JAIST Repository
https://dspace.jaist.ac.jp/

Title	Specification Translation of State Machines from Equational Theories into Rewrite Theories
Author(s)	Zhang, Min; Ogata, Kazuhiro; Nakamura, Masaki
Citation	Lecture Notes in Computer Science, 6447/2010: 678-693
Issue Date	2010-11-09
Type	Journal Article
Text version	author
URL	http://hdl.handle.net/10119/9946
Rights	This is the author-created version of Springer, Min Zhang, Kazuhiro Ogata and Masaki Nakamura, Lecture Notes in Computer Science, 6447/2010, 2010, 678-693. The original publication is available at www.springerlink.com, http://dx.doi.org/10.1007/978-3-642-16901-4_44
Description	

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Specification Translation of State Machines from Equational Theories into Rewrite Theories

Min Zhang[1], Kazuhiro Ogata[1], and Masaki Nakamura[2]

[1] School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
{zhangmin,ogata}@jaist.ac.jp
[2] School of Electrical and Computer Engineering
Kanazawa University
masaki-n@is.t.kanazawa-u.ac.jp

Abstract. Specifications of state machines in CafeOBJ are called equational theory specifications (EQT Specs) which are based on equational logic, and in Maude are called rewrite theory specifications (RWT Specs) which are based on rewriting logic. The translation from EQT Specs to RWT Specs achieves the collaboration between CafeOBJ's theorem proving facilities and Maude's model checking facilities. However, translated specifications by existing strategies are of inefficiency and rarely used for model checking in practice. This paper defines a specific class of EQT Specs called EADS Specs, and proposes a strategy for the translation from EADS Specs to RWT Specs. It is proved that translated specifications by the strategy are more efficient than those by existing strategies.

Keywords: Algebraic specification, automatic translation, rewrite theory, equational theory, CafeOBJ, Maude

1 Introduction

Specification translation is a traditional way of achieving the collaboration between different verification tools, with duplicate effort reduced at the specification level. Translations between different formalisms have been widely studied. For instance, the translation from Z into B [1] integrates the tool PROZ for Z specifications into PROB; safe Petri Nets are translated into statecharts to enable the automated exchange of models between Petri net and statechart tools [2]; and Raise Specification Language (RSL) is translated into CSPM so that LTL formulae in RAISE can be model checked by the model checker FDR [3].

CafeOBJ [4] and Maude [5] are two state-of-the-art verification systems based on algebraic approaches. CafeOBJ is equipped with theorem proving facilities [6], while Maude with model checking facilities. Whenever a property fails to be proved in CafeOBJ, a counterexample is desired. In this situation, Maude is a better alternative than other model checking tools for the following reasons: (1) it is a sister language of CafeOBJ and has similar syntax, which reduces duplicate effort at specification level, and (2) the efficiency of Maude model checking facilities is comparable to those of other prevalent tools like SPIN [7].

Specifications of state machines in CafeOBJ are equational theory specifications (EQT Specs), and in Maude are rewrite theory specifications (RWT Specs). There are multiple styles of equational theory or rewrite theory specifications of state machines. EQT Specs in this paper only refer to a class of specifications that are developed in OTS/CafeOBJ method [8], and RWT Specs to a sub-class of rewrite theory specifications where states are represented by sets of observable components and action components (see Section 2.3 for details). Automatic translation from EQT Specs to RWT Specs is much more preferable because manually developing an RWT Spec for the state machine that is specified by an EQT Spec is not only effort-consuming, but at risk of causing inconsistencies between the EQT Spec and the RWT Spec. Recently, studies on the specification translation of state machines between the two formalisms have been conducted. Three strategies have been proposed so far to automate the translation and translators have been developed [9,10,11]. However, specifications generated by these strategies are rarely used in practice for model checking due to the low efficiency of the translated specifications.

This paper proposes a translation strategy for a specific sub-class of EQT Specs, aiming at generating more efficient model checkable RWT Specs. We argue that not all EQT Specs can be translated into RWT Specs, and hence introduce a specific class of EQT Specs called EADS Specs from a practical point of view. EADS Specs are mainly used to specify a class of asynchronous systems called Extended Asynchronous Distributed Systems (EADS). We compare the efficiencies of translated specifications that are obtained in different strategies with two concrete examples. The experimental result indicates that the efficiency of translated specifications is significantly improved. The contributions of this work are manifold: (1) a specific class (EADS Specs) of EQT Specs that are used for practical verifications are discovered; (2) a translation strategy is proposed to automate the translation from EADS Specs into RWT Specs; and (3) the efficiency of translated specifications is significantly improved so that they can be used by Maude for practical model checking.

The rest of this paper is organized as follows: Section 2 introduces state machines, EQT Specs and RWT Specs. Section 3 introduces EADS Specs and explains the reason why EADS Specs are selected. Section 4 describes a strategy for the translation from EADS Specs into RWT Specs. The efficiency of the specifications generated by our strategy is evaluated through comparing with those generated by three existing strategies in Section 5. Section 6 concludes this paper and mentions ongoing work.

2 Preliminaries

2.1 State machines

A state machine consists of (1) a set $\mathcal{U}$ of states, (2) the set $\mathcal{I}(\mathcal{I} \subseteq \mathcal{U})$ of initial states, and a set $\mathcal{T}$ of transitions. Each $u \in \mathcal{U}$ is a (possibly infinite) record $\{l_1 = d_1, l_2 = d_2, \ldots\}$ of type $\{l_1 : D_1, l_2 : D_2, \ldots\}$, where D with a subscript such as D_i is a type for data. 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.

Let us consider a mutual exclusion protocol called *Qlock* to show how to model dynamic systems with state machines. Multiple processes participate in Qlock and each process p executes the following program:

Loop
rs: *enqueue(queue,p);*
ws: **repeat until** *top(queue) = p;*
critical section;
cs: *dequeue(queue);*

The *queue* records the processes that are requesting to enter the critical section according to the request order. Initially, all processes are at label rs, and the shared queue is empty. A process p puts its process identifier at the bottom of the queue and then waits to enter the critical section. It is allowed to enter the critical section whenever its identifier is at the top of the queue and it is at the label ws. The process executes the *dequeue* operation on the shared queue when it leaves the critical section.

