Facilitating the Transformation of State Machines from Equations into Rewrite Rules

Min Zhang and Kazuhiro Ogata

School of Information Science
Japan Advanced Institute of Science and Technology (JAIST)
{zhangmin, ogata}@jaist.ac.jp

Abstract. The multiplicity of formalisms and corresponding verification systems makes the transformation useful for interoperations among them. We have proposed an approach to the transformation of state machines from a syntax-constrained class of equational theories into rewrite theories, which can be verified by Maude's model checking facilities. However, the efficiency of model checking generated rewrite theories varies, depending on the forms of original equational theories. This paper serves as a practical guide for developing equational theories to facilitate the transformation, aiming at generating efficiently model-checkable rewrite theories. Three case studies are conducted, and experimental results show that the efficiency is significantly improved.

1 Introduction

The multiplicity of formalisms and corresponding verification systems makes the transformations useful for the interoperations among formalisms and collaborations between verification systems. For instance, in the field of algebraic approach based formalizations and verifications, CafeOBJ uses equational theories for theorem proving [1], while Maude uses rewrite theories for model checking [2]. An approach called induction-guided falsification (IGF) has been proposed to achieve the combination of the two verifications [3]. To facilitate the combination, we have proposed an approach to the transformation from a syntax-constrained class of equational theories into rewrite theories [4]. Not only does the transformation save us duplicate effort of developing specifications, but it guarantees the consistency between two specifications of the same system.

Equational theories that are transformed by the approach are represented as OTSs (observational transition systems), and generated rewrite theories are represented as CTSs (component-based transition systems). Briefly, OTSs treat each system state as a set of observable values and each transition as a set of equations, while CTSs treat each state as a set of components and each transition as a rewrite rule. The transformation approach can generate efficiently model checkable CTSs, although it is only suitable for a syntax-constrained class of OTSs. We have shown that not all OTSs have corresponding rewrite theories in this approach [4]. To the best of our knowledge, most OTSs conform to (or can be tailored to) the syntax-constraints. However, for a state machine there

may exist two or more OTSs that represent it, and the efficiency of generated CTSs from them varies. The goal of this paper is to serve as a practical guide for developing OTSs that meet two requirements: (1) they can be transformed; and (2) generated CTSs are efficiently model-checkable. We conduct three non-trivial case studies including an authentication protocol NSPK (Needham-Schroeder Public Key), an electronic payment protocol *i*KP, and a distributed mutual exclusion protocol SKP (Suzuki-Kasami protocol). The case studies show that the CTSs generated from those OTSs that are tailored by following the guidance can be more efficiently model checked than the CTSs generated from those ones before being tailored, i.e. less time is needed to model check the CTSs generated from tailored OTSs, compared to those that are translated from unmodified ones.

The rest of this paper is organized as follow. Section 2 introduces the definitions of OTSs and CTSs. Section 3 describes the transformation approach, and Section 4 presents the tips for developing desirable OTSs. Section 5 describes the three case studies, and presents the experimental result. Section 6 talks about some related work and Section 7 concludes the paper.

2 Preliminaries

State machine is a mathematical concept for modeling computer systems. In algebraic approaches, a state machine can be specified as an equational theory, or as a rewrite theory [2, Chpt. 6]. OTS is proposed to describe state machines as equational theories [5], while CTS to describe state machines as rewrite theories.

2.1 Observational Transition Systems (OTSs)

We suppose that there exists a universal state space denoted by Υ and that each data type used in OTSs is provided. A data type is denoted by D with a subscript.

Definition 1 (OTSs). *An OTS $\mathcal{S}$ is a triple $\langle \mathcal{O}, \mathcal{I}, \mathcal{T} \rangle$ s.t.:*

- *$\mathcal{O}$: A set of observers. Each observer is a function $o : \Upsilon\ D_{o1} \ldots D_{om} \to D_o$. Two states υ_1, υ_2 are called equal (denoted by $\upsilon_1 =_{\mathcal{S}} \upsilon_2$) whenever each observer returns the same value with the same parameters from the two states.*
- *$\mathcal{I}$: A set of initial states s.t. $\mathcal{I} \subseteq \Upsilon$.*
- *$\mathcal{T}$: A set of transitions. Each transition is a function $t : \Upsilon\ D_{t1} \ldots D_{tn} \to \Upsilon$. Each transition t preserves the equivalence between two states in that whenever $\upsilon_1 =_{\mathcal{S}} \upsilon_2$, $t(\upsilon_1, y_1, \ldots, y_n) =_{\mathcal{S}} t(\upsilon_2, y_1, \ldots, y_n)$ for any other parameters $y_1, \ldots, y_n$. Each t has an effective condition $c\text{-}t : \Upsilon\ D_{t1} \ldots D_{tn} \to Bool$, s.t. for any state υ if $\neg c\text{-}t(\upsilon, y_1, \ldots, y_n)$, $t(\upsilon, y_1, \ldots, y_n) =_{\mathcal{S}} \upsilon$.*

In OTSs, equations are used to formalize initial states and the changes of observed values of each observer caused by transitions. Any state υ_0 in $\mathcal{I}$

must satisfy a set of equations, each of which corresponds to an observer $o : \Upsilon\ D_{o1} \dots D_{om} \to D_o$ as follows:

$$o(\upsilon_0, x_{o1}, \dots, x_{om}) = f_o(\upsilon_0, x_{o1}, \dots, x_{om}) \tag{1}$$

where $f_o(\upsilon_0, x_{o1}, \dots, x_{om})$ returns a value of D_o, and no transitions are allowed to occur in it. There is an equation declared for each pair of observer o and transition t in the following form:

$$o(t(\upsilon, y_{t1}, \dots, y_{tn}), x_{o1}, \dots, x_{om}) = f'_o(\upsilon, y_{t1}, \dots, y_{tn}, x_{o1}, \dots, x_{om}) \tag{2}$$

where $f'_o(\upsilon, y_{t1}, \dots, y_{tn}, x_{o1}, \dots, x_{om})$ returns a value of D_o, and transitions cannot occur in it. Equation 2 holds under the condition that $c\text{-}t(\upsilon, y_{t1}, \dots, y_{tn})$ is true, specifying how all the values observed by o are changed by the transition from the state represented by υ to the one by $t(\upsilon, y_{t1}, \dots, y_{tn})$. An equation is declared to define $c\text{-}t(\upsilon, y_{t1}, \dots, y_{tn})$.

