Automatic Verification of a Model Checker by Reflection

Bow-Yaw Wang *

Institute of Information Science
Academia Sinica
128 Sec 2 Academia Rd
Taipei 115, Taiwan
TEL: +886-2-2788-3799x1717 FAX: +886-2-2782-4814
bywang@iis.sinica.edu.tw

Abstract. Intuitively, reflection is the feature that can represent and reason meta-level entities at the object level. In this paper, we use a reflective language to implement a local model checker and analyze the implementation. The implementation is greatly simplified by reflection. Further, we show the feature can be applied to verify the concise implementation rather easily. The simplicity of our approach suggests that reflection may be useful in the implementation and verification of other explicit-state model checking algorithms.
Key words: Reflection, Rewriting Logic, Model Checking, Logic Programming

1 Introduction

Model checking has become a popular technique to improve system quality during the past decade. Thanks to its success in hardware verification, many model checkers are being developed in research laboratories and sold by companies. But building a model checker requires sophisticated programming and algorithm-developing skills. A typical model checker may contain tens, even hundreds, of thousands of lines of C code. Since model checkers have been deployed in the design of many critical systems, one wonders whether there is a way to ensure the quality of these verification tools.

In this paper, we use rewriting logic [15] as the formalism to verify a working model checker. Following the framework proposed in [23, 10, 21], we implement a model checker in Maude, a logic programming language based on rewriting logic [6]. Unlike other model checkers which use different languages in their implementation and model specification, the Maude language is also used as the modeling language of our model checker.

The key to use Maude as the algorithm implementation *and* the model specification language is reflection. Intuitively, reflection is the feature that can represent and reason about meta-level entities at the object level. In the framework

* This research was partly supported by NSC 94-2213-E-001 -003 -

of [23, 10, 21], model specifications reside in the meta level. The model checking algorithm inspects meta-level specifications by reflection. Hence, we can implement a model checking algorithm in Maude. It is unnecessary to have different languages in different levels. Additionally model simulation can be performed by reflection. This simplifies our implementation in Maude significantly.

Furthermore, we can verify our implementation by another application of reflection. While verifying our model checker, the implementation becomes an entity in the meta level. We are able to use other object-level model checkers to analyze our implementation. Specifically, we verify our model checker by two different model checkers — the abstract model checking algorithm and the Maude built-in LTL model checker in [10].

The advantages of our approach are its simplicity and clarity. With the reflective language Maude, the model checking paradigm is modeled as two levels of computation. Using the same principle, it is straightforward to model the verification of model checkers as another level of computation. We feel the same task would be too complicated to achieve had the concept of reflection not been introduced in the framework. Reflection in declarative languages is not only of theoretical endeavor, but also of practical interests.

1.1 Related Work

Model checking algorithms have been formally verified by proof assistants [19, 13]. In these work, the semantics and algorithms are formalized in the meta logic of proof assistants. Verifying model checking algorithms amounts to proving that the outcomes of algorithms agree with the semantics in the meta logic. In principle, it is possible to verify systems that can be formalized in the meta logic. But intensive human intervention is required.

An LTL model checker is available in recent releases of Maude [10]. The performance of the built-in LTL model checker is comparable to the model checker Spin [11]. But the implementation is written in C++. It is difficult for verification tool developers to modify and improve the internal model checker.

The inconvenience is resolved in [21] where a proof-theoretic μ-calculus model checking algorithm [9, 20, 24] is presented. The μ-calculus model checking algorithm is implemented in an older version of Maude, and requires extension to core Maude system for technical reasons. Subsequently, it is less efficient than what we present in this paper.

In [2, 7], reflection is used for reasoning families of membership equational theories. Metatheorems about families of theories are represented and proved as theorems at object level by reflection. The idea is realized in the theorem prover ITP for membership equational theories [5].

1.2 Outline

The paper is organized as follows. Section 2 provides necessary technical backgrounds. An abstract μ-calculus model checking algorithm is presented in Section 3. It is followed by its concrete implementation in Section 4. We use the

concrete implementation in Section 4 to verify properties of Peterson's algorithm in Section 5. The μ-calculus model checker is then verified by two different algorithms in Section 6. Finally, we discuss future work and conclude the paper in Section 7.

2 Preliminaries

We briefly review μ-calculus model checking and rewriting logic. For a more detailed exposition, the reader is referred to [12, 9, 20, 24, 8, 4, 10].

2.1 μ-Calculus

A μ-calculus formula φ is constructed by the following rules [12]:

- propositional variables: $X, Y, Z, \ldots$;
- atomic propositions (AP): $p, q, r, \ldots$;
- Boolean operators: $\neg\varphi$, $\varphi \vee \varphi'$;
- modal existential next-state operator: $\langle\bar{\ell}\rangle\varphi$, where $\bar{\ell}$ is a set of transition labels;
- greatest fixed-point operator: $\nu X.\varphi$, where the bound variable X occurs positively in φ.

As usual, we use derived operators such as $\varphi \wedge \varphi' (\equiv \neg(\neg\varphi \vee \neg\varphi'))$, $[\bar{\ell}]\varphi (\equiv \neg\langle\bar{\ell}\rangle\neg\varphi)$ and $\mu X.\varphi (\equiv \neg\nu X.\neg\varphi[\neg X/X])$.

