared for `init`, and hence `init` corresponds to `init(p1;p2)`. Similarly, term `enter(init,p1)` in $\mathcal{S}_{\mathrm{ME}}$ corresponds to `(locked:true)(pc[p1]:cs)(pc[p2]:rs)`. Since the effective condition represented by `c-enter(S,p1)` holds with `S` being `init`, there is a transition from `init` to `enter(init,p1)`. The transition corresponds to the first step in the above state transition chain. Consequently, we have the following state transition chain in $\mathcal{S}_{\mathrm{ME}}$ to correspond to the above chain:

```
init ↪ enter(init,p1) ↪
exit(enter(init,p1),p1) ↪
enter(exit(enter(init,p1),p1),p2).
```

The basic idea of proving the correspondence is to show that the original EADS Spec can simulate the generated RWT Spec. We first show that for an arbitrary initial state t_0 in the RWT Spec, there exists an initial state υ_0 in the EADS Spec corresponding to t_0, in that the state represented by t_0 is also represented by υ_0. We then show that for arbitrary states t_i, t_{i+1} in the RWT Spec and an arbitrary state υ_i in the EADS Spec such that t_{i+1} is successor state of t_i and υ_i corresponds to t_i, there exists a state υ_{i+1} in the EADS Spec such that υ_{i+1} is a successor state of υ_i and corresponds to t_{i+1}. Then we can claim that the original EADS Spec simulates the generated RWT Spec. It is a piece of our future work to complete the proof in theory.

References

[1] M. Zhang, K. Ogata, and M. Nakamura, "Specification translation of state machines from equational theories into rewrite theories," 12th ICFEM, LNCS 6447, pp.678–693, 2010.
[2] R. Diaconescu and K. Futatsugi, CafeOBJ Report, World Scientific, 1998.
[3] M. Clavel, F. Duran, S. Eker, P. Lincoln, N. Marti-Oliet, J. Meseguer, and C. Talcott, "All about Maude," LNCS 4350, Springer, 2007.
[4] K. Ogata and K. Futatsugi, "Proof scores in the OTS/CafeOBJ method," FMOODS '03, LNCS 2884, pp.170–184, Springer, 2003.
[5] K. Ogata and K. Futatsugi, "Some tips on writing proof scores in the OTS/CafeOBJ method," Essays Dedicated to Jeseph A. Goguen, LNCS 4060, pp.596–615, 2006.
[6] K. Ogata, M. Nakano, W. Kong, and K. Futatsugi, "Induction-guided falsification," 8th ICFEM, LNCS 4260, pp.114–131, 2006.
[7] W. Kong, K. Ogata, T. Seino, and K. Futatsugi, "A lightweight integration of theorem proving and model checking for system verification," 12th APSEC, pp.59–66, 2005.
[8] M. Nakamura, W. Kong, K. Ogata, and K. Futatsugi, "A specification translation from behavioral specifications to rewrite specifications," IEICE Trans. Inf. & Syst., vol.E91-D, no.5, pp.1492–1503, May 2008.
[9] M. Sipser, Introduction to the Theory of Computation, PWS Pub. Co., 1996.
[10] K. Ogata, M. Nakano, M. Nakamura, and K. Futatsugi, "Chocolat/SMV: A translator from CafeOBJ into SMV," 6th PDCAT, pp.416–420, 2005.
[11] M. Clavel, F. Durán, S. Eker, J. Meseguer, and M.O. Stehr, "Maude as a formal meta-tool," FM '99, LNCS, pp.1684–1703, Springer, 1999.
[12] F. Durán and P. Ölveczky, "A guide to extending full Maude illustrated with the implementation of Real-Time Maude," 7th WRLA, ENTCS, vol.238, no.3, pp.83–102, 2009.
[13] R. Needham and M. Schroeder, "Using encryption for authentication in large networks of computers," Commun. ACM, vol.21, no.12, pp.993–999, 1978.
[14] M. Zhang and K. Ogata, "Modular implementation of a translator from behavioral specifications to rewrite theory specifications," 9th QSIC, pp.406–411, 2009.
[15] K. Bartlett, R. Scantlebury, and P. Wilkinson, "A note on reliable full-duplex transmission over half-duplex links," Commun. ACM, vol.12, no.5, pp.260–261, 1969.
[16] I. Suzuki and T. Kasami, "A distributed mutual exclusion algorithm," TOCS, vol.3, no.4, pp.344–349, 1985.
[17] L. De Moura, S. Owre, H. Rueß, J. Rushby, N. Shankar, M. Sorea, and A. Tiwari, "SAL 2," 16th CAV, pp.251–254, Springer, 2004.
[18] S. Rajan, N. Shankar, and M. Srivas, "An integration of model checking with automated proof checking," 7th CAV, pp.84–97, Springer, 1995.

[19] D. Plagge and M. Leuschel, "Validating Z specifications using the ProB animator and model checker," 6th IFM, LNCS 4591, pp.480–500, 2007.
[20] P. Vargas, A. Garis, S. Tarifa, and C. George, "Model checking LTL formulae in RAISE with FDR," 7th IFM, LNCS 5423, p.245, 2009.

Min Zhang is presently a Ph.D. student at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his M.S. degree in computer science from Department of Computer Science & Engineering, SJTU (Shanghai Jiao Tong University). His current research interests include formal methods, algebraic specification, and automatic translation.

Kazuhiro Ogata is an associate professor at Graduate School of Information Science, JAIST (Japan Advanced Institute of Science and Technology). He received his Ph.D. in engineering from Graduate School of Science and Technology, Keio University in 1995. He was a research associate at JAIST from 1995 to 2001, a researcher at SRA Key Technology Laboratory, Inc. from 2001 to 2002, a research expert at NEC Software Hokuriku, Ltd. from 2002 to 2006, and a research associate professor at JAIST from 2006 to 2009. Among his research interests are software engineering, formal methods and formal verification.

Masaki Nakamura is an assistant professor at School of Electrical and Computer Engineering, College of Science and Engineering, Kanazawa University. He received his Ph.D. in Information Science from JAIST (Japan Advanced Institute of Science and Technology) in 2002. He was an assistant professor at Graduate School of Information Science, JAIST from 2002 to 2008. His research interest includes software engineering, formal methods, algebraic specification and term rewriting.