2.2 Component-based Transition Systems (CTSs)

CTS is another notion of describing state machines as rewrite theories where transitions are represented as rewrite rules [6]. A state in CTSs is represented as a *configuration* which is essentially a set of components. This is inspired by the treatment of states based on an associative and commutative (AC) operation in rewriting logic [7]. There are two kinds of components in configurations, namely *observable components* and *transitional components*. Each observable component in a configuration c corresponds to a value in the state represented by c. All the observable components together in a configuration identify a state. Transitional components are used to provide transition information, including the names and parameters of transitions.

Transitions among states are formalized as rewrite rules. A rewrite rule is applied only in one direction from left to right. In CTS, a rewrite rule specifies how values are changed by the transformations from states to their successors. If the pattern in the lefthand side of a rule matches a fragment of a configuration c and the rule's condition is satisfied, the state transition specified by the rule takes place, and the matched fragment of c is transformed into the corresponding instance of the righthand side. Let c' be a configuration by applying a rule to c. We call c' a *successor configuration* of c.

Let $\mathbf{C}$ be a universal set of configurations, D_{oc} and D_{tc} be the data types for observable components and transitional components respectively.

Definition 2 (CTSs). *A CTS $\mathscr{S}$ is a four-tuple $\langle \mathcal{C}^o, \mathcal{C}^t, \mathbf{C}^0, \mathcal{R} \rangle$, where:*

- $\mathcal{C}^o$ *: a set of observable component constructors e.g.* $o[_, \dots, _]{:}_ : D_{o1} \dots D_{om}\ D_o \to D_{oc}$;
- $\mathcal{C}^t$ *: a set of transitional component constructors e.g.* $t : D_{t1} \dots D_{tn} \to D_{tc}$;
- $\mathbf{C}^0$ *: a set* $\mathbf{C}^0 \subseteq \mathbf{C}$ *of initial configurations;*
- $\mathcal{R}$ *: a set of rewrite rules.*

Observable component constructors are declared in mixfix style by convention. Underbars in constructors indicate where corresponding arguments go. A rewrite rule r in $\mathcal{R}$ is in the form of $L \Rightarrow R$ or $L \Rightarrow R$ *if* C, where L, R are configurations or segments of configurations, and C is a condition. A rewrite rule $L \Rightarrow R$ can be considered as a special case of $L \Rightarrow R$ *if* C with C being always true. Without loss of generality, r refers to $L \Rightarrow R$ *if* C by default in the paper.

3 Transformation from OTSs to CTSs

In this section, we describe the definition of the syntax-constrained OTSs, and the approach to the transformation from them into corresponding CTSs.

3.1 Syntax-constrained OTSs

We only allow two classes of observers in the syntax-constrained OTSs. An observer must be in the form of either $o : \Upsilon \rightarrow D_o$ or $\hat{o} : \Upsilon\ D_p \rightarrow D_{\hat{o}}$. D_p is usually a data type of processes, principals, or agent in systems to be formalized. o is called a system-level observer, and $\hat{o}$ a process-level observer. All process-level observers in a syntax-constrained OTS must have the same arity.

The equation declared for o or $\hat{o}$ w.r.t a transition $t : \Upsilon\ D_{t1} \ldots D_{tn} \rightarrow \Upsilon$ must be in one of the following three forms:

$$o(t(\upsilon, y_{t1}, \ldots, y_{tn})) = f_o(\upsilon, y_{t1}, \ldots, y_{tn}) \tag{3}$$

$$\hat{o}(t(\upsilon, y_{t1}, \ldots, y_{tn}), x_p) = \hat{o}(\upsilon, x_p) \tag{4}$$

$$\hat{o}(t(\upsilon, y_{t1}, \ldots, y_{ti-1}, x'_p, y_{ti+1}, \ldots, y_{tn}), x_p) = \begin{cases} f_{\hat{o}}(\upsilon, y_{t1}, \ldots, y_{ti-1}, x'_p, y_{ti+1}, \ldots, y_{tn}) & \text{if } x_p = x'_p \\ \hat{o}(\upsilon, x_p) & \text{otherwise} \end{cases} \tag{5}$$

Equation 3 says that the value observed by o is changed into $f_o(\upsilon, y_{t1}, \ldots, y_{tn})$ by the transition from υ into $t(\upsilon, y_{t1}, \ldots, y_{tn})$, and Equation 4 says no values observed by $\hat{o}$ are changed. When D_p is in t's arity, we assume $D_{ti}(1 \leq i \leq n)$ is D_p. Let x'_p be a variable of D_p. Equation 5 says that the value observed by $\hat{o}$ w.r.t. x_p is changed into $f_{\hat{o}}(\upsilon, y_{t1}, \ldots, y_{tn})$, and no other values are changed.

The above three equations guarantee that there is at most one value among the values observed by each observer that is changed by a transition, and make it decidable to compute which value may be changed. We can transform each OTS that conforms to the above constraints into a corresponding CTS.

3.2 The transformation

The transformation from the constrained class of OTSs into CTSs consists of two phases, i.e., *translation* and *optimization*.

Translation phase. The translation phase consists of three steps, i.e., (1) to translate observers and transitions into observable and transitional component constructors, (2) to generate initial configurations, and (3) to translate equations into rewrite rules.

1. Translation of observers and transitions. Observers and transitions are translated into corresponding observable and transitional component constructors. The correspondences between them are as follows:

$$o : \Upsilon \to D_o \Longrightarrow o{:}_ : D_o \to D_{oc}$$
$$\hat{o} : \Upsilon\ D_p \to D_{\hat{o}} \Longrightarrow \hat{o}{:}[_]_ : D_p\ D_{\hat{o}} \to D_{oc}$$
$$t : \Upsilon\ D_{t1} \ldots D_{tn} \to \Upsilon \Longrightarrow t : D^*_{t1} \ldots D^*_{tn} \to D_{tc}$$

D^*_{ti} denotes a data type of sets of elements which are of D_{ti}. By default, elements of D_{ti} are concatenated associatively and commutatively by an empty operator. Let y_{ti} be a variable of D_{ti}, and y^*_{ti} of D^*_{ti}. $(y_{ti}\ y^*_{ti})$ matches an arbitrary non-empty set ψ of elements of D_{ti}, with y_{ti} being instantiated by an arbitrary element e in ψ, and y^*_{ti} by $\psi - \{e\}$. This feature will be used in the generation of rewrite rules from equations.