The semantics of μ-calculus formulae is defined over a *Kripke structure* $K = (S, Labl, \rightarrow, s_0, P)$ where S is the set of states, $Labl$ the set of transition labels, $\rightarrow \subseteq S \times Labl \times S$ the transition relation, $s_0 \in S$ the initial state, and $P \in S \rightarrow 2^{AP}$ the labeling function which maps each state to a set of atomic propositions satisfied in the state. For clarity, we write $s \xrightarrow{a} t$ for $(s, a, t) \in \rightarrow$. A *valuation* ρ is a function mapping propositional variables to subsets of S. Let $R \subseteq S$. We write $\rho[X \mapsto R]$ for the valuation mapping X to R and Y to $\rho(Y)$ for $X \neq Y$. Given the valuation ρ, the semantic function $[\![\varphi]\!]\rho$ for a μ-calculus formula φ computes the set of states satisfying φ under the valuation ρ:

- $[\![X]\!]\rho = \rho(X)$;
- $[\![p]\!]\rho = \{s \in S : p \in P(s)\}$;
- $[\![\neg\varphi]\!]\rho = S \setminus [\![\varphi]\!]\rho$;
- $[\![\varphi \vee \varphi']\!]\rho = [\![\varphi]\!]\rho \cup [\![\varphi']\!]\rho$;
- $[\![\langle\bar{\ell}\rangle\varphi]\!]\rho = \{s \in S : \exists a \in \{\bar{\ell}\}, t \in S. s \xrightarrow{a} t \text{ and } t \in [\![\varphi]\!]\rho\}$;
- $[\![\nu X.\varphi]\!]\rho = \bigcup\{R \subseteq S : R \subseteq [\![\varphi]\!](\rho[X \mapsto R])\}$.

For any μ-calculus formula φ and Kripke structure $K = (S, L, \rightarrow, s_0, P)$, we write $K, s \models \varphi$ when $s \in [\![\varphi]\!]\emptyset$. The *$\mu$-calculus model checking problem* is to determine whether $K, s_0 \models \varphi$.

In order to solve the μ-calculus model checking problem, various algorithms have been developed (see, for example, [3]). In tableau-based local model checking algorithms [9, 20], the problem is solved by constructing proofs of the judgment $K, s \vdash \varphi$. The tableau-based algorithms were then simplified to a set of reduction rules in [24]. The following extension to the greatest fixed point operator, $\nu X\{\bar{r}\}\varphi$ where $\bar{r}$ is a set of states, is introduced in [24]:

$$[\![\nu X\{\bar{r}\}\varphi]\!]\rho = \bigcup \{R \subseteq S : R \subseteq \{\bar{r}\} \cup [\![\varphi]\!](\rho[X \mapsto R])\}\,.$$

Note that $\nu X\{\}\varphi \equiv \nu X.\varphi$. Any fixed-point operator can be translated to its extended form syntactically. Intuitively, the formula $\nu X\{\bar{r}\}\varphi$ records previously visited states in $\{\bar{r}\}$, which is handy for co-inductive proofs. The extension reduces the side condition of tableau-based algorithms to membership checking and allows the proof search to be performed by rewriting. Given a Kripke structure $K = (S, Labl, \rightarrow, s_0, P)$ and a μ-calculus formula φ, the following rules reduce $K, s \vdash \varphi$ to Boolean values true or false [24]:

- $(K, s \vdash p) = \mathsf{true}$ if $p \in P(s)$;
- $(K, s \vdash p) = \mathsf{false}$ if $p \notin P(s)$;
- $(K, s \vdash \mathsf{false}) = \mathsf{false}$;
- $(K, s \vdash \neg\varphi) = \neg b$ where $(K, s \vdash \varphi) = b$;
- $(K, s \vdash \varphi \vee \varphi') = b_0 \vee b_1$ where $(K, s \vdash \varphi) = b_0$ and $(K, s \vdash \varphi') = b_1$;
- $(K, s \vdash \langle\bar{\ell}\rangle\varphi) = \mathsf{true}$ if $(K, t \vdash \varphi) = \mathsf{true}$ for some t and a such that $a \in \{\bar{\ell}\}$ and $s \xrightarrow{a} t$;
- $(K, s \vdash \nu X\{\bar{r}\}\varphi) = \mathsf{true}$ if $s \in \{\bar{r}\}$;
- $(K, s \vdash \nu X\{\bar{r}\}\varphi) = (K, s \vdash \varphi[\nu X\{s, \bar{r}\}\varphi/X])$ if $s \notin \{\bar{r}\}$.

Let K be a finite Kripke structure and φ a μ-calculus formula. It is shown that $(K, s \vdash \varphi) = \mathsf{true}$ if and only if $K, s \models \varphi$ [24].

2.2 Rewriting Logic

Since its introduction in [15], rewriting logic has been used as a unified formalism for modeling concurrency [15, 16, 14] and as a logical framework [1]. It is not hard to see that rewriting logic is capable of property and model specification [23, 10, 21]. In the following, we will briefly review rewriting logic and its verification framework as proposed in [10].

In rewriting logic, a *term* is constructed by function and constant symbols. Each term belongs to one or several *sorts*. *Equations* specify equivalent terms. *Rewriting rules* specify how to transform a term into another. A rewrite theory consists of equations and rewriting rules for terms. If a rewrite theory does not contain any rewriting rules, we also call it an *equational theory*.