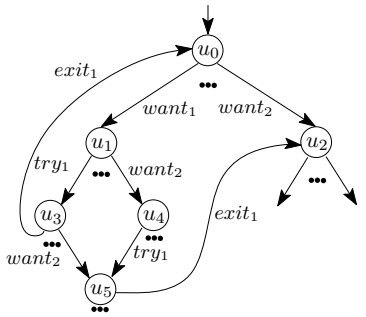
Let *Queue*, *Label* and *Pid* be types respectively for queues of process identifiers, labels (rs, ws and cs) and process identifiers $(p_1, p_2, \ldots)$. A state machine $\mathcal{M}_{\text{Qlock}}$ modeling Qlock is as follows:

- $\mathcal{U}_{\text{Qlock}} \triangleq \{u | u : \{queue : Queue, pc_1 : Label, pc_2 : Label, \ldots\}\}$;
- $\mathcal{I}_{\text{Qlock}} \triangleq \{u_0 \in \mathcal{U}_{\text{Qlock}} | queue(u_0) = empty, pc_i(u_0) = rs \; for \; each \; process \; p_i\}$;
- $\mathcal{T}_{\text{Qlock}} \triangleq \{want_1, want_2, \ldots\} \cup \{try_1, try_2, \ldots\} \cup \{exit_1, exit_2, \ldots\}$.
  - $(u, u') \in want_i$ iff $pc_i(u) = rs, pc_i(u') = ws, queue(u') = (p_i | queue(u))$ and $pc_j(u') = pc_j(u)$ for each process p_j s.t. $p_j \neq p_i$;
  - $(u, u') \in try_i$ iff $pc_i(u) = ws, top(queue(u)) = k, pc_i(u') = cs, queue(u) = queue(u')$ and $pc_j(u') = pc_j(u)$ for each $p_j \neq p_i$;
  - $(u, u') \in exit_i$ iff $pc_i(u) = cs, queue(u') = dequeue(queue(u)), pc_i(u') = rs$ and $pc_j(u') = pc_j(u)$ for each $p_j \neq p_i$.

Fig. 1 shows a part of state transitions in $\mathcal{M}_{\text{Qlock}}$. An arrow labelled by a transition t from a state u to u' denotes $(u, u') \in t$. Elements in queues are concatenated by $|$. The rightmost element is taken as the top one in the queue.

2.2 EQT Specs

EQT Specs are based on equational logic in the sense that transitions are specified with a set of equations. Let Υ be a sort for states. An EQT Spec consists of (1) a finite set $\mathcal{O}$ of observers, (2) a constant *init* of Υ, representing an arbitrary initial state, (3) a finite set $\mathcal{A}$ of actions, and (4) a family $\mathcal{E}$ of sets of equations. Each observer o is a function symbol 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 in state machines. Each action a is a function symbol whose rank is $\Upsilon \; D_{a1} \ldots D_{an} \rightarrow \Upsilon$. An action a represents a (possibly infinite) set of transitions in state machines.



$u_0 = \{queue = empty, pc_1 = rs, pc_2 = rs, \ldots\}$
$u_1 = \{queue = p_1, pc_1 = ws, pc_2 = rs, \ldots\}$
$u_2 = \{queue = p_2, pc_1 = rs, pc_2 = ws, \ldots\}$
$u_3 = \{queue = p_1, pc_1 = cs, pc_2 = rs, \ldots\}$
$u_4 = \{queue = p_2 | p_1, pc_1 = ws, pc_2 = ws, \ldots\}$
$u_5 = \{queue = p_2 | p_1, pc_1 = cs, pc_2 = ws, \ldots\}$

Fig. 1. State transitions in the state machine of *Qlock*

Each action a is given a function symbol *c-a* with the same arity of a and Bool as its coarity, denoting the condition under which a transition represented by action a takes place. $\mathcal{E}$ consists of a set $\mathcal{E}_{init}$ of equations which *init* must satisfy, and a set $\mathcal{E}_a$ of equations for each action a which can be interpreted as the definition of a set of transitions denoted by a.

We take an EQT Spec $\mathcal{S}_{\text{Qlock}}$ for $\mathcal{M}_{\text{Qlock}}$ as an example. Let `Pid`, `Queue`, and `Label` be sorts for process identifiers, queues and labels[1]. Function symbols `enqueue`, `dequeue` and `top` correspond to basic functions *enqueue*, *dequeue* and *top* on type Queue. Constants `rs`, `ws` and `cs` are of sort `Label`, corresponding to labels rs, ws and cs, respectively.

$\mathcal{S}_{\text{Qlock}}$

$\mathcal{O}_{\text{Qlock}} \triangleq \{\texttt{pc} : \Upsilon\ \texttt{Pid} \to \texttt{Label}, \texttt{queue} : \Upsilon \to \texttt{Queue}\}$;
$\mathcal{A}_{\text{Qlock}} \triangleq \{\texttt{want} : \Upsilon\ \texttt{Pid} \to \Upsilon, \texttt{try} : \Upsilon\ \texttt{Pid} \to \Upsilon, \texttt{exit} : \Upsilon\ \texttt{Pid} \to \Upsilon\}$;
$\mathcal{E}_{\text{Qlock}} \triangleq \{\mathcal{E}_{\texttt{init}}, \mathcal{E}_{\texttt{want}}, \mathcal{E}_{\texttt{try}}, \mathcal{E}_{\texttt{exit}}\}$, where:

- $\mathcal{E}_{\texttt{init}} \triangleq \{\texttt{pc}(\texttt{init}, x) = \texttt{rs}, \texttt{queue}(\texttt{init}) = \texttt{empty}\}$
- $\mathcal{E}_{\texttt{want}} \triangleq \{\texttt{c-want}(v, y) = \texttt{pc}(v, y) \doteq \texttt{rs}$
  $\texttt{pc}(\texttt{want}(v, y), x) = (\textbf{if}\ x \doteq y\ \textbf{then}\ \texttt{ws}\ \textbf{else}\ \texttt{pc}(v, x)\ \textbf{fi})\ \textbf{if}\ \texttt{c-want}(v, y)$
  $\texttt{queue}(\texttt{want}(v, y)) = (y | \texttt{queue}(v))\ \textbf{if}\ \texttt{c-want}(v, y)$
  $\texttt{want}(v, y) = v\ \textbf{if}\ \neg\texttt{c-want}(v, y)\}$
- $\mathcal{E}_{\texttt{try}} \triangleq \{\texttt{c-try}(v, y) = \texttt{pc}(v, y) \doteq \texttt{ws} \wedge \texttt{top}(\texttt{queue}(v)) \doteq y$
  $\texttt{pc}(\texttt{try}(v, y), x) = (\textbf{if}\ x \doteq y\ \textbf{then}\ \texttt{cs}\ \textbf{else}\ \texttt{pc}(v, x)\ \textbf{fi})\ \textbf{if}\ \texttt{c-try}(v, y)$
  $\texttt{queue}(\texttt{try}(v, y)) = \texttt{queue}(v)\ \textbf{if}\ \texttt{c-try}(v, y),$
  $\texttt{try}(v, y) = v\ \textbf{if}\ \neg\texttt{c-try}(v, y)\}$
- $\mathcal{E}_{\texttt{exit}} \triangleq \{\texttt{c-exit}(v, y) = \texttt{pc}(v, y) \doteq \texttt{cs}$
  $\texttt{pc}(\texttt{exit}(v, y), x) = (\textbf{if}\ x \doteq y\ \textbf{then}\ \texttt{rs}\ \textbf{else}\ \texttt{pc}(v, x)\ \textbf{fi})\ \textbf{if}\ \texttt{c-exit}(v, y)$
  $\texttt{queue}(\texttt{exit}(v, y)) = \texttt{dequeue}(\texttt{queue}(v))\ \textbf{if}\ \texttt{c-exit}(v, y)$
  $\texttt{exit}(v, y) = v\ \textbf{if}\ \neg\texttt{c-exit}(v, y)\}$

where v is a variable of Υ, and x, y are of `Pid`. Symbol $\doteq$ denotes equivalence relations over data types. Every variable in an equation (or a rewriting rule) is universally quantified and its scope is in the equation (or the rewriting rule).