The translations of observers and transitions are formalized by functions $h_o : \mathcal{O} \to \mathcal{C}^o$ and $h_t : \mathcal{T} \to \mathcal{C}^t$:

$$h_o(obs) \triangleq \begin{cases} o{:}_ : D_o \to D_{oc} & \text{if } obs \equiv o : \Upsilon \to D_o \\ \hat{o}[_]{:}_ : D_p\ D_{\hat{o}} \to D_{oc} & \text{if } obs \equiv \hat{o} : \Upsilon\ D_p \to D_{\hat{o}} \end{cases}$$
$$h_t(tran) \triangleq t : D^*_{t1} \ldots D^*_{tn} \to D_{tc} \quad \text{if } tran \equiv t : \Upsilon\ D_{t1} \ldots D_{tn} \to \Upsilon$$

We also write $h_o(o)$, $h_o(\hat{o})$ and $h_t(t)$ for convenience if no confusions are caused.

2. Generation of initial configurations. In OTSs, each initial state is identified by a set of all values returned by all observers. Such a value is represented by an observable component in CTSs. All the observable components together with a set of transitional components form an initial configuration. For a system-level observer o, its corresponding observable component is $(o : f_o(\upsilon_0))$, where $f_o(\upsilon_0)$ denotes the value observed by o in the initial state υ_0. For a process-level observer $\hat{o} : \Upsilon\ D_p \to D_{\hat{o}}$, there are $|D_p|$ values observed by $\hat{o}$. $|D_p|$ denotes the number of all the elements of D_p. We declare an auxiliary function $mk\text{-}\hat{o} : D^*_p \to \mathbf{C}$ s.t. $mk\text{-}\hat{o}(\emptyset) \triangleq \emptyset', mk\text{-}\hat{o}(p\ p^*) \triangleq (\hat{o}[p] : f_{\hat{o}}(\upsilon_0, p))\ (mk\text{-}\hat{o}(p^*))$. Function $mk\text{-}\hat{o}$ takes a set ψ_p of elements of D_p, and returns a set of observable components, each of which corresponds to an value observed by $\hat{o}$ w.r.t. an element $p \in \psi_p$, i.e., $f_{\hat{o}}(\upsilon_0, p)$. For a transition $t : \Upsilon\ D_{t1} \ldots D_{tn} \to \Upsilon$, the corresponding transitional component of t is $t(y^*_{t1}, \ldots, y^*_{tn})$ in initial configurations, where y^*_{ti} is a variable of D^*_{ti}, denoting a set of elements of D_{ti} for $i = 1, \ldots, n$.

We declare an initial configuration generator $init$, which takes a set of parameters and returns an initial configuration. Suppose that there are k_1 and k_2 observers of the first and second kinds, and k_3 transitions in a restricted OTS.

$$init : D^*_p\ D^*_1 \ldots D^*_l \to \mathbf{C} \text{ s.t.:}$$
$$init(p^*, y^*_1, \ldots, y^*_l) \triangleq (o_1{:}\ f_{o_1}(\upsilon_0))\ \ldots\ (o_{k_1}{:}\ f_{o_{k_1}}(\upsilon_0))\ mk\text{-}\hat{o}_1(p^*)\ \ldots\ mk\text{-}\hat{o}_{k_2}(p^*)$$
$$t_1(y^*_{t_1 1}, \ldots, y^*_{t_1 n_1})\ \ldots\ t_{k_3}(y^*_{t_{k_3} 1}, \ldots, y^*_{t_{k_3} n_{k_3}})$$

where, $\{D_1^*, \ldots, D_l^*\}$ is a set of $l (l \geq 0)$ data types s.t. each $D_j (j = 1, \ldots, l)$ must be in the arity of at least one transition in the original OTS and each D_{ti}^* in the arity of each transitional component constructor t is in the set. Each $n_i (i = 1 \ldots, k_3)$ is the number of parameters of transitional component constructor t_i, and each $y_{t_i j}^* (j = 1 \ldots, n_i)$ is a parameter in $\{y_1^*, \ldots, y_l^*\}$.

3. Transformation of equations into rewrite rules. We generate a rewrite rule for each t in $\mathcal{T}$ from all the equations that are declared for t. Table 1

Table 1. Representation of the changes of values in a transition t w.r.t. parameters $y_{t1}, \ldots, y_{tn}$ in OTS and CTS

transition/observer	OTS		CTS	
	state/ value	successor state/ value	component in configuration	component in successor configuration
$t : \Upsilon\ D_{t1} \ldots D_{tn} \to \Upsilon$	υ	$t(\upsilon, y_{t1}, \ldots, y_{tn})$	$t(y_{t1}\ y_{t1}^*, \ldots, y_{tn}\ y_{tn}^*)$	$t(y_{t1}\ y_{t1}^*, \ldots, y_{tn}\ y_{tn}^*)$
$o : \Upsilon \to D_o$	$o(\upsilon)$	$f_o(\upsilon, y_{t1}, \ldots, y_{tn})$	$(o : o(\upsilon))$	$(o : f_o(\upsilon, y_{t1}, \ldots, y_{tn}))$
$\hat{o} : \Upsilon\ D_p \to D_{\hat{o}}$	$\hat{o}(\upsilon, p)$	$\hat{o}(\upsilon, p)$	$(\hat{o}[p] : \hat{o}(\upsilon, p))$	$(\hat{o}[p] : \hat{o}(\upsilon, p))$
	$\hat{o}(\upsilon, y_{ti})$	$f_{\hat{o}}(\upsilon, y_{t1}, \ldots, y_{tn})$	$(\hat{o}[y_{ti}] : \hat{o}(\upsilon, y_{ti}))$	$(\hat{o}[y_{ti}] : f_{\hat{o}}(\upsilon, y_{t1}, \ldots, y_{tn}))$

shows how to represent the change of the values caused by a transition t w.r.t. parameters $y_{t1}, \ldots, y_{tn}$. In OTS, υ denotes a state, and $t(\upsilon, y_{t1}, \ldots, y_{tn})$ denotes the successor state of υ if $c\text{-}t(\upsilon, y_{t1}, \ldots, y_{tn})$. Since each y_{ti} is a variable of D_{ti}, y_{ti} can be instantiated by an arbitrary element of D_{ti}. The transition from the state represented by υ to the one represented by $t(\upsilon, y_{t1}, \ldots, y_{tn})$ specifies all possible state transitions caused by t. The changes of values are parameterized by υ, $y_{t1}, \ldots, y_{tn}$. In CTS, we use a transitional component $t(y_{t1}\ y_{t1}^*, \ldots, y_{tn}\ y_{tn}^*)$ to provide these parameters except υ. To ensure each y_{ti} can be instantiated by an arbitrary element of D_{ti}, a transitional component in a concrete configuration carries the set of all the elements of D_{ti}. Thus, whenever $(y_{ti}\ y_{ti}^*)$ matches these elements, y_{ti} can be instantiated by an arbitrary one in the set with the rest of all elements being assigned to y_{ti}^*, due to the associativity and commutativity of the empty concatenation operation.