We follow the syntax of Maude in our presentation. Maude is a term rewriting system based on rewriting logic. In Maude, function and constant symbols are declared by the keyword **op**. Sorts are declared by the keyword **sort**. Equations are specified by **eq** *lhs* = *rhs*; conditional equations are specified by **ceq** *lhs* =

rhs **if** *cond*. Similarly, rewriting rules and conditional rewriting rules are defined by **rl** $[l] : lhs \Rightarrow rhs$ and **crl** $[l] : lhs \Rightarrow rhs$ **if** *cond* respectively, where l is the label of the rule. The left-hand side of equations and rewriting rules allows pattern matching. Since there may be several ways to match a term, applying a rewriting rule to a given term may yield multiple results. All results obtained by any of these applications are admissible in rewriting logic.

For any term t, we write $[t]$ for its equivalence class defined by the equations in a rewrite theory. Let $\mathcal{R}$ be a rewrite theory and t, t' two terms in $\mathcal{R}$. We write

$$\mathcal{R} \vdash_l [t] \rightarrow [t']$$

if there is a rule labeled l in $\mathcal{R}$ that rewrites t to t'.

In rewriting logic, there is a universal theory $\mathcal{U}$ such that any rewrite theory $\mathcal{R}$ and a term t can be represented as meta-level terms $\underline{\mathcal{R}}$ and $\underline{t}$ in $\mathcal{U}$ respectively. Furthermore, we have

$$\mathcal{R} \vdash_l [t] \rightarrow [t'] \Leftrightarrow \mathcal{U} \vdash_{l,n} [\underline{\mathcal{R}}, \underline{t}] \rightarrow [\underline{\mathcal{R}}, \underline{t'}]$$

if t' is the n-th result obtained by applying the rewriting rule labeled l to t in $\mathcal{R}$. By the universal theory $\mathcal{U}$, we can manipulate meta-level terms at the object level. We call the feature that can represent and reason meta-level terms at the object level as *reflection*.

3 An Abstract μ-Calculus Model Checker

```
sorts MuVariable MuFormula

ops False True : ⇀ MuFormula
op ¬ _ : MuFormula ⇀ MuFormula
op _∨_ : MuFormula MuFormula ⇀ MuFormula
op _∧_ : MuFormula MuFormula ⇀ MuFormula
op ⟨_⟩_ : QidList MuFormula ⇀ MuFormula
op [_]_ : QidList MuFormula ⇀ MuFormula
op Nu___ : MuVariable TermSet MuFormula ⇀ MuFormula
op Mu___ : MuVariable TermSet MuFormula ⇀ MuFormula
```

Fig. 1. Symbols for μ-Calculus Terms

We begin with the representation of μ-calculus formulae. A μ-calculus formula is represented by a term of sort MuFormula. Figure 1 shows the symbols in MuFormula terms. In addition to the sort MuFormula, the sort MuVariable is declared to be used in fixed points. The underlines (_) denote the positions of parameters. For instance, the declaration _∧_ specifies that the symbol ∧ is an infix operator. For modal operators, transition labels are quoted identifiers

corresponding to rewriting rule labels. Hence the set of transition labels $\bar{\ell}$ in $\langle\bar{\ell}\rangle\varphi$ and $[\bar{\ell}]\varphi$ is denoted by the built-in sort QidList in $\langle\_\rangle\_$ and $[\_]\_$ respectively. Finally, the state set $\bar{r}$ in the fixed point operators $\mu X\{\bar{r}\}\varphi$ and $\nu X\{\bar{r}\}\varphi$ is represented as a set of meta-level terms. We define the sort TermSet for the representation of meta-level term sets. The symbols $\mathsf{Mu}\_\_\_$ and $\mathsf{Nu}\_\_\_$ form terms of sort MuFormula from a MuVariable term, a TermSet term, and a MuFormula term.

eq $\mathsf{True} = \neg\ \mathsf{False}$
eq $\neg\ \neg\ f = f$
eq $f \wedge g = \neg\ (\neg\ f \vee \neg\ g)$
eq $[\ L\ ]\ f = \neg\ (\langle\ L\ \rangle\ \neg\ f)$
eq $\mathsf{Mu}\ X\ TS\ f = \neg\ (\mathsf{Nu}\ X\ TS\ \mathsf{subst}\ (\neg\ f,\ X,\ \neg\ X))$

Fig. 2. Equations for Derived Operators

To reduce the number of rules in our model checker, Figure 2 provides a set of equations for derived constant and function symbols. These equations follow directly from the corresponding logical equivalence relations. For the greatest fixed point, the substitution of μ-calculus formula is needed. The function subst $(f,\ Z,\ g)$ replaces free occurrences of the variable Z in f by g (Figure 3). If f is the term False or an atomic proposition, subst leaves it unchanged. If f is a MuVariable term not equal to Z, the MuVariable term is returned; otherwise, it is replaced by g. For negative, disjunctive, and existential modal operators, subst invokes itself recursively. Finally, if f is a fixed point formula, subst substitutes the variable Z if Z is not bound. Note that we do not need rules for derived operators.

eq subst $(\mathsf{False},\ Z,\ g) = \mathsf{False}$
eq subst $(p,\ Z,\ g) = p$
ceq subst $(X,\ Z,\ g) = X$ **if** $X \neq Z$
ceq subst $(X,\ Z,\ g) = g$ **if** $X = Z$
eq subst $(\neg\ f,\ Z,\ g) = \neg\ \mathsf{subst}\ (f,\ Z,\ g)$
eq subst $(f_0 \vee f_1),\ Z,\ g) = \mathsf{subst}\ (f_0,\ Z,\ g) \vee \mathsf{subst}\ (f_1,\ Z,\ g)$
eq subst $(\langle\ L\ \rangle\ f,\ Z,\ g) = \langle\ L\ \rangle\ \mathsf{subst}\ (f,\ Z,\ g)$
ceq subst $(\mathsf{Nu}\ X\ TS\ f,\ Z,\ g) = \mathsf{Nu}\ X\ TS\ (\mathsf{subst}\ (f,\ Z,\ g))$ **if** $X \neq Z$
ceq subst $(\mathsf{Nu}\ X\ TS\ f,\ Z,\ g) = \mathsf{Nu}\ X\ TS\ f$ **if** $X = Z$