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

The observer `queue` specifies the field *queue* : *Queue* in the record type in $\mathcal{M}_{\text{Qlock}}$, and `pc` specifies an infinite set of fields $\{pc_1 : Label, pc_2 : Label, \ldots\}$. Constant `init` together with $\mathcal{E}_{\texttt{init}}$ specifies $\mathcal{I}_{\text{Qlock}}$. Action `want` corresponds to an infinite set $\{want_1, want_2, \ldots\}$ of transitions. The set $\mathcal{E}_{\texttt{want}}$ of equations can be interpreted as the definition of the set of transitions. Actions `try` and `exit` together with $\mathcal{E}_{\texttt{try}}$ and $\mathcal{E}_{\texttt{exit}}$ specify the sets of transitions $\{try_1, try_2, \ldots, \}$ and $\{exit_1, exit_2, \ldots, \}$. Term $\texttt{want}(v, y)$ represents a successor of the state denoted by v if $\texttt{c-want}(v, y)$ holds. Otherwise, $\texttt{want}(v, y)$ is considered equivalent to v.

States in $\mathcal{M}_{\text{Qlock}}$ are represented by terms of Υ. For example, states u_0, u_1, and u_4 as shown in Fig. 1 are represented by terms `init`, $\texttt{want}(\texttt{init}, p_1)$, and $\texttt{want}(\texttt{want}(\texttt{init}, p_1), p_2)$, respectively. Taking u_1 for instance, we have $pc_1(u_1) = ws$. Term $\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_1)$ equals `ws` for the following reasons. According to the second equation in $\mathcal{E}_{\text{want}}$ with v being `init`, x and y being p_1, we have

$$\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_1) = (\textbf{if } p_1 \doteq p_1 \textbf{ then ws else } \texttt{pc}(\texttt{init}, p_1) \textbf{ fi}) \textbf{ if } \texttt{c-want}(\texttt{init}, p_1)$$

According to the first equation in $\mathcal{E}_{\texttt{init}}$, $\texttt{pc}(\texttt{init}, p_1)$ is equivalent to `rs`, which indicates $\texttt{c-want}(\texttt{init}, p_1)$ holds. Because $p_1 \doteq p_1$ holds, the right-hand side (RHS) of the equation above equals `ws`, namely that $\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_1)$ equals `ws`. Similarly, we have $\texttt{pc}(\texttt{want}(\texttt{init}, p_1), p_i)$ equals `rs` for $p_i (i > 1)$ and $\texttt{queue}(\texttt{want}(\texttt{init}, p_1))$ equals p_1.

2.3 RWT Specs

RWT Specs are based on rewriting logic in the sense that transitions are specified by rewriting rules. A state is represented as a set of components denoted by sort `State`. Components in a state can be divided into two kinds, namely *action components* and *observable components* whose sorts are `AComp` and `OComp` as subsorts of `State`. Each action component corresponds to a set of transitions in state machines, and each 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. Each observable component constructor $o[\_, \ldots, \_]:\_$ is a function symbol whose rank is $D_{o1} \ldots D_{om}\ D_o \rightarrow \texttt{OComp}$. We adopt mixfix operators in CafeOBJ and Maude. An underscore indicates the place where an argument is put. An observable component constructor corresponds to a (possibly infinite) set of fields of record type (i.e. the type of states) in a state machine. An observable component is expressed by a term whose top is $o[\_, \ldots, \_]:\_$. Each action component constructor ac is a function symbol whose rank is $SetD_{t1} \ldots SetD_{tn} \rightarrow \texttt{AComp}$, where $SetD_{ti}$ is a sort for sets of elements of D_{ti} [2]. An action component is expressed by a term

[2] Basic operations on $SetD_{tk}$ follow the definition of basic set in [5, chap. 5]. Whitespace character is defined as concatenation operation which is associative and commutative. Therefore, variable y_k of D_{tk} can be any element of D_{tk} in a pattern $(y_k\ ys_k)$, where variable ys_k is of $SetD_{tk}$.

whose top is ac, corresponding to a set of transitions in state machines. One rewriting rule in $\mathcal{R}$ specifies a (possibly infinite) set of transitions. For instance, the following rewriting rule specifies all transitions in $\{want_1, want_2, \ldots\}$

$\texttt{want}((y\ ys))\ (\texttt{queue:}\ q)\ (\texttt{pc[}y\texttt{]:}\ l) \Rightarrow$
$\texttt{want}((y\ ys))\ (\texttt{queue:}\ (y|q))\ (\texttt{pc[}y\texttt{]:}\ \texttt{ws})$ **if** $l \doteq \texttt{rs}$,

where, q is a variable of sort `Queue` and l of `Label`. A state containing the fields $queue = q'$ and $pc_i = rs$ is partially denoted by $\texttt{want}(p_i\ ys)(\texttt{queue:}\ q)(\texttt{pc[}y\texttt{]:}\ l)$. The term is rewritten into $\texttt{want}(p_i\ ys)(\texttt{queue:}\ (p_i|q'))(\texttt{pc[}p_i\texttt{]:}\ \texttt{ws})$, which means that the two corresponding data fields in a state are changed into $queue = (p_i|q')$ and $pc_i = ws$. The rewriting rule says that if there is a process y whose label is `rs` and a queue is q in a state, there is a successor state where the label of process y becomes `ws` and the queue $(y|q)$.