In OTS, the value that is observed by o in υ is $o(\upsilon)$, which is changed into $f_o(\upsilon, y_{t1}, \ldots, y_{tn})$ in $t(\upsilon, y_{t1}, \ldots, y_{tn})$ according to Equation 3. This can be represented by the transformation of an observable component $(o : o(\upsilon))$ into the one $(o : f_o(\upsilon, y_{t1}, \ldots, y_{tn}))$. The values observed by $\hat{o}$ are not changed according to Equation 4, or only one value $\hat{o}(\upsilon, y_{ti})$ is changed into $f_{\hat{o}}(\upsilon, y_{t1}, \ldots, y_{tn})$ according to Equation 5. In CTS, we use an observable component $(\hat{o}[x_p] : \hat{o}(\upsilon, x_p))$ to represent the observed value $\hat{o}(\upsilon, x_p)$. The component will not be changed in the successor configuration, and is not necessarily specified in a rule. Similarly, we construct an observable component $(\hat{o}[y_{ti}] : \hat{o}(\upsilon, y_{ti}))$ to represent the observed value $\hat{o}(\upsilon, y_{ti})$. The component is transformed into $(\hat{o}[y_{ti}] : f_{\hat{o}}(\upsilon, y_{t1}, \ldots, y_{tn}))$ in the second case. Without loss of generality, we assume that the equation defined for each $\hat{o}$ conforms to Equation 5. We introduce new variables z_i of

$D_{o_i}(i = 1, \ldots, k_1)$ and z_{k_1+j} of $D_{\hat{o}_j}(j = 1, \ldots, k_2)$ to replace $o_i(\upsilon)$ and $\hat{o}_j(\upsilon, y_{ti_j})$ $(1 \leq i_j \leq n)$. We obtain the following rule:

$$
\begin{aligned}
&t(y_{t1}\ y^*_{t1}, \ldots, y_{tn}\ y^*_{tn})(o_1 : z_1) \ldots (o_{k_1} : z_{k_1})(\hat{o}_1[y_{ti_1}] : z_{k_1+1}) \ldots (\hat{o}_{k_2}[y_{ti_{k_2}}] : z_{k_1+k_2}) \\
&\Rightarrow t(y_{t1}\ y^*_{t1}, \ldots, y_{tn}\ y^*_{tn})\ (o_1 : f_{o_1}(\upsilon, y_{t1}, \ldots, y_{tn}))\ \ldots\ (o_{k_1} : f_{o_{k_1}}(\upsilon, y_{t1}, \ldots, y_{tn})) \\
&(\hat{o}_1[y_{ti_1}] : f_{\hat{o}_1}(\upsilon, y_{t1}, \ldots, y_{tn})) \ldots (\hat{o}_{k_2}[y_{ti_{k_2}}] : f_{\hat{o}_{k_2}}(\upsilon, y_{t1}, \ldots, y_{tn})) \\
&\textit{if } c\text{-}t(\upsilon, y_{t1}, \ldots, y_{tn})\ [z_i \mapsto o_i(\upsilon), z_{k_1+j} \mapsto \hat{o}_j(\upsilon, y_{ti_j})]. \qquad (6)
\end{aligned}
$$

It is worth mentioning that there may be some observed values that are not changed but only used in the change of other values. In this situation, we need to add their corresponding observable components to the both sides of the rule. We omit this case for simplicity since it is straightforward.

Optimization. The generated rewrite rule is not optimal because there are some redundant parameters which can be removed. These parameters may lead to the drastic increase the size of each configuration. For instance, the existence of a parameter $(y_{ti}\ y^*_{ti})$ in a transitional component indicates that each instance of a configuration must contain a transitional component that takes a set of all elements of D_{ti} as one of its parameters. This may lead to the drastic increase of the size of configurations when the number of the elements of some D_{ti} is huge. Thus, it is desirable to remove these redundant parameters from generated rules. We propose three optimization functions to achieve this.

1. Condition simplification g_c: For each generated rewrite rule r,

$$
g_c(r) \triangleq \begin{cases} L[x \mapsto T] \Rightarrow R[x \mapsto T] & \text{if } r \equiv (L \Rightarrow R \textit{ if } x = T) \\ L[x \mapsto T] \Rightarrow R[x \mapsto T] \textit{ if } C[x \mapsto T] & \text{if } r \equiv (L \Rightarrow R \textit{ if } x = T \wedge C) \\ r & \text{otherwise} \end{cases}
$$

   where, x is variable and T is an *M-pattern* term [1].
2. Variable elimination g_v: If $r \equiv t((y_{t1}\ y^*_{t1}), \ldots, (y_{ti}\ y^*_{ti}), \ldots, (y_{tn}\ y^*_{tn}))\ L \Rightarrow t((y_{t1}\ y^*_{t1}), \ldots, (y_{ti}\ y^*_{ti}), \ldots, (y_{tn}\ y^*_{tn}))\ R \textit{ if } C$, y_{ti} is in L, or not in R and C:

$$
\begin{aligned}
g_v(r) \triangleq\ & t((y_{t1}\ y^*_{t1}), \ldots, (y_{ti-1}\ y^*_{ti-1}), (y_{ti+1}\ y^*_{ti+1}), \ldots, (y_{tn}\ y^*_{tn}))L \Rightarrow \\
& t((y_{t1}\ y^*_{t1}), \ldots, (y_{ti-1}\ y^*_{ti-1}), (y_{ti+1}\ y^*_{ti+1}), \ldots, (y_{tn}\ y^*_{tn}))\ R \textit{ if } C.
\end{aligned}
$$

   Meanwhile, the arity of t is updated, i.e. $t : D^*_{t1} \ldots D^*_{ti} \ldots D^*_{tn} \rightarrow D_{tc}$ is changed into $t : D^*_{t1} \ldots D^*_{ti-1}\ D^*_{ti+1} \ldots D^*_{tn} \rightarrow D_{tc}$. Otherwise, $g_v(r) \triangleq r$. Computationally, y_{ti} in the transitional component makes sure that y_{ti} occurs at the lefthand side of the rule, making the rule executable. If y_{ti} occurs in L, y_{ti} in the transitional component is instantiated by the same value as the one assigned to y_{ti} in L when the rule is executed. If y_{ti} does not occur in R, C, the result after the rule is executed is not determined by the value of y_{ti}. Thus, removing y_{ti} does not change the executability of the rule.

[1] T is an *M-pattern* if for any well-formed substitution σ s.t. for each variable x in its domain the term $\sigma(x)$ is in canonical form w.r.t. the equations in M (a functional module that represents an equational theory), then $\sigma(T)$ is also in canonical form.