Fig. 3. Definition of *subst*

We represent Winskel's reduction rules as rewriting rules for entailment terms.[1] Let K be a quoted identifier denoting the name of a rewrite theory,

[1] Equational theory would suffice for model checking, but we need rule labels for formal verification.

```
eq exists (K, s, f, nil, N) = false
eq exists (K, s, f, l L, N) =
   if U ⊢_{l,N} [K, s] → [K, t] then
      (K t ⊢ f) or-else (exists (K, s, f, l L, N + 1))
    else
      exists (K, s, f, L, 0)
   fi
```

Fig. 4. Definition of exists

$\underline{s}$ a meta-level term representing a state, and f a MuFormula term. The entailment term $K\ \underline{s} \vdash f$ is of sort Bool. The idea is to rewrite the entailment term to false or true for any Kripke structure specified by the rewrite theory named K. It is crucial to use a meta-level term $\underline{s}$ in entailment terms. The rules of the rewrite theory $\mathcal{K}$ would rewrite the state s had we used the object-level term s in entailment terms.[2]

It is straightforward to write the rules for Boolean operators in Maude.

$$\textbf{rl}\ [\mathit{ff}] : K\ \underline{s} \vdash \mathsf{False} \Rightarrow \mathsf{false}$$
$$\textbf{rl}\ [\mathit{neg}] : K\ \underline{s} \vdash \neg\ f \Rightarrow \mathsf{not}\ (K\ \underline{s} \vdash f)$$
$$\textbf{rl}\ [\mathit{disj}] : K\ \underline{s} \vdash f_0 \vee f_1 \Rightarrow (K\ \underline{s} \vdash f_0)\ \mathsf{or\text{-}else}\ (K\ \underline{s} \vdash f_1)$$

The rule *ff* rewrites the entailment term $K\ \underline{s} \vdash \mathsf{False}$ to the built-in Bool constant term false. Similarly, the rule *neg* rewrites $K\ \underline{s} \vdash \neg\ f$ to not $(K\ \underline{s} \vdash f)$. The built-in Bool operator not waits until $K\ \underline{s} \vdash f$ rewrites to either false or true, and then rewrites the Bool constant term to its complementary term. The rule *disj* uses the built-in short-circuited Boolean operator or-else. Observe how the computation is performed by a sequence of rewrites in rewriting logic.

For the existential modal operator, we use the following rule:

$$\textbf{rl}\ [\mathit{ex}] : K\ \underline{s} \vdash \langle L \rangle\ f \Rightarrow \mathsf{exists}\ (K, \underline{s}, f, L, 0)$$

The function exists $(K, \underline{s}, f, L, N)$ checks if it is possible to rewrite the entailment $K\ \underline{t} \vdash f$ to true at an L-successor t of s, where N serves as a counter (Figure 4). The built-in QidList term nil represents the empty quoted identifier list. Notice the semantics differ from those in [10]. Our semantics do not have implicit self-loops. If there is no transition label, the function returns false.

On the other hand, the universal theory $\mathcal{U}$ finds the N-th rewriting result t by applying the rule l in $\mathcal{K}$. Then the function exists rewrites the new entailment term $K\ \underline{t} \vdash f$. If it does not rewrite to true, the next successor of s will be checked by exists $(K, \underline{s}, f, l\ L, N + 1)$. On the other hand, if there is no successor of the current label, we look for a successor by applying the next rule.

Observe how the universal theory $\mathcal{U}$ is used to find the successor t of the current state s. The distinction between the object and meta levels clarifies the relation between the model specification and the algorithm implementation. Furthermore, model simulation by reflection allows us to present the algorithm succinctly.

[2] The calligraphic $\mathcal{K}$ is the rewrite theory with the quoted name K.

It is rather straightforward to write the greatest fixed point rules by substitution:

crl $[nu]$: $K\ \underline{s} \vdash \mathsf{Nu}\ X\ TS\ f \Rightarrow \mathsf{true}$
  if $\underline{s}$ isIn TS
crl $[nu]$: $K\ \underline{s} \vdash \mathsf{Nu}\ X\ TS\ f \Rightarrow K\ \underline{s} \vdash \mathsf{subst}\ (f,\ X,\ \mathsf{Nu}\ X\ (\{\underline{s}\}\ \mathsf{U}\ TS)\ f)$
  if $\mathsf{not}\ (\underline{s}\ \mathsf{isIn}\ TS)$

The *nu* rules check whether the current state $\underline{s}$ has been visited. If so, it rewrites the entailment term to true. Otherwise, the current state is added to the meta-level term set TS and the new set is used in the unfolding of the fixed-point formula. The function _isIn_ checks whether a meta-level term is in a term set. Also, the symbol _U_ implements the union of term sets. Both can be easily defined in an equational theory.

4 Concrete Implementation

The rules shown in Section 3 use the pre-defined Maude equations for or-else. Since we cannot fully control internal strategies at object level, we do not know how or-else works internally. In this section, we will get rid of this uncertainty and provide a concrete implementation of the rules.