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

$\mathfrak{S}_{\text{Qlock}}$

$\mathcal{OC}_{\text{Qlock}} \triangleq \{\texttt{pc[\_]:\_:}\ \texttt{Pid Label} \to \texttt{OComp}, \texttt{queue:\_:}\ \texttt{Queue} \to \texttt{OComp}\}$;
$\mathcal{AC}_{\text{Qlock}} \triangleq \{\texttt{want:}\ \texttt{SetPid} \to \texttt{AComp}, \texttt{try:}\ \texttt{SetPid} \to \texttt{AComp}, \texttt{exit:}\ \texttt{SetPid} \to \texttt{AComp}\}$;
$\mathcal{F}_{\text{Qlock}} \triangleq \{\texttt{init:}\ \texttt{SetPid} \to \texttt{State}, \texttt{mk-pc:}\ \texttt{SetPid} \to \texttt{State}\}$;
$\mathcal{E}_{\mathcal{F}_{\text{Qlock}}} \triangleq \{\texttt{init}(ys) = \texttt{want}(ys)\ \texttt{try}(ys)\ \texttt{exit}(ys)\ (\texttt{queue:}\ \texttt{empty})\ \texttt{mk-pc}(ys),$
$\texttt{mk-pc}(\texttt{empty-set}) = \texttt{empty-state},$
$\texttt{mk-pc}(y\ ys) = (\texttt{pc[}y\texttt{]:}\ \texttt{rs})\ \texttt{mk-pc}(ys)\}$
$\mathcal{R}_{\text{Qlock}} \triangleq \{rw_{\texttt{want}}, rw_{\texttt{try}}, rw_{\texttt{exit}}\}$, where:

- $rw_{\texttt{want}} \triangleq \texttt{want}((y\ ys))(\texttt{pc[}y\texttt{]:}\ l)(\texttt{queue:}\ q) \Rightarrow$
  $\texttt{want}((y\ ys))(\texttt{pc[}y\texttt{]:}\ \texttt{ws})(\texttt{queue:}\ (y|q))$ **if** $l \doteq \texttt{rs}$,
- $rw_{\texttt{try}} \triangleq \texttt{try}((y\ ys))(\texttt{pc[}y\texttt{]:}\ l)(\texttt{queue:}\ q) \Rightarrow$
  $\texttt{try}((y\ ys))(\texttt{pc[}y\texttt{]:}\ \texttt{cs})(\texttt{queue:}\ q)$ **if** $l \doteq \texttt{ws}$ **and** $top(q) \doteq y$,
- $rw_{\texttt{exit}} \triangleq \texttt{exit}((y\ ys))(\texttt{pc[}y\texttt{]:}\ l)(\texttt{queue:}\ q) \Rightarrow$
  $\texttt{exit}((y\ ys))(\texttt{pc[}y\texttt{]:}\ \texttt{rs})(\texttt{queue:}\ \texttt{dequeue}(q))$ **if** $l \doteq \texttt{cs}$

The sort `SetPid` denotes sets of process identifiers. Given a term ps denoting a set of process identifiers, $\texttt{init}(ps)$ denotes the initial state when the processes participate in Qlock. Three rewriting rules $rw_{\texttt{want}}$, $rw_{\texttt{try}}$ and $rw_{\texttt{exit}}$ in $\mathcal{R}_{\text{Qlock}}$ specify three sets of transitions $\{want_1, want_2, \ldots\}$, $\{try_1, try_2, \ldots\}$, and $\{exit_1, exit_2, \ldots\}$ in $\mathcal{M}_{\text{Qlock}}$.

3 State Machines Specifiable in RWT Specs

In RWT Specs, segments of states used in rewriting rules consist of one action component, and a finite collection of observable components. When a rewriting rule is applied to a state, only a segment of the state that matches the LHS can be changed and the rest keeps unchanged. Since one observable component corresponds to an element in a state in state machines, the number of values that are changed by a rewriting rule must be finite. Hence, if a rewriting rule can be declared for a transition $(u, u') \in t$ in a state machine, the number of data fields in u that are different from their corresponding data fields in u' must

be finite, and the changes of these data fields from u to u' depends on a finite number of data fields in u.

However, the condition is 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 $i \leq l_0(s)$ then $l_i(u') = l_i(u) + 1$, otherwise, $l_i(u') = l_i(u)$. The transition chain from the initial state is like:

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

The transition inc cannot be specified in the way of RWT Specs, because the numbers of changed natural numbers from u to u' vary for all $(u, u') \in inc$, although inc can be specified in equational theories. After each transition, the number of changed natural numbers is increased and unbounded. Hence, for each transition $t \in T$ in a state machine which is specifiable in an RWT Spec, there must be only a bounded number of elements changed from u to u' for each $(u, u') \in t$. Moreover, each changed value in u' must depend upon a bounded number of elements in u. We call the state machines that satisfy the condition are *double bounded*, and corresponding EQT Specs *double bounded* EQT Specs. Any state machines that can be specified in RWT Specs are double bounded.

To automatically generate an RWT Spec from an EQT Spec, we first need to check if the EQT Spec, namely the state machine denoted by it, is double bounded. However, it is not decidable for all EQT Specs whether they are double bounded. We have such a concrete EQT Spec which cannot be decided to be double bounded or not.

Let us consider an EQT Spec $\mathcal{S}_{\text{PCP}}$ that specifies a state machine of finding solutions to Post's Correspondence Problem (PCP)[12]. Let `pcp-instance` be an arbitrary instance of PCP on the alphabet $\{a, b\}$, and `Seq` be a sort for sequences of natural numbers.

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

where, $\texttt{isSolution}(\upsilon, sq)$ denotes if sq is a solution to `pcp-instance` in υ, and $\texttt{solve}(\upsilon)$ denotes a successor of υ. The function symbol `check` denotes a function that checks if sq is a solution to `pcp-instance`. Since it is undecidable if `pcp-instance` has solutions according to the undecidability of PCP, it is also undecidable if the number of values observed by `isSolution` and changed from υ to its successor state $\texttt{solve}(\upsilon)$ is bounded.

To automate the translation from EQT Specs into RWT Specs, some constraints need to be imposed on EQT Specs. We focus on a specific class of double bounded EQT Specs called EADS Specs. Without loss of generality, we suppose that a special sort `Pid` is predefined for the processes (or principals) in dynamic systems. All EADS Specs must conform to the following syntax-level constraints:

1. Each $o \in \mathcal{O}$ should be declared in the form of $o : \Upsilon \to D_o$ or $o : \Upsilon$ `Pid` $\to D_o$;
2. If there exists $o \in \mathcal{O}$ s.t. $o : \Upsilon$ `Pid` $\to D_o$, the declaration of each $a \in \mathcal{A}$ should be one of the following two forms:
   (a) $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \to \Upsilon$, where each D_{ai} cannot be `Pid` [3];
   (b) $a : \Upsilon$ `Pid` $\{D_{a2} \dots D_{an}\} \to \Upsilon$;
   Otherwise, there is no restriction on the declaration of each $a \in \mathcal{A}$;
3. If there exists $o \in \mathcal{O}$ s.t. $o : \Upsilon$ `Pid` $\to D_o$, equations declared for t w.r.t o are in one of the following forms:
   (a) for $o : \Upsilon \to D_o$ and $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \to \Upsilon$ (D_{ai} cannot be `Pid`): $o(a(\upsilon\{, y_1, \dots, y_n\})) = T_{oa}$ **if** $c\text{-}a(\upsilon\{, y_1, \dots, y_n\})$;
   (b) for $o : \Upsilon$ `Pid` $\to D_o$ and $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \to \Upsilon$ (D_{ai} cannot be `Pid`): $o(a(\upsilon\{, y_1, \dots, y_n\}), y) = o(\upsilon, y)$;
   (c) for $o : \Upsilon \to D_o$ and $a : \Upsilon$ `Pid` $\{D_{a2} \dots D_{an}\} \to \Upsilon$: $o(a(\upsilon, y_1\{, y_2, \dots, y_n\})) = T_{oa}$ **if** $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$;
   (d) for $o : \Upsilon$ `Pid` $\to D_o$ and $a : \Upsilon$ `Pid` $\{D_{a2} \dots D_{an}\} \to \Upsilon$: $o(a(\upsilon, y_1\{, y_2, \dots, y_n\}), y) = (\textbf{if } y \doteq y_1 \textbf{ then } T_{oa} \textbf{ else } o(\upsilon, y) \textbf{ fi})$ **if** $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$;
   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 $o \in \mathcal{O}$ 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 $o' \in \mathcal{O}$ (can be o) in T_{oa} and $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$ 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 $oa(\upsilon, y_1, \{y_2, \dots, y_n\})$ and $c\text{-}a(\upsilon, y_1\{, y_2, \dots, y_n\})$.