3. Component eliminations g_o: $g_o(r) \triangleq L \Rightarrow R$ *if* C if r satisfies one of the following conditions:
   (a) $r \equiv t\ L \Rightarrow t\ R$ *if* C;
   (b) $r \equiv (o : z_i)L \Rightarrow (o : z_i)R$ *if* C, z_i is not in R and C;
   (c) $r \equiv (\hat{o}[y_{ti}] : z_i)L \Rightarrow (\hat{o}[y_{ti}] : z_i)R$ *if* C, z_i is not in R and C, and y_{ti} is in L or not in R and C.

   The above three conditions make sure that a related component is neither changed nor used by other components in the rule. They are not necessarily included in rules. Removing them makes generated rule simpler, without changing the meaning of the rule.
   If r does not satisfy neither of the conditions, $g_o(r) = (r)$.

Condition simplification is usually first applied to a rule, making the rule satisfy the conditions of other two functions and hence can be further optimized. We apply the three optimization functions to the generated rewrite rules until no functions can be applied further, and obtain optimized rules. It can drastically reduce the size of each configuration by removing unnecessary arguments from transitional components, and hence makes generated CTSs more efficiently model-checkable.

4 Facilitating the Transformation

4.1 Developing syntax-constrained OTSs

It is sometimes not straightforward to develop a syntax-constrained OTS for a given system. One way is *to use as many system-level observers as possible* to make the OTSs conform to the constraints. It is straightforward since no restrictions are imposed to the equations declared for such observers, but nontrivial because it sometimes needs tricks to use them from case to case.

Example 1. Let us consider a system where there is an infinite list of natural numbers. Initially, all are 0. After the ith time tick, the first ith numbers are increased by one.

To model the system, we declare in an OTS $\mathcal{S}_{tick}$ two observers $step : \Upsilon \to N$ and $num : \Upsilon\ N \to N$. They observe the current step and each natural number in the list. We use transition $tick : \Upsilon \to \Upsilon$ to represent the tick action. We omit the definition of initiate states since it is straightforward. We define the following two equations for $tick$:

$$step(tick(\upsilon)) = step(\upsilon) + 1 \tag{7}$$

$$num(tick(\upsilon), x) = \begin{cases} num(\upsilon, x) + 1 & \text{if } x \leq step(\upsilon) \\ num(\upsilon, x) & \text{otherwise} \end{cases} \tag{8}$$

Obviously, $\mathcal{S}_{tick}$ does not conform to the restrictions and cannot be transformed. Actually, there does not exist such a CTS corresponding to $\mathcal{S}_{tick}$ if each observable component in the CTS corresponds to one natural number in the list.

That is because the natural numbers observed by *num* and changed by *tick* are unbounded. It needs unbounded number of rewrite rules to specify, which is impossible in a CTS.

An alternative way is to use two system-level observers *nonzeros* and *zeros* instead of *num*. The former observes the list of non-zero numbers, and the latter observes the list of all the zeros in the system. Let *NatList* be the data type of lists of natural numbers. Then, $nonzeros : \Upsilon \rightarrow NatList$, and $zeros : \Upsilon \rightarrow NatList$. Equations defined for them are as follows:

$$nonzeros(tick(\upsilon)) = (step(\upsilon) + 1) \mid nonzeros(\upsilon) \tag{9}$$

$$zeros(tick(\upsilon)) = zeros(\upsilon) \tag{10}$$

where, "$_|_ : N\ NatList \rightarrow NatList$" is a constructor of list of natural numbers. The two equations can be explained by Figure 1. Initially, the list of non-zeros

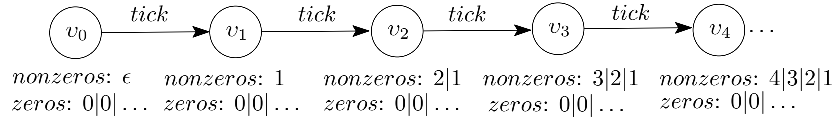

Fig. 1. Changes of the non-zero and zero lists by *tick*

is empty which is represented by $nonzeros(\upsilon_0) = \epsilon$ (ϵ denotes the empty list), and the other is an infinite list of zeros represented by $zeros(\upsilon_0) = 0|0|\ldots$. After each tick, the number of proceeding steps is appended to the head of the non-zero list, and the zero list is unchanged. Since all observers are system-level, the OTS conforms to the syntax constraints and can be transformed.

The syntax-constrained OTSs are well suitable for (but no limited to) specifying asynchronous distributed systems [4]. For other systems, it may need some tricks to use system-level observers instead of process-level observers depending concrete cases like Example 1.

4.2 Tailoring OTSs for model-checkable CTSs

As described in Section 3, the existence of a parameter ($y_{ti}\ y^*_{ti}$) in a transitional component may lead to the drastic increase of the size of corresponding configurations. Large size of configurations reduces the efficiency of model checking a CTS, because it is not only memory-consuming, but time-consuming to apply rewrite rules to the configurations. A worse case is that when the number of elements of D_{ti} is large or even infinite. It may make the size of CTSs too large to be fed into model checker. Although we can choose only a reasonably smaller subset to represent in components, this may lead to the loss of some cases during model checking, making the verification result less convincing. In order to make use of optimization functions to simply transitional components, we need

to make developed OTSs satisfy the conditions of optimization functions, especially g_c. That is because after g_c is applied, the simplified rewrite rule can satisfy the conditions of other two functions.

Let us consider the following rewrite rule:

$$t((y_{t1}\ y^*_{t1}), \ldots, (y_{ti}\ y^*_{ti}), \ldots, (y_{tn}\ y^*_{tn}))\ L \Rightarrow t((y_{t1}\ y^*_{t1}), \ldots, (y_{ti}\ y^*_{ti}), \ldots, (y_{tn}\ y^*_{tn}))\ R \text{ if } x = T \wedge C$$

We assume that y_{ti} does not occur in L, but in T, which is an *M-pattern* term. By g_c, x in T is substituted by T. Since, y_{ti} occurs in T, g_v can be applied to the rule, and the transitional component is simplified. Thus, a way of making generated CTSs maximally simplified by optimizations is to *specify effective conditions as conjunctions with conjuncts being equations.*

Example 2. Suppose in a system processes communicate with each other by exchanging messages. Once a message m is put into the network and sent to a process p, p fetches m from the network and puts it into its own message set.