Consider the definition of exists in Figure 4. We would like the term $(K\ \underline{t} \vdash f)$ or-else $(\mathsf{exists}\ (K,\ \underline{s},\ f,\ l\ L,\ N+1))$ to rewrite $K\ \underline{t} \vdash f$ first, even though there are equational rules for exists in the other subterm. In order not to reduce the second subterm unintentionally, we will not construct a term with the function symbol exists until necessary.

```
sort SuccResult
op ⌊ _,_,_ ⌋ : Term QidList Nat ⇀ SuccResult
op none : ⇀ SuccResult
op succ : Qid Term QidList Nat ⇀ SuccResult
eq succ (K, s, nil, N) = none
eq succ (K, s, l L, N) =
   if U ⊢_{l,N} [K, s] → [K, t] then
      ⌊ t, l L, N + 1 ⌋
   else
      succ (K, s, L, 0)
   fi
```

Fig. 5. Definition of succ

To realize the idea, we define a new function succ $(K,\ \underline{s},\ L,\ N)$ which returns a SuccResult term $\lfloor\ \underline{t},\ L',\ N'\ \rfloor$ if s has a successor t by applying the rules in L (Figure 5). If L is nil, it returns none. Otherwise, succ $(K,\ \underline{s},\ l\ L,\ N)$ checks whether s has the N-th successor t by applying rule l. If so, it returns $\lfloor\ \underline{t},\ l\ L,\ N+1\ \rfloor$. If not, it returns another successor of s by applying the remaining rules in L.

```
op wrapper : Bool Qid Term MuFormula SuccResult ⇀ Bool
eq wrapper (true, K, s, f, R) = true
eq wrapper (false, K, s, f, none) = false
eq wrapper (false, K, s, f, ⌊ t, L, N ⌋) =
   wrapper (K t ⊢ f, K, s, f, succ (K, s, L, N))
```

Fig. 6. Definition of wrapper

```
sorts Mode Proc

op _ _ : Proc Proc ⇀ Proc
ops outCS reqCS inCS : ⇀ Mode
op ◁ _,_,_ ▷ : Nat Mode Bool ⇀ Proc

rl [request0] : ◁ 0, outCS, X ▷  ◁ 1, N, Y ▷ ⇒ ◁ 0 , reqCS, Y ▷  ◁ 1, N, Y ▷
rl [request1] : ◁ 0, M, X ▷  ◁ 1, outCS, Y ▷ ⇒ ◁ 0, M, X ▷  ◁ 1, reqCS, not X ▷
crl [enter0] : ◁ 0, reqCS, X ▷  ◁ 1, N, Y ▷ ⇒ ◁ 0, inCS, X ▷  ◁ 1, N, Y ▷
   if N = outCS or X ≠ Y
crl [enter1] : ◁ 0, M, X ▷  ◁ 1, reqCS, Y ▷ ⇒ ◁ 0, M, X ▷  ◁ 1, inCS, Y ▷
   if M = outCS or X = Y
rl [leave] : ◁ i, inCS, X ▷  ◁ j, N, Y ▷ ⇒ ◁ i, outCS, X ▷  ◁ j, N, Y ▷
```

Fig. 7. Peterson's Algorithm

With the function succ, we can implement the rule *ex* by the wrapper function (Figure 6) as follows.

```
rl [ex] : K s ⊢ ⟨ L ⟩ f ⇒ wrapper (false, K, s, f, succ (K, s, L, 0))
```

To check whether s satisfies $\langle\ L\ \rangle\ f$, we compute the first successor of s by succ $(K, \underline{s}, L, 0)$ and pass the result to wrapper. The wrapper function will check whether the successor satisfies f and compute the next successor. Observe that wrapper does not have a subterm formed by wrapper.

We can use wrapper to implement the rule *disj* as well. The idea is to form a SuccResult term without invoking succ.

```
rl [disj] : K s ⊢ f0 ∨ f1 ⇒ wrapper (K s ⊢ f0, K, s, f1, ⌊ s, nil, 0 ⌋)
```

Similarly, the *nu* rules can be simplified by wrapper:

```
rl [nu] : K s ⊢ Nu X TS f ⇒
   wrapper (s isIn TS, K, s, subst (f, X, Nu X ({ s } U TS) f), ⌊ s, nil, 0 ⌋)
```

5 Verification of Peterson's Algorithm

We verify Peterson's algorithm [18] by our model checker as an example. The mutual exclusion algorithm is shown in Figure 7. Let i be 0 or 1, M a Mode term (outCS, reqCS, or inCS), and X a Bool term, a process term of sort Proc is represented by $\lhd\ i,\ M,\ X\ \rhd$. The rules *request0*, *request1*, *enter0*, *enter1*, and *leave* implement the transitions of Peterson's algorithm by rewriting the composition of two process terms.

In the rule *request0*, process 0 moves from outCS to reqCS by setting its local Bool term to that of process 1. When process 0 is in reqCS, it moves to inCS if process 1 is in outCS or the two local Bool terms are not equal (the rule enter0). Finally, any process can move out of inCS by the rule leave.