Assume an EADS Spec specifies a state machine $\mathcal{M}$. Constraint 1 indicates that there are only two kinds of data fields in $\mathcal{M}$. One is called *system-level* data field which is denoted by $o : \Upsilon \to D_o$ and the other is *process-level* data field denoted by $o : \Upsilon$ `Pid` $\to D_o$. 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 $a : \Upsilon\ \{D_{a1} \dots D_{an}\} \to \Upsilon$ and the other is *process-level* transition represented by $a : \Upsilon$ `Pid` $\{D_{a2} \dots D_{an}\} \to \Upsilon$. 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 system-level values can be changed by system-level transitions, and equations 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 the number of terms representing data fields in T_{oa} is bounded and the result of T_{oa} depends on only these terms, namely that the change of each data field depends upon a bounded number of data fields, so does each condition for each action. Constraint 6 assures that each action can be interpreted as a transition.

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

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 Qlock. 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 construct 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 optimizations of translated RWT Specs and the construction of initial states for the optimized RWT Specs.

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

Observable component constructors $\mathcal{OC}$ and action component constructors $\mathcal{AC}$ can be directly generated from the declarations of observers $\mathcal{O}$ and actions $\mathcal{A}$. Fig. 2 shows the translation from the declarations of observers and actions to declarations of both observable and action component constructors.

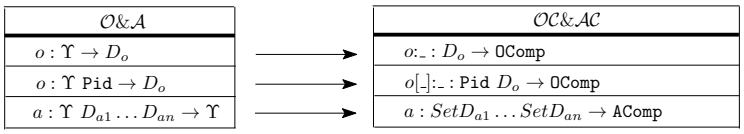


Fig. 2. The translation from $\mathcal{O}$ and $\mathcal{A}$ into $\mathcal{OC}$ and $\mathcal{AC}$

4.2 Generation of $\mathcal{R}$

For each action $a \in \mathcal{A}$ in an EADS Spec $\mathcal{S}$, we construct a rewriting rule to specify the same set of transitions that are denoted by a in a state machine.

For $a \in \mathcal{A}$ s.t. $a : \Upsilon\ D_{a1} \ldots D_{an} \to \Upsilon (n \geq 0)$, a denotes a set of system-level transitions, by which only system-level data fields can be accessed. Therefore, we only need to consider those observers that denote system-level data fields in a state υ. For $o \in \mathcal{O}$ s.t. $o : \Upsilon \to D_o$, $o(\upsilon)$ denotes the value of a system-level data field in υ. According to the equation 3a, $o(\upsilon)$ is changed into $T_{oa\downarrow\mathcal{S}}$ [4] when $c\text{-}a(\upsilon)$ holds. We introduce a fresh variable d_o of D_o denoting the value denoted by $o(\upsilon)$

[4] $T_{oa\downarrow\mathcal{S}}$ represents the canonical form of T_{oa} in context $\mathcal{S}$.

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:

$a((y_1\ ys_1), \ldots, (y_n, ys_n))(o_1\colon o_1(\upsilon)) \ldots (o_m\colon o_m(\upsilon)) \Rightarrow a(y_1\ ys_1), \ldots, (y_n, ys_n))$
$(o_1\colon T_{o_1 a\downarrow\mathcal{S}}) \ldots (o_m\colon T_{o_m a\downarrow\mathcal{S}})$ **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 (if there is) 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 Subsection 4.3 for details). Moreover, if all observers $o \in \mathcal{O}$ are declared like $o : \Upsilon \to D_o$, D_{ai} in the declaration of a can be any sort, otherwise, D_{ai} cannot be `Pid`, according to Constraint 2.

For $a \in \mathcal{A}$ s.t. $a : \Upsilon$ `Pid` $D_{a2} \ldots D_{an} \to \Upsilon (n \geq 0)$, a denotes a process-level transition. Besides system-level data fields, a process-level transition can access process-level data fields in the process where the transition takes place. 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 each $o \in \mathcal{O}$ s.t. $o : \Upsilon \to D_o$, we deal with it similarly like in the construction of rewriting rules for system-level transitions. For each $o \in \mathcal{O}$ s.t. $o : \Upsilon$ `Pid` $\to D_o$, 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 the equation 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 $o : \Upsilon \to D_o$ and rest of $o : \Upsilon$ `Pid` $\to D_o$ in m observers $\{o_1, \ldots, o_k, o_{k+1}, \ldots, o_m\}$. A rewriting rule specifying a set of system-level transitions denoted by action a and $\mathcal{E}_a$ w.r.t. $y_1, y_2, \ldots, y_n$ is as follow:

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

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

For instance, $\mathcal{S}_{\text{Qlock}}$ is an EADS Spec, according to the four restrictions. Fig. 3 shows the translation of the declarations of observers and actions in $\mathcal{S}_{\text{Qlock}}$ into the declarations of observable component constructors and action component constructors in $\mathfrak{S}_{\text{Qlock}}$. According to $\mathcal{E}_{\texttt{want}}$ in $\mathcal{S}_{\text{Qlock}}$, we construct the following rewriting rule to specify the set of transitions denoted by the action `want`:

`want`$((y\ ys))$(`pc`$[y]$`: pc`$(\upsilon, y))$ (`queue: queue`$(\upsilon))$ $\Rightarrow$ `want`$((y\ ys))$(`pc`$[y]$`:`
`pc(want`$(\upsilon, y), y)_{\downarrow\mathcal{S}_{\text{Qlock}}}$) (`queue: queue(want`$(\upsilon, y))_{\downarrow\mathcal{S}_{\text{Qlock}}}$**if** `c-want`$(\upsilon, y)_{\downarrow\mathcal{S}_{\text{Qlock}}}$.