Let P, M, M^* be the data types of processes, messages, and sets of messages respectively. We declare a system-level observer $nw : \Upsilon \to M^*$ for the set of messages in the network, a process-level observer $ms : \Upsilon\ P \to M^*$ for the set of messages owned by each process, and a transition $\mathit{fetch} : \Upsilon\ P\ M \to \Upsilon$ to model the action. The effective condition of *fetch* is represented by $\mathit{c\text{-}fetch} : \Upsilon\ P\ M \to \mathit{Bool}$ s.t.:

$$\mathit{c\text{-}fetch}(\upsilon, p, m) = \mathit{in?}(m, nw(\upsilon)) \wedge (\mathit{dst}(m) = p)$$

where, p and m are variables of P and M respectively, $\mathit{dst}(m)$ denotes the destination of m, and $\mathit{in?}$ is a predicate denoting if a message m is in the network. Then, the following two equations specify that m is deleted from the network in υ if $\mathit{c\text{-}fetch}(\upsilon, p, m)$ is true.

$$nw(\mathit{fetch}(\upsilon, p, m)) = \mathit{del}(m, nw(\upsilon))$$

$$ms(\mathit{fetch}(\upsilon, p, m), p') = \begin{cases} \mathit{add}(m, ms(\upsilon, p)) & \text{if } p = p' \\ ms(\upsilon, p') & \text{otherwise} \end{cases}$$

where, p' is a variable of P, and *del* (*add*) is a function of deleting (adding) a message from (to) a set of messages.

The equations are transformed into the following rewrite rule according to the approach:

$$\mathit{fetch}((m\ m^*))(nw\text{: } m^*_1)(ms[p]\text{: } m^*_2) \Rightarrow \mathit{fetch}((m\ m^*))(nw\text{: } \mathit{del}(m, m^*_1))$$
$$(ms[p]\text{: } \mathit{add}(m, m^*_2)) \text{ if } \mathit{in?}(m, m^*_1) \wedge \mathit{dst}(m) = p \qquad (11)$$

where m^*, m^*_1, m^*_2 are variables of M^*, and $\mathit{fetch} : M^* \to D_{tc}$. The rule cannot be simplified further by the proposed optimization functions. Each configuration must contain a transitional component w.r.t. *fetch*, with a set of all messages in it. This may lead to the huge size of the configurations, depending upon the

number of messages used in the network. In practice, it is usually large, or even infinite, which makes the generated CTS even cannot be fed into model checker, as demonstrated in Section 5.

An alternative is to specify the conjunct $in?(m, nw(\upsilon))$ in the effective condition of *fetch* as an equation. We modify the arity of *c-fetch*, and obtain a new one $c\text{-}fetch' : \Upsilon\ P\ M\ M^* \rightarrow \Upsilon$.

$$c\text{-}fetch'(\upsilon, p, m, m^*) = (nw(\upsilon) = (m\ m^*)) \wedge (dst(m) = p)$$

The conjunct $nw(\upsilon) = (m\ m^*)$ also denotes that m is in the network. Consequently, the transition *fetch* needs to be modified to be $fetch' : \Upsilon\ P\ M\ M^* \rightarrow \Upsilon$.

An intermediate rule generated from the equation is in the form of:

$$fetch'((m\ m_1^*), (m^*\ m^{**}))(nw\colon m_2^*)(ms[p]\colon m_3^*) \Rightarrow fetch'((m\ m_1^*), (m^*\ m^{**}))$$
$$(nw\colon del(m, m_2^*))(ms[p]\colon add(m, m_3^*)) \text{ if } (m_2^* = (m\ m^*)) \wedge dst(m) = p \qquad (12)$$

where, $send' : M^*\ M^{**} \rightarrow D_{tc}$, and m^{**} is a variable of M^{**}. M^{**} is a data type of sets of elements of M^*. We can apply g_c, g_v, and g_o to the rule 12, and obtain an optimized one:

$$(nw\colon (m\ m^*))(ms[p]\colon m_3^*) \Rightarrow (nw\colon m^*)(ms[p]\colon add(m, m_3^*)) \text{ if } dst(m) = p$$

Transitional component w.r.t. *fetch'* is removed from configurations, which reduces the size of each configuration. Moreover, we do not need to enumerate all messages to initialize the transitional component, or choose a subset of messages if the number of messages is large or infinite. It makes the generated CTS more efficiently model checkable.

5 Case studies

We conduct three case studies including NSPK, *i*KP, and SKP, which have been modeled as OTSs and verified in CafeOBJ [3,8,9]. Although the OTSs can be transformed into CTSs, neither of the CTSs can be efficiently model checked due to the large size of configurations. Following the guidance in Section 4, we tailor these old OTSs and transform the tailored OTSs into CTSs, model check the generated CTSs and record the time spent on model checking. Experimental results show that CTSs that are generated from tailored OTSs need less time to be model checked. The experimental environment is Maude 2.6, running on Ubuntu 10.04 on a laptop with a 1.20GHz dual-core processor and 4GB memory.

NSPK is a security protocol to achieve the mutual authentication between two principals over a network [10]. Each principal holds a pair of public and private keys. A connection between two principals is established by three message exchanges. Firstly, a principal p encrypts the tuple of a nonce n_p and p's identifier with a principal q's public key, and sends the ciphertext $\mathcal{E}_q(n_p, p)$ to q. q decrypts the ciphertext with its private key and obtains n_p and p's identifier. Then, q

generates a nonce n_q, encrypts the tuple of n_q and n_p with p's public key, and sends the ciphertext $\mathcal{E}_p(n_q, n_p)$ to p. p decrypts the ciphertext with its private key and obtains n_q, which convinces p that it is communicating with q. Finally, p encrypts n_q with q's public key and sends the ciphertext $\mathcal{E}_q(n_q)$ to q. q decrypts the ciphertext with its private key, and obtains n_q, which convinces q that it is communicating with p.

In [3], NSPK is formalized as an OTS $\mathcal{S}_{\rm NSPK}$. There are three observers *rand*, *nw* and *nonces* in $\mathcal{S}_{\rm NSPK}$ respectively for random numbers, messages that are put in the network, and nonces collected by intruders. Without the loss of generality, we only consider in $\mathcal{S}_{\rm NSPK}$ the formalization of the action that q sends $\mathcal{E}_p(n_q, n_p)$ to p as an example. Let P and N are data types of principals and nonces. The action is specified by a transition $send_2 : \Upsilon\ P\ P\ P\ N \rightarrow \Upsilon$. Its effective condition is denoted by $c\text{-}send_2 : \Upsilon\ P\ P\ P\ N \rightarrow Bool$ s.t.:

$$c\text{-}send_2(\upsilon, q, p, p', n) = p \neq q \wedge in?(m(p', p, q, enc_1(p, n, q)), nw(\upsilon))$$