Define the initial state term init to be $\lhd$ 0, outCS, false $\rhd$ $\lhd$ 1, outCS, true $\rhd$ and the QidList term labels to be (*'request0 'request1 'enter0 'enter1 'leave*). We are interested in verifying whether the two processes cannot be in the critical section at the same time. Hence, we check whether the entailment term
eq prop0 = 'PETERSON init $\vdash$ Nu X {} ($\neg$ in-cs (0) $\vee$ $\neg$ in-cs (1)) $\wedge$ [labels] X
rewrites to true or not. The rules for the atomic proposition in-cs (i) is defined as follows.

$$\textbf{rl}\ [AP] : \text{'PETERSON}\ \underline{s} \vdash \text{in-cs}\ (i) \Rightarrow \text{critical}\ (s, i)$$

$$\textbf{eq}\ \text{critical}\ (\lhd\ 0, M, X \rhd\ P, 0) = (M = \text{inCS})$$
$$\textbf{eq}\ \text{critical}\ (\lhd\ 1, N, Y \rhd\ P, 1) = (N = \text{inCS})$$

Similarly, we can check if process 0 always enters the critical section first. The corresponding entailment term is the following:
eq prop1 = 'PETERSON init $\vdash$ Nu X {} (in-cs (0) $\vee$ ($\neg$ in-cs (1) $\wedge$ [labels] X))

Finally, we would like to check if process 0 can enter the critical section infinitely often.
eq prop2 = 'PETERSON init $\vdash$ Nu X {} Mu Y {} $\langle$ labels $\rangle$ ((in-cs (0) $\wedge$ X) $\vee$ Y)

The entailment terms prop0, prop1, and prop2 rewrite to true, false, and true in 0.5, $\ll$0.1, $\ll$0.1 seconds by Maude respectively.[3] The model checker contains 250 lines of Maude code. The concise implementation shows that reflection indeed helps in writing an explicit-state model checker. Since model simulation in explicit-state model checkers is implemented by the universal theory $\mathcal{U}$, programmers can pay more attention to the model checking algorithm. Additionally, the short implementation may be feasible for formal analysis. Theorem provers based on rewriting logic (such as ITP [5]) may be used to verify our implementation semi-automatically.

6 Model Checking μ-Calculus Model Checker

The correspondence between Winskel's rules and the concrete implementation is less obvious than that of abstract rules. Additionally, a typo or a missing case in the definitions of subst, succ, wrapper, and term sets may make our implementation incorrect, even if the correspondence is ensured. The verification of Peterson's algorithm in Section 5 only shows that our model checker has *one* intended behavior. It does not imply *all* internal rewriting strategies will produce the same result. Particularly, if our model checker could yield contradictory results or fail to rewrite an entailment term by different strategies, the user would be very confused.

[3] The experiments are conducted in a 2.8GHz Pentium 4 with 2GB memory running Fedora Core 4 Linux system.

These questions call for the analysis of our model checker. Since the abstract algorithm is known to be sound, we are more interested in the correctness of our particular implementation. Specifically, we would like to verify if the concrete implementation always rewrites the entailment terms prop0, prop1, and prop2 to true, false, and true respectively.

This problem can be formalized as follows. Define a Kripke structure $\mathcal{M}_0 = (E, RL, \Rightarrow, \mathsf{prop0}, P)$ where E is the set of all entailment terms, RL the set of all rule labels, and

$$P(e) = \begin{cases} \{ \text{ isTrue } \} & \text{if } e = \mathsf{true} \\ \{ \text{ isFalse } \} & \text{if } e = \mathsf{false} \\ \emptyset & \text{otherwise} \end{cases} .$$

For any two entailment terms e and e', $e \stackrel{l}{\Rightarrow} e'$ if e rewrites to e' by applying the rule l in our model checker. To verify whether the term prop0 always rewrites to true, it amounts to checking whether $\mathcal{M}_0 \models \mu X.\ \text{isTrue} \vee [RL]X$. Similarly, we can define two Kripke structures $\mathcal{M}_1$ and $\mathcal{M}_2$ with initial states prop1 and prop2 respectively, and check $\mathcal{M}_1 \models \mu X.\ \text{isFalse} \vee [RL]X$ and $\mathcal{M}_2 \models \mu X.\ \text{isTrue} \vee [RL]X$. Hence we can resolve the aforementioned questions if we solve these model checking problems.

With the help of reflection, these problems can be solved rather easily. Notice that the Kripke structures $\mathcal{M}_0$, $\mathcal{M}_1$, and $\mathcal{M}_2$ are infinite-state structures. There are countably infinite entailment terms in E. Fortunately, the number of reachable entailment terms is finite because our model checker always terminates. Since both the local model checking and the Maude LTL model checking algorithms explore the reachable states only, they can be used to solve these problems.

6.1 Checking with Abstract Local Model Checker

```
eq M e ⊩ False = false
eq M e ⊩ ¬ f = ¬ (M e ⊩ f)
eq M e ⊩ f0 ∨ f1 = (M e ⊩ f0) or-else (M e ⊩ f1)
eq M e ⊩ ⟨ L ⟩ f = meta-exists (M, e, f, L, 0, false)
ceq M e ⊩ Nu X TS f = true
   if e isIn TS
ceq M e ⊩ Nu X TS f = M e ⊩ subst (f, X, Nu X ({ e } U TS) f)
   if not (e isIn TS)
```

Fig. 8. Abstract Local Model Checker

Let M be the quoted identifier of a model checking theory, e an entailment term, and f a MuFormula term. We define the abstract entailment term M e ⊩ f to be of sort Bool. It is easy to implement Winskel's reduction rules in an

equational theory (Figure 8). However, specifying properties of a model checker exposes a subtle semantic issue. Consider the following entailment term:

'PETERSON $\underline{\mathsf{init}} \vdash$ False.