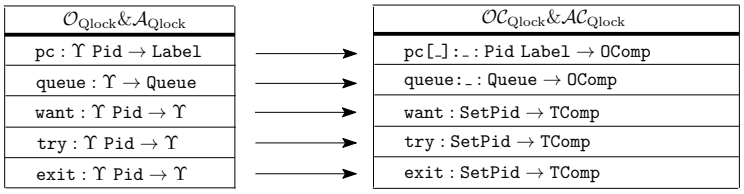


Fig. 3. The translation from $\mathcal{O}_{\text{Qlock}}$ and $\mathcal{A}_{\text{Qlock}}$ into $\mathcal{OC}_{\text{Qlock}}$ and $\mathcal{AC}_{\text{Qlock}}$

According to $\mathcal{E}_{\texttt{want}}$, $\texttt{pc(want}(\upsilon, y), y)$ is reduced to ws, $queue(\texttt{want}(\upsilon, y))$ to $(y|queue(\upsilon))$ and $\texttt{c-want}(\upsilon, y)$ to $\texttt{pc}(\upsilon, y) \doteq \text{rs}$. Consequently, we obtain the following rewriting rule:

$\texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ pc(\upsilon, y))\ (\texttt{queue:}\ queue(\upsilon)) \Rightarrow \texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ \texttt{ws})$
$(\texttt{queue:}\ (q|queue(\upsilon)))$ **if** $\texttt{pc}(\upsilon, y) \doteq \texttt{rs}$.

Further, we substitute l for $pc(\upsilon, y)$ and q for $queue(\upsilon)$, then we obtain the rewriting rule $rw_{\texttt{want}}$:

$\texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ l)\ (\texttt{queue:}\ q) \Rightarrow \texttt{want}((y\ ys))(\texttt{pc}[y]\texttt{:}\ \texttt{ws})\ (\texttt{queue:}\ (y|q))$
if $l \doteq \texttt{rs}$.

Similarly, we can construct the rewriting rules $rw_{\texttt{try}}$ and $rw_{\texttt{exit}}$ for the actions `try` and `exit` in $\mathcal{S}_{\text{Qlock}}$.

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 [5, 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. From the program point of view, it provides necessary variables that guarantee the rewriting rule is executable, because Maude generally requires variables that occur in the RHS or condition must occur in the LHS to make rewriting rules executable [5, 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. We take the rewriting rule $rw_{\texttt{want}}$ in $\mathfrak{S}_{\text{Qlock}}$ for instance. The variable y in $\texttt{want}((y\ ys))$ is also used in $(\texttt{pc}[y]\texttt{:}\ l)$. Since there is only one parameter taken by the action component constructor *want*, deleting x means that we can delete the whole action component. After deleting $\texttt{want}((y\ ys))$, we obtain a simpler rewriting rule, as follow:

$\texttt{(pc[}y\texttt{]: }l\texttt{)(queue: }q\texttt{)} \Rightarrow \texttt{(pc[}y\texttt{]: ws)(queue: }(y|q)\texttt{)}$ **if** $l \doteq \texttt{rs}$.

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, without changing the meaning of the 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. For instance, the rewriting rule can be further simplified to be the following one:

$\texttt{(pc[}y\texttt{]: rs)(queue: }q\texttt{)} \Rightarrow \texttt{(pc[}y\texttt{]: ws)(queue: }(q|y)\texttt{)}$.

An optimized RWT Spec of Qlock is as follow:

$\mathfrak{S}'_{\text{Qlock}}$

$\mathcal{OC}'_{\text{Qlock}} \triangleq \{\texttt{pc[\_]:\_: Pid Label} \to \texttt{OComp}, \texttt{queue:\_: Queue} \to \texttt{OComp}\}$;
$\mathcal{AC}'_{\text{Qlock}} \triangleq \emptyset$;
$\mathcal{F}'_{\text{Qlock}} \triangleq \{\texttt{init: SetPid} \to \texttt{State}, \texttt{mk-pc: SetPid} \to \texttt{State}\}$;
$\mathcal{E}'_{\mathcal{F}'_{\text{Qlock}}} \triangleq \{\texttt{init}(ys) = \texttt{(queue: empty) mk-pc}(ys),$
$\texttt{mk-pc}(\texttt{empty-set}) = \texttt{empty-state},$
$\texttt{mk-pc}(y\ ys) = \texttt{(pc[}y\texttt{]: rs) mk-pc}(ys).\}$
$\mathcal{R}'_{\text{Qlock}} \triangleq \{rw_{\texttt{want}}, rw_{\texttt{try}}, rw_{\texttt{exit}}\}$, where:
- $rw_{\texttt{want}} \triangleq \texttt{(pc[}y\texttt{]: rs)(queue: }q\texttt{)} \Rightarrow \texttt{(pc[}y\texttt{]: ws)(queue: }(y|q)\texttt{)}$;
- $rw_{\texttt{try}} \triangleq \texttt{(pc[}y\texttt{]: ws)(queue: }q\texttt{)} \Rightarrow \texttt{(pc[}y\texttt{]: cs)(queue: }q\texttt{)}$ **if** $\texttt{top}(q) \doteq y$;
- $rw_{\texttt{exit}} \triangleq \texttt{(pc[}y\texttt{]: cs)(queue: }q\texttt{)} \Rightarrow \texttt{(pc[}y\texttt{]: rs)(queue: dequeue}(q)\texttt{)}$.

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

The last step is to construct a set $\mathcal{F}$ of function symbols and a set of equations $\mathcal{E}_{\mathcal{F}}$ for $\mathcal{F}$ to specify initial states in an RWT Spec $\mathfrak{S}$, according to the specification of the initial states denoted by *init* in the original EQT Spec $\mathcal{S}$.

For each $o \in \mathcal{O}$ in $\mathcal{S}$ s.t. $o : \Upsilon \to D_o$, the value of the data field corresponding to o in initial states is $o(init)_{\downarrow\mathcal{S}}$. Consequently, an observable component $(o : o(init)_{\downarrow\mathcal{S}})$ in $\mathfrak{S}$ 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, to construct a set of observable components that represent all data fields corresponding to o in the initial states, we define an auxiliary function which is denoted by $mk\text{-}o : \texttt{PidSet} \rightarrow \texttt{State}$. The following two equations are declared for $mk\text{-}o$:

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

where y is a variable of `Pid` and ys of `SetPid`. Let $sa_i (1 \leq i \leq n)$ denote a set of elements of $SetD_{ai}$ that are used in the specified system. The action component of a in the initial states can be constructed as $a(sa_1, \ldots, sa_n)$, for each $a : SetD_{a1} \ldots SetD_{an} \rightarrow \texttt{TComp}$.