It says that transition from υ to $send_2(\upsilon, q, p, p', n)$ takes place whenever $q \neq p$ and the message $m(p', p, q, enc_1(p, n, q))$ is in the network. $m(p', q, p, enc_1(p, n, q))$ denotes the message that is sent by p' from p to q with a ciphertext $\mathcal{E}_q(n, p)$ that is specified as $enc_1(p, n, q)$. The effective condition can also be specified by $c\text{-}send'_2 : \Upsilon\ P\ P\ P\ N\ M^* \rightarrow Bool$ (M^* is the data type of sets of messages) s.t.

$$c\text{-}send'_2(\upsilon, q, p, p', n, ms) = p \neq q \wedge nw(\upsilon) = (m(p', p, q, enc_1(p, n, q))\ ms) \quad (13)$$

We modify the representations of other transitions' effective conditions in $\mathscr{S}_{\rm NSPK}$, and obtain a new one denoted by $\mathscr{S}'_{\rm NSPK}$. We transform $\mathcal{S}_{\rm NSPK}$ and $\mathcal{S}'_{\rm NSPK}$ into their corresponding CTSs denoted by $\mathscr{S}_{\rm NSPK}$ and $\mathscr{S}'_{\rm NSPK}$.

We assume that there are three principals (including two normal ones and an intruder), two random numbers used the protocol. This leads to 18 nonces and 32706 messages. $\mathscr{S}_{\rm NSPK}$ is hardly model checked because the transitional component corresponding to $send_2$ takes 32706 messages as one of its arguments, which makes the size of $\mathscr{S}_{\rm NSPK}$ to large to be fed into Maude model checker. In $\mathscr{S}'_{\rm NSPK}$, transitional components are removed or simplified by optimization based on the equations like 13, making the size of each configuration reasonably small. We model check $\mathscr{S}'_{\rm NSPK}$ in Maude against the secrecy property, which says that nonces generated by normal principals cannot be gleaned by intruders. Maude returns a counterexample in 181.2 seconds. This result coincides with the claim that NSPK does not satisfy secrecy property [3].

***i*KP** is an electronic payment protocol to achieve authenticated payment among three parties: acquirer, buyers and sellers [11]. The protocol is based on public key cryptography. Each acquirer A has a secret and public key pair. Only the public key is known by both sellers and buyers. Each seller S in 2KP/3KP and each buyer B in 3KP has a secret and public key pair. There are six message exchanges among a buyer B, a seller S and an acquire A to complete a transaction.

*i*KP has been modeled as an OTS $\mathcal{S}_{i\rm KP}$ which consists of 11 observers and 38 transitions. We only consider one transition *sdvm* as an example. The transition

specifies an action that a seller s sends an *invoice* message to a buyer b once s receives an *initiate* message generated by b. An *initiate* message is represented by $im(b', b, s, hbn)$ where b' is the sender of the message, b and s are the buyer and seller involved in the transaction, and hbn is the keyed hashed BAN used in the transaction between b and s. The effective condition of $sdvm$ is specified by the following equation:

$$c\text{-}sdvm(\upsilon, b', b, s, pr) = in?(im(b', b, s, hbn), nw(\upsilon))$$

which says that an *initiate* message must be in the network. pr denotes a price which is a piece of information of the *invoice* message to be sent to b. We tailor the effective condition as follows:

$$c\text{-}sdvm'(\upsilon, b', b, s, pr, ms) = (nw(\upsilon) = (im(b', b, s, hbn)\ ms)) \tag{14}$$

where, ms is a variable denoting a set of messages. It is similar to tailor the effective conditions of other 37 transitions. We obtain a new OTS $\mathcal{S}'_{i\mathrm{KP}}$.

We transform $\mathcal{S}_{i\mathrm{KP}}$ and $\mathcal{S}'_{i\mathrm{KP}}$ into $\mathscr{S}_{i\mathrm{KP}}$ and $\mathscr{S}'_{i\mathrm{KP}}$ respectively, and model check them against the *payment agreement property.* In $\mathscr{S}_{i\mathrm{KP}}$, each configuration contains a set of messages that are to be used in message exchanges. We suppose there exists one trustable buyer, one trustable seller, one intruder buyer and one intruder seller. The number of all messages is more than 10 billion, and hence impossible to include in a component. Even if we had all the messages in the component, it would drastically increase the size of each configuration, making $\mathscr{S}_{i\mathrm{KP}}$ impossible be model checked.

However, in $\mathscr{S}'_{i\mathrm{KP}}$ we only provide the number of buyers and sellers for each configuration, because messages in transitional components are removed by optimizations based on the equations like 14. We use two buyers and two sellers in $\mathscr{S}'_{i\mathrm{KP}}$, and model check it in Maude against the *payment agreement property.* The property says that whenever an acquire authorizes a payment, both the buyer and seller involved in the payment must have agreed on it. Maude returns a counterexample in 1.196 seconds. The counterexample coincides with the reality that iKP protocols do not satisfy the property [8].

SKP is used to solve the distributed mutual exclusion problem for a computer network where there are a fixed number $N(\geq 1)$ of nodes, and the nodes have no memory in common and can communicate only by exchanging messages. The communication delay is completely unpredictable, namely that although messages eventually arrive at their destinations, they are not guaranteed to be delivered in the same order as they are sent. The mutual exclusion means that at most one node is allowed to stay in its critical section at any moment. The basic idea is to transfer the privilege among nodes for entering critical sections. We omit the details of the algorithm. One can refer to [12] for details.

SKP has been modeled as an OTS $\mathcal{S}_{\mathrm{SKP}}$, and theorem proved to enjoy the mutual exclusion property in CafeOBJ [9]. In $\mathcal{S}_{\mathrm{SKP}}$, a transition $receiveReq$: $\Upsilon\ P\ R \rightarrow \Upsilon$ is declared to specify the action that a process p receives a request

from other processes for the privilege of entering the critical section. P and R are data types of processes and requests. The effective condition of the transition is $c\text{-}receiveReq : \Upsilon\ P\ R \to Bool$ s.t.:

$$c\text{-}receiveReq(\upsilon, p, r) = in?(rm(p, r), nw(\upsilon)) \wedge (fst(r) \neq p)$$

The equation says that a request message denoted by $rm(p, r)$ is in the network and the request r is not sent by p.

We tailor the representation of the effective condition according to the second case in Section 4.2. Let $c\text{-}receiveReq' : \Upsilon\ P\ R\ M^* \to Bool$, s.t.:

$$c\text{-}receiveReq'(\upsilon, p, r, m^*) = (nw(\upsilon) = rm(p, r)\ m^*) \wedge (fst(r) \neq p) \tag{15}$$

We obtain a new OTS denoted by $\mathcal{S}'_{\text{SKP}}$.