It rewrites to false trivially. However, $\underline{\mathsf{init}}$ also satisfies $\mu X.$ isTrue $\vee [RL]X$. This is because our model checker always terminates after a finite number of rewrites. Subsequently, the property $[\ L\ ]\ f$ will be true eventually for any QidList term L and MuFormula term f. In the example, the least fixed point rewrites to a disjunction after one unfolding. But the second disjunct rewrites to true because there is no successor.

```
eq meta-exists (M, e, f, nil, N, hasSuccessor) =
     if hasSuccessor then false else M e ⊨ f fi
eq meta-exists (M, e, f, l L, N, hasSuccessor) =
   if U ⊢_{l,N} [M, e] → [M, f] then
      (M f ⊨ f) or-else meta-exists (M, e, l L, N + 1, true)
   else
      meta-exists (M, e, f, L, 0, hasSuccessor)
   fi
```

Fig. 9. Definition of meta-exists

Our solution is to add implicit self-loops to irreducible terms. If an entailment term has successors, we leave them unchanged. But if an entailment term does not have any successor, we make the entailment term to be its only successor. This can be done by the meta-exists function (Figure 9).

The function meta-exists checks if any successor has been found. If there is no label, it reduces to false if the entailment term $\underline{e}$ has other successors. Otherwise, meta-exists checks whether the current entailment term satisfies the MuFormula term f. Effectively, the entailment term $\underline{e}$ is its only successor when no successor can be found.

Let LOCAL-MODEL-CHECK be the name of our model checking rewrite theory and rules the QidList (*'AP 'ff 'neg 'disj 'ex 'nu*). To verify whether prop0 will always rewrite to true, we check whether the entailment term
eq meta-prop0 = 'LOCAL-MODEL-CHECK $\underline{\mathsf{prop0}} \vDash$ Nu X {} isTrue $\vee$ [rules] X
reduces to true where

eq $M\ \underline{e} \vDash$ isTrue = $(\underline{e} = \underline{\mathsf{true}})$.

Similarly, we define entailment terms meta-prop1 and meta-prop2 as follows.
eq meta-prop1 = 'LOCAL-MODEL-CHECK $\underline{\mathsf{prop1}} \vDash$ Nu X {} isFalse $\vee$ [rules] X
eq meta-prop2 = 'LOCAL-MODEL-CHECK $\underline{\mathsf{prop2}} \vDash$ Nu X {} isTrue $\vee$ [rules] X
where

eq $M\ \underline{e} \vDash$ isFalse = $(\underline{e} = \underline{\mathsf{false}})$.

Maude reduces meta-prop0, meta-prop1, and meta-prop2 to true in 341.5, 0.3, 6.5 seconds respectively. Hence the abstract model checker verifies that our model checker always rewrites prop0, prop1, and prop2 to true, false, and true respectively, independent of rewrite strategies.

6.2 Checking with Maude LTL Model Checker

Alternatively, we can use the built-in Maude LTL model checker to verify whether prop0, prop1, and prop2 rewrite to true, false, and true regardless of rewrite strategies. The Maude LTL model checker uses an automata-theoretic algorithm to verify LTL properties. It is implemented in C++ and integrated in Maude version 2 [10, 6].

The Maude LTL model contains several equational theories. Related LTL symbols are defined in the theory $\mathcal{LTL}$. The sorts Prop and Formula defined in $\mathcal{LTL}$ are used for atomic proposition and LTL formula terms. We first define two atomic proposition terms:

op isFalse isTrue : $\rightharpoonup$ Prop

To define the reduction rules for atomic proposition terms, we use the modeling term $\models$ defined in the theory $\mathcal{SATISFACTION}$:

subsort Entailment $\prec$ State
eq $e \models$ isFalse = $(e$ = false$)$
eq $e \models$ isTrue = $(e$ = true$)$.

The term $\_\models\_$ takes a term of sort State (defined in $\mathcal{SATISFACTION}$) and a Prop term to form a Bool term. The equations for isFalse and isTrue tell the Maude LTL model checker how to reduce a modeling term to a Bool term.

The property "p holds eventually" is represented by the LTL term $\Diamond\ p$. Thus, the properties that we would like to verify are represented by $\Diamond$ isFalse and $\Diamond$ isTrue. Finally, we use the built-in function modelCheck to verify whether an initial entailment term rewrites to a Bool term eventually:

modelCheck (prop0, $\Diamond$ isTrue)
modelCheck (prop1, $\Diamond$ isFalse)
modelCheck (prop2, $\Diamond$ isTrue)

The Maude LTL model checker is able to verify these three properties in 2.9, $\ll$0.1, 0.1 seconds respectively. The built-in model checker performs significantly better than our abstract model checker. Since the built-in model checker is implemented in C++, it is expected to run much faster than our abstract model checker. On the other hand, we are free to modify our abstract model checker for different purposes. For instance, the built-in LTL model checker may not terminate on structures with infinite reachable states. But a bounded local model checker for such structures has been implemented using the same framework in [22].

7 Conclusion and Future Work

Reflection has been used for formal metareasoning of membership equational theories [2] and semantics of specifications [7]. In this paper, we present a concise implementation of a local model checking algorithm in the reflective language Maude. We show how the implementation is simplified by exploiting reflection and then verify Peterson's algorithm with our implementation in Maude. In our model checker, the model behavior is explored by the reflective feature of the language. The universal theory is used as a model simulator and thus simplifies

the implementation. Since model simulation is required in explicit-state model checking algorithms, we feel the technique can simplify the implementations of other explicit-state algorithms as well.