We suppose $\mathcal{OC}$ consists of m observable component constructors, where the first k constructors are declared as $o_i\texttt{:\_} : D_{oi} \rightarrow \texttt{OComp}$ for $i = 1, \ldots, k$, and the rest as $o_i\texttt{[\_]:\_} : \texttt{Pid}\ D_{oi} \rightarrow \texttt{OComp}$ for $i = k+1, \ldots, m$. We also 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. $\texttt{init} : \texttt{SetPid}\ SetD_1 \ldots SetD_{n'} \rightarrow \texttt{State}$, and declare the following equation for `init`:

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

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 $\mathcal{F} = \{\texttt{init}, mk\text{-}o_{k+1}, \ldots, mk\text{-}o_m\}$, and a set $\mathcal{E}_\mathcal{F}$ of equations that are declared for `init` and $mk\text{-}o_{k+1}, \ldots, mk\text{-}o_n$.

We take the construction of $\mathcal{F'}_{\text{Qlock}}$ and $\mathcal{E'}_{\mathcal{F'}_{\text{Qlock}}}$ for $\mathfrak{S}'_{\text{Qlock}}$ as an example. Since $\mathcal{AC}'_{\text{Qlock}}$ is empty, we only need to consider $\mathcal{OC}'_{\text{Qlock}}$. Sort `Pid` is taken as a parameter sort, therefore `init` is declared as $\texttt{init} : \texttt{PidSet} \rightarrow \texttt{State}$, and we have $\texttt{init}(ys) = (\texttt{queue:}\ queue(init)_{\downarrow\mathcal{S}_{\text{Qlock}}})\ \texttt{mk-pc}(ys)$. That is $\texttt{init}(ys) = (\texttt{queue:empty})\ \texttt{mk-pc}(ys)$, and the equations declared for $\texttt{mk-pc} : \texttt{PidSet} \rightarrow \texttt{State}$ are as follows:

$\texttt{mk-pc}(\texttt{empty-set}) = \texttt{empty-state}$
$\texttt{mk-pc}(y\ ys) = (\texttt{pc}[y]\texttt{:}\ \texttt{rs})\ \texttt{mk-pc}(ys)$.

4.5 Principles of defining EADS Specs for efficient RWT Specs

Given an extended asynchronous distributed system, we can develop one or more EADS Specs, which consequently correspond 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 $o(v\{, x_1\}) \doteq T$ if possible, where T is a term.

We take $rw_{\texttt{try}}$ in $\mathfrak{S}_{\text{Qlock}}$ for instance. To be able to remove the condition $\texttt{top}(q) \doteq y$, we need to revise the condition $\texttt{c-try}(\upsilon, y)$ in $\mathcal{S}_{\text{Qlock}}$, according to the two principles. The condition $\texttt{top}(q) \doteq y$ corresponds to $\texttt{top}(\texttt{queue}(\upsilon)) = y$ in $\mathcal{S}_{\text{Qlock}}$ which means that y is at the top of the shared queue in state υ. An equivalent condition is $\texttt{queue}(\upsilon) \doteq (y \mid q)$. Consequently, we need to revise the declaration of *try* and $\mathcal{E}_{\texttt{try}}$, like:

$try : \Upsilon$ Pid Queue $\rightarrow \Upsilon$, and
$\mathcal{E}_{\texttt{try}} \triangleq \{\texttt{c-try}(\upsilon, y, q) = \texttt{pc}(\upsilon, y) \doteq \texttt{ws} \wedge \texttt{queue}(\upsilon) \doteq (y \mid q),$
$\texttt{pc}(\texttt{try}(\upsilon, y, q), x) = \textbf{if } x \doteq y \textbf{ then } \texttt{cs} \textbf{ else } \texttt{pc}(\upsilon, x) \textbf{ fi if } \texttt{c-try}(\upsilon, y, q),$
$\texttt{queue}(\texttt{try}(\upsilon, y, q)) = \texttt{queue}(\upsilon) \textbf{ if } \texttt{c-try}(\upsilon, y, q),$
$\texttt{try}(\upsilon, y, q) = \upsilon \textbf{ if } \neg\texttt{c-try}(\upsilon, y, q)\}$

The translated rewriting rule of the modified $\mathcal{E}_{\texttt{try}}$ is:

$\texttt{try}((y\ ys), (q_1\ qs))\texttt{(pc[}y\texttt{]: }l\texttt{)(queue: }q) \Rightarrow \texttt{try}((y\ ys), (q_1\ qs))$
$\texttt{(pc[}y\texttt{]: cs)(queue: }q) \textbf{ if } l \doteq \texttt{ws} \textbf{ and } q \doteq (y \mid q_1),$

where q_1 is a variable of Queue and qs_1 of `SetQueue`. After the optimization, we obtain a much simpler rewriting rule $rw_{\texttt{try}}$:

$\texttt{(pc[}y\texttt{]: ws)(queue: }(y \mid q_1)) \Rightarrow \texttt{(pc[}y\texttt{]: cs)(queue: }(y \mid q_1))$

5 Experimental Results

To the best of our knowledge, three strategies for the translation of specifications of state machines from CafeOBJ to Maude have been proposed. An implementation of the most straightforward strategy (called TS1) is described in [11]. Another strategy (called TS2) is proposed and its implementation is described in [9]. Yet another strategy (called TS3) is proposed in [10] and its implementation is described in [11]. We take Qlock and NSPK as examples to evaluate the efficiency of the specifications translated by our proposed strategy by comparing with the three strategies. The efficiency is measured by the number of states in the state space from initial states with the same depth and the time that Maude takes to finish searching these states. We use $\mathcal{S}^{\dagger}$, $\mathcal{S}^{\ddagger}$, $\mathcal{S}^{\star}$ and $\mathcal{S}^{\divideontimes}$ to denote specifications generated by TS1, TS2, TS3 and the proposed one in this paper. $\mathcal{S}^{*}$ denotes manually developed specifications. The experiment has been conducted on Ubuntu in a laptop with 2x1.20GHz Duo Core processor and 4GB memory.

Table 1 shows that the number of states increases with the increase of depth, which consequently causes time on model checking to increase. Except in $\mathcal{S}^{\dagger}_{\text{Qlock}}$, the number of states in other specifications is the same with the same depth. This is because states in $\mathcal{S}^{\dagger}_{\text{Qlock}}$ are implicitly represented like in EQT Specs, while states in other specifications are explicitly represented by a set of components. The time spent in $\mathcal{S}^{\divideontimes}_{\text{Qlock}}$ is the least and is the same as the one in $\mathcal{S}^{*}_{\text{Qlock}}$, indicating that the efficiency of $\mathcal{S}^{\divideontimes}_{\text{Qlock}}$ is higher than other translated ones, and equal to the manually developed one. Searching fails when the depth is 9 in both $\mathcal{S}^{\ddagger}_{\text{Qlock}}$ and $\mathcal{S}^{\star}_{\text{Qlock}}$ in a reasonable time, but successfully finishes in $\mathcal{S}^{\divideontimes}_{\text{Qlock}}$,

Table 1. Times (ms) taken by Maude for checking the mutual exclusion property of Qlock with 9 processes, and the number of states traversed in different depths.