We transform $\mathcal{S}_{\text{SKP}}$ and $\mathcal{S}'_{\text{SKP}}$ into corresponding CTSs $\mathscr{S}_{\text{SKP}}$ and $\mathscr{S}'_{\text{SKP}}$, and model check them by Maude respectively. Table 2 shows the time that is taken to model check the mutual exclusion property with different number of processes from 2 to 7. It is obvious that $\mathscr{S}'_{\text{SKP}}$ needs less time than $\mathscr{S}_{\text{SKP}}$. That is because

Table 2. Time (s) spent on model checking $\mathscr{S}_{\text{SKP}}$ and $\mathscr{S}'_{\text{SKP}}$ with the increase of the number of processes from 2 to 7.

	2	3	4	5	6	7
$\mathscr{S}_{\text{SKP}}$	0.008	0.128	2.252	41.514	1272.467	–
$\mathscr{S}'_{\text{SKP}}$	0.004	0.032	0.260	2.916	38.650	4095.575

each configuration in $\mathscr{S}_{\text{SKP}}$ contains a transitional component which consists of a set of n^2 requests between n processes. With the increase of the number of processes in the protocol, the size of each configuration increases, leading to the drop in the efficiency of model checking $\mathscr{S}_{\text{SKP}}$. The transitional component is removed by optimizations in $\mathscr{S}'_{\text{SKP}}$ because of the equation 15. The efficiency is only affected by the increase of processes. The time increases with the growth of the number of processes mostly due to the increase of the number of states.

In sum, the three case studies show that by following the guidance described in Section 4 we can develop desirable syntax-constrained OTSs from which generated CTSs are efficiently model-checkable.

6 Related work

To the best of our knowledge, there are at lest three other related approaches proposed to the translation of state machines from equations into rewrite rules [13,14,15]. The purpose of these approaches including ours is the same, i.e., to generate rewrite theory for model checking. However, none of the three approaches concern with the efficiency issue. For some OTSs of protocols such as

NSPK, iKP, and SKP, no result can be returned by model checking the generated rewrite theories in reasonable time (e.g. a few of days). Even worse, generated rewrite theory may be to huge too be fed into Maude model checkers.

Most algebraic languages endow users with freedom to choose their own style to formalize systems. However, the lack of guidance may cause their formalisms to be less readable, at higher risk of having errors, or of lower efficiency. It is necessary to have some systematic methods or guidance for user to formalize systems uniformly. This can bring multiple benefits, e.g., the improvement of readability, ease or efficiency in verification or transformation, etc. Several related methods and advises in this field have been proposed. The OTS/CafeOBJ modeling method is another example [5]. It guides users for formalizing systems as a subclass of behavioral specifications, which can be systematically theorem proved by proof score approach in CafeOBJ. Although the purposes of these methods are different, they share some commons i.e., making systems more uniformly formalized in certain approaches or by syntactical constraints. This may lead to less flexibility of formalisms, but we believe that this sacrifice is worthwhile with little or no loss of the expressiveness.

7 Conclusion and Future Work

We have presented some guidance for developing desirable equational theories, which are represented as a syntax-constrained class of OTSs. Such OTSs can be transformed into a class of rewrite theories that are represented by CTSs. We conducted three non-trivial case studies to show that the rewrite theories that are generated from tailored OTSs are more efficiently model-checkable than those generated from those OTSs before being tailored.

The three cases have been well formally analyzed. We used them as benchmarks to evaluate how much the effectiveness of generated CTSs can be improved in model checking. We will try to formalize new systems as OTSs by following the guidance, transforming them into CTSs, and using the IGF approach to verify the systems with both OTSs and CTSs. One candidate system is OSEK/VDX, a set of standards in the automotive industry for a distributed, real-time architecture for control unit in vehicles [16]. We choose OSEK/VDX as an application for three reasons: (1) it is a safety-critical system, and some important properties must be guaranteed e.g., deadlock freedom; (2) it is a really industrial specification and has not been sufficiently studied from a formal verification point of view; and (3) it is not a typical asynchronous distributed system, and the guide suites to specify the system as a syntax-constrained OTS.

References

1. Diaconescu, R., Futatsugi, K.: CafeOBJ Report. World Scientific (1998)
2. Clavel, M., Durán, F., et al.: All about Maude. LNCS 4350, Springer (2007)
3. Ogata, K., Nakano, M., Kong, W., Futatsugi, K.: Induction-Guided Falsification. In: 8th ICFEM, LNCS 4260, Springer (2006) 114–131

4. Zhang, M., Ogata, K., Nakamura, M.: Specification Translation of State Machines from Equational Theories into Rewrite Theories. In: 12th ICFEM, LNCS 6447, Springer (2010) 678–693
5. Ogata, K., Futatsugi, K.: Some tips on writing proof scores in the OTS/CafeOBJ method. Essays Dedicated to Joseph A. Goguen, LNCS 4060 (2006) 596–615
6. Bruni, R., Meseguer, J.: Generalized Rewrite Theories. ICALP'03, LNCS **2719** (2003) 252–266
7. Meseguer, J.: Conditional Rewriting Logic as a Unified Model of Concurrency. TCS **96** (1992) 73–155
8. Ogata, K., Futatsugi, K.: Formal Analysis of *i*KP Electronic Payment Protocols. In: ISSS'02, LNCS 2609, Springer (2002) 441–460
9. Ogata, K., Futatsugi, K.: Formal Analysis of Suzuki & Kasami Distributed Mutual Exclusion Algorithm. In: FMOODS'02, LNCS, Springer (2002) 181–195
10. Needham, R., Schroeder, M.: Using encryption for authentication in large networks of computers. CACM **21** (1978) 993–999
11. Bellare, M., Garay, J., et al.: Design, implementation, and deployment of the *i*KP secure electronic payment system. IEEE Journal of Selected Areas in Communications **18** (2000) 611–627
12. Suzuki, I., Kasami, T.: A distributed mutual exclusion algorithm. ACM Trans. Comput. Syst. **3** (1985) 344–349
13. Futatsugi, K.: Fostering Proof Scores in CafeOBJ. In: 12th ICFEM, LNCS 6447, Springer (2010) 1–20
14. Nakamura, M., Kong, W., et al: A Specification Translation from Behavioral Specifications to Rewrite Specifications. IEICE Transactions **91-D** (2008) 1492–1503
15. Kong, W., Ogata, K., et al: A Lightweight Integration of Theorem Proving and Model Checking for System Verification. In: 12th APSEC. (2005) 59–66
16. Lemieux, J.: Programming in the OSEK/VDX Environment. Cmp (2001)