More interestingly, we are able to verify our implementation by applying reflection again. We define a Kripke structure of entailment terms characterizing the behavior of our model checker. Hence the verification of our model checkers can be formalized as model checking problems. We then use an abstract local model checker and the Maude LTL model checker to solve these problems. Reflection not only simplifies our implementation of an explicit-state model checking algorithm, but also allows us to model-check our model checker rather easily. To the best of our knowledge, this is the first work which proposes an automatic formal verification technique of model checkers by reflection. From the simplicity of our approach, we believe it will be of use to ensure the quality of other verification tools as well.

Currently, we are interested in applying our technique in other model checking algorithms. Particularly, the analysis of binary decision diagram-based or SAT-based algorithms would be more useful to model checking community. We are investigating the theory developed in [17, 25] and specifying a BDD-based algorithm in rewriting logic as the first step.

Acknowledgments. The author would like to thank anonymous reviewers for their constructive comments and suggestions in improving the paper.

References

1. Basin, D., Clavel, M., Meseguer, J.: Rewriting logic as a metalogical framework. In Kapoor, S., Prasad, S., eds.: The 20th Conference on Foundations of Software Technology and Theoretical Computer Science. Volume 1974 of LNCS., Springer-Verlag (2000) 55–80
2. Basin, D., Clavel, M., Meseguer, J.: Reflective metalogical frameworks. ACM Transactions on Computational Logic **5** (2004) 528–576
3. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
4. Clavel, M.: Reflection in general logics, rewriting logic, and Maude. In Kirchner, C., Kirchner, H., eds.: Proceedings Second International Workshop on Rewriting Logic and its Applications. Volume 15 of Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (1998) 317–328
5. Clavel, M.: The ITP Tool - An Inductive Theorem Prover Tool for Maude Membership Equational Specifications. (2004)
6. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.0 Manuel. version 1.0 edn. (2003)
7. Clavel, M., Martí-Oliet, N., Palomino, M.: Formalizing and proving semantic relations between specifications by reflection. In Stirling, ed.: Algebraic Methodology and Software Technology: 10th International Conference. Volume 3116 of LNCS. (2004) 72–86
8. Clavel, M., Meseguer, J.: Reflection and strategies in rewriting logic. In Meseguer, J., ed.: Proceedings First International Workshop on Rewriting Logic and its Applications. Volume 4 of Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (1996) 125–147

9. Cleaveland, R.: Tableau-based model checking in the propositional mu-calculus. Acta Informatica **27** (1989) 725–747
10. Eker, S., Meseguer, J., Sridharanarayanan, A.: The Maude LTL model checker. In: Proceedings of the Fourth International Workshop on Rewriting Logic. Volume 71 of Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (2002)
11. Holzmann, G.: The model checker SPIN. IEEE Transaction on Software Engineering **23** (1997) 279–295
12. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27** (1983) 333–354
13. Manolios, P. In: Mu-Calculus Model-Checking. Kluwer Academic Publishers (2000) 93–111
14. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theoretical Computer Science **285** (2002) 121–154
15. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96** (1992) 73–155
16. Meseguer, J.: Rewriting logic as a semantic framework for concurrency: A progress report. In Montanari, U., Sassone, V., eds.: CONCUR '96: Concurrency Theory, 7th International Conference. Volume 1119 of LNCS., Springer-Verlag (1996) 331–372
17. van de Pol, J., Zantema, H.: Binary decision diagrams by shared rewriting. In Nielsen, M., Rovan, B., eds.: Mathematical Foundations of Computer Science 2000, 25th International Symposium. Volume 1893 of LNCS., Springer-Verlag (2000) 609–618
18. Silberschatz, A., Galvin, P.B., Gagne, G.: Operating System Concepts. 7th edn. John Wiley & Sons, Inc. (2004)
19. Sprenger, C.: A verified model checker for the modal μ-calculus in coq. In Steffen, B., ed.: Tools and Algorithms for the Construction and Analysis of Systems. Volume 1384 of LNCS., Springer-Verlag (1998) 167–183
20. Stirling, C., Walker, D.: Local model checking in the modal mu-calculus. In Díaz, J., Orejas, F., eds.: Proceedings Int. Joint Conf. on Theory and Practice of Software Development. Volume 351 of LNCS. Springer-Verlag, Berlin (1989) 369–383
21. Wang, B.Y.: μ-calculus model checking in maude. In Martí-Oliet, N., ed.: 5th International Workshop on Rewriting Logic and its Applications. Volume 117 of Electronic Notes in Theoretical Computer Science., Elsevier Science Publishers (2004) 135–152
22. Wang, B.Y.: Automatic verification of a model checker in rewriting logic. Technical Report TR-IIS-05-009, Institute of Information Science, Academia Sinica (2005) http://www.iis.sinica.edu.tw/LIB/TechReport/tr2005/tr05009.pdf.
23. Wang, B.Y., Meseguer, J., Gunter, C.A.: Specification and formal analysis of a PLAN algorithm in Maude. In Hsiung, P.A., ed.: Proceedings International Workshop on Distributed System Validation and Verification. (2000) 49–56
24. Winskel, G.: A note on model checking the modal nu-calculus. Theoretical Computer Science **83** (1991) 157–167
25. Zantema, H., van de Pol, J.: A rewriting approach to binary decision diagrams. Journal of Logic and Algebraic Programming **49** (2001) 61–86