(a) *Time on model checking*

	$\mathcal{S}^{\dagger}_{\text{Qlock}}$	$\mathcal{S}^{\ddagger}_{\text{Qlock}}$	$\mathcal{S}^{\star}_{\text{Qlock}}$	$\mathcal{S}^{\divideontimes}_{\text{Qlock}}$	$\mathcal{S}^{*}_{\text{Qlock}}$
1	0	0	0	**0**	0
3	116	64	20	**8**	8
5	20361	2596	764	**268**	268
7	–	33458	16373	**5336**	5336
9	–	–	–	**42072**	42072

(b) *The number of states traversed*

	$\mathcal{S}^{\dagger}_{\text{Qlock}}$	$\mathcal{S}^{\ddagger}_{\text{Qlock}}$	$\mathcal{S}^{\star}_{\text{Qlock}}$	$\mathcal{S}^{\divideontimes}_{\text{Qlock}}$	$\mathcal{S}^{*}_{\text{Qlock}}$
1	10	10	10	**10**	10
3	748	667	667	**667**	667
5	36262	22339	22339	**22339**	22339
7	–	339859	339859	**339859**	339859
9	–	–	–	**1609939**	1609939

although the number of the states in the three specifications are the same. This is because the efficiency of a specification depends upon not only the state space, but the forms of rewriting rules in the specification, as explained in Section 4.3.

NSPK is a security protocol to achieve mutual authentication between two principals over network [13]. To generate a rewrite theory specification from the corresponding equational theory specification of NSPK, the existing translation strategies require a fixed number of nonces and messages. Hence, we need to fix both of the numbers of random numbers and principals. However, the number of messages are huge, which consequently makes the terms representing states huge. For instance, 3 principals and 2 random numbers lead to 18 nonces and 32076 different messages. Moreover, the huge number of messages drastically increases the number of rewriting rules in the target specifications, and hence it becomes impossible to reasonably model check the generated specifications. In our approach, we do not need to fix the number of nonces and messages thanks to the optimization, to generate an RWT Spec of NSPK. The generated Maude specification is denoted by $\mathcal{S}^{\divideontimes}_{\text{NSPK}}$ and the manually developed one by $\mathcal{S}^{*}_{\text{NSPK}}$.

Table 2 shows that the time that is spent on searching in $\mathcal{S}^{\divideontimes}_{\text{NSPK}}$ is more than in $\mathcal{S}^{*}_{\text{NSPK}}$, although the number of states in $\mathcal{S}^{\divideontimes}_{\text{NSPK}}$ is the same as the one in $\mathcal{S}^{*}_{\text{NSPK}}$ with the same depth. It indicates a translated specification by the proposed strategy may not be the most optimized one because some optimizations cannot be automatically done. For example, if a condition in a rewriting rule is in the form of $\neg(x \doteq u)$, we cannot automatically transform the rewriting rule into an unconditional one. It takes some more time on pattern matching. However, both of the specifications are comparably efficiently model checked.

Table 2. Times (ms) taken by Maude to model check the secrecy property for NSPK with 3 principals, and the number of states traversed in different depths.

(a) *Time on model checking*

	1	2	3	4	5
$\mathcal{S}^{*}_{\text{NSPK}}$	0	4	36	716	24617
$\mathcal{S}^{\divideontimes}_{\text{NSPK}}$	**0**	**4**	**44**	**1104**	**47654**

(b) *The number of states traversed*

	1	2	3	4	5
$\mathcal{S}^{*}_{\text{NSPK}}$	7	105	1745	33901	710899
$\mathcal{S}^{\divideontimes}_{\text{NSPK}}$	**7**	**105**	**1745**	**33901**	**710899**

6 Conclusion and Future Work

We have proposed a specific class of EQT Specs called EADS Specs, and proposed a strategy for the translation from EADS Specs into RWT Specs. Case studies have been conducted to show the efficiency of translated specifications is significantly improved. Although the strategy can only deal with EADS Specs, most of EQT Specs that are developed for practical verifications belong to this class based on our experience of theorem proving in CafeOBJ.

Regarding the correctness of the translation, we argue that a counterexample in the generated RWT Spec by the strategy $\mathfrak{S}$ of an EADS Spec $\mathcal{S}$ is also a counterexample in $\mathcal{S}$. We can prove it by show that for any state transition chain in $\mathfrak{S}$, there is a corresponding chain in $\mathcal{S}$. First, we show that for an arbitrary initial state denoted by t_0 in $\mathfrak{S}$, there exists a term (which is actually `init`) in $\mathcal{S}$, corresponding to t_0. Then we assume two arbitrary states denoted by t_i and t_{i+1} in $\mathfrak{S}$ and an arbitrary state denoted by υ_i in $\mathcal{S}$ such that t_{i+1} denotes a successor state of the one by t_i and υ_i corresponds to t_i. We can show there exists a state denoted by υ_{i+1} in $\mathcal{S}$ such that υ_{i+1} corresponds to t_{i+1}, and υ_{i+1} denotes a successor state of υ_i. Hence, we can claim that $\mathfrak{S}$ simulates $\mathcal{S}$, which indicating the correctness of the proposed translation strategy. A detailed proof in theory for the correctness of the translation is one piece of our future work. Moreover, A prototype of the present translation strategy has been implemented and successfully applied to Qlock and NSPK. We will further improve the translator, especially the optimization part, for practical applications.

References

1. Plagge, D., Leuschel, M.: Validating Z specifications using the ProB animator and model checker. In: 6th IFM, LNCS 4591. (2007) 480–500
2. Eshuis, R.: Translating safe Petri nets to statecharts in a structure-preserving way. In: 16th FM, LNCS 5850. (2009) 239–255
3. Vargas, P., et al.: Model Checking LTL Formulae in RAISE with FDR. In: 7th IFM, LNCS 5423. (2009) 245
4. Diaconescu, R., Futatsugi, K.: CafeOBJ Report. World Scientific (1998)
5. Clavel, M., Durán, F., et al.: All about Maude. LNCS 4350, Springer (2007)
6. Diaconescu, R., Futatsugi, K., Ogata, K.: CafeOBJ: Logical foundations and methodologies. Computing and Informatics **22** (2003) 257–283
7. Holzmann, G.: The SPIN model checker. Addison Wesley (2004)
8. Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method. Essays Dedicated to Jeseph A. Goguen, LNCS 4060 (2006) 596–615
9. Kong, W., Ogata, K., et al: A Lightweight Integration of Theorem Proving and Model Checking for System Verification. In: 12th APSEC. (2005) 59–66
10. Nakamura, M., Kong, W., et al: A Specification Translation from Behavioral Specifications to Rewrite Specifications. IEICE Transactions **91-D** (2008) 1492–1503
11. Zhang, M., Ogata, K.: Modular implementation of a translator from behavioral specifications to rewrite theory specifications. In: 9th QSIC. (2009) 406–411
12. Sipser, M.: Introduction to the Theory of Computation. PWS Pub. Co. (1996)
13. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. CACM **21** (1978) 993–999