Available online at www.sciencedirect.com

SCIENCE DIRECT®

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 117 (2005) 135–152

www.elsevier.com/locate/entcs

μ-Calculus Model Checking in Maude

Bow-Yaw Wang

Institute of Information Science
Academia Sinica
Nankang 115, Taipei
Taiwan

Abstract

In this paper, a rewrite theory for checking μ-calculus properties is developed. We use the same framework proposed in [11] and demonstrate how rewriting logic can be used as a unified formalism from model specification to verification algorithm implementation. Furthermore, since μ-calculus is more expressive than LTL, this work can be seen as an extension to [11] in theory. We also develop a CTL to μ-calculus translator to help users write CTL specifications more easily. However, the corresponding LTL to μ-calculus translator is missing. The LTL model checker in [11] is still preferred in practice.

Keywords: Rewriting logic, model checking, μ-calculus.

1 Introduction

For the past decade, model checking has found many applications in hardware and software design industry. Given a formal model and its specification, the goal of model checking is to determine whether the model conforms to the specification formally. Take hardware verification as an example. The formal model for digital circuits could be the circuit itself, which consists of combinational and sequential components. The specification could be a predicate over its wires (say, `read` and `write` cannot both be asserted at any time). Using a formal verification tool, a verification engineer can check whether the predicate holds universally.

[⋆] Partially supported by National Science Council NSC 92-2213-E-001 -023-. Author's URL: `www.iis.sinica.edu.tw/~bywang`

doi:10.1016/j.entcs.2004.06.025

From the scenario illustrated above, we observe that the process of model verification can be divided into three parts:

- Specification of the model;
- Specification of the property; and
- Mechanism to check the property against the model.

Indeed, the observation was made in [23] where the authors propose to use Maude [5,18] as a verification platform for model checking. Maude is a rewriting system based on rewriting logic. Since rewriting logic was introduced in [16], it has been used as a unified formalism for modeling concurrency [16,17,19] and as a logical framework [3]. Thanks to the expressive power of rewriting logic, it is possible for researchers from various research communities to work within a single formalism.

In the light of the unification of formalisms within rewriting logic, it is perhaps an interesting challenge to ask whether it is possible to perform all three tasks of model verification in rewriting logic. The challenge has been partially answered in [23] where an active network protocol is verified. In [23], the protocol and a property are specified at object level. The authors use a meta-level theory to explore all possible behaviors of the model and check it against the property. However, the result is not satisfactory since only invariant checking is discussed under the proposed framework.

More recently, a linear temporal logic (LTL) model checker has been described in [11]. The model is specified as a module in Maude. The property specification language is defined by an equational theory. The user can specify the model with a rewrite theory, import the specification equational theory to write LTL formula, then invoke the built-in model checking algorithm to verify whether the model satisfies the LTL formula. However, the model checking algorithm is implemented in C++. Thus, we do not think the challenge has been met fully.

In this work, we propose a new idea to answer the challenge. Our solution uses a rewrite theory to model specification as in [11]. Instead of LTL, a more expressive specification logic, the μ-calculus, is used. Moreover, we are able to construct the model checker in rewriting logic rather than implementing our μ-calculus model checker in C++. In other words, we have successfully achieved all three tasks of model verification in rewriting logic.

It is known that μ-calculus is strictly more expressive than CTL*. Consequently, the present work can be seen as an extension to the results in [11]. In practice, however, CTL or LTL formulae are preferred because μ-calculus formulae sometimes are hard to understand. We therefore provide a rewrite theory for translating any given CTL formula to its equivalent μ-calculus formula. Unfortunately, we still do not know how to translate LTL formulae. Hence the LTL model checker in [11] is still needed in practice.

The key to implement a μ-calculus model checker in rewriting logic is local model checking [20,6,22]. Unlike traditional model checking algorithms where all states satisfying the property are computed, local model checking tries to find a proof for the initial state. Since the proof tree is generated syntactically, rewriting logic can be used to implement the algorithm. Furthermore, states are explored only when necessary during the proof search. As a result, it is possible to verify properties for infinite-state systems should they be proved without traversing infinite number of states.

In this report, we briefly review the framework proposed in [11] and focus on developing a Maude theory for a μ-calculus local model checker. The paper is organized as follows. We start with preliminaries in Section 2. The overview of model checking in Maude is given in Section 3. The Maude theory for μ-calculus local model checking is presented in Section 4. It is followed by a simple theory for translating CTL to μ-calculus in Section 5. Finally, we discuss some future directions and conclude in Section 6.

2 Preliminary

A μ-calculus formula ϕ is generated by the following rules:

- propositional variables: $X, Y, Z, \ldots$;
- atomic propositions (AP): $p, q, r, \ldots$;
- Boolean connectives: $\neg\phi$, $\phi \wedge \psi$;
- modal next-state operator: $\Diamond\phi$;
- greatest fixed-point operator: $\nu X.\phi$, where the bound variable X occurs positively.

As usual, we use derived operators such as $\phi \vee \psi (\equiv \neg(\neg\phi \wedge \neg\psi))$, $\Box\phi (\equiv \neg\Diamond\neg\phi)$ and $\mu X.\phi (\equiv \neg\nu X.\neg\phi[\neg X/X])$. In [13], labeled modal next-state quantifier $< a >$ is allowed. Here we restrict ourselves to the unlabeled version for simplicity. Note that the restriction is expressive enough to include CTL* [7].

The semantics of ϕ is defined over a *Kripke structure* $K = (S, \rightarrow, s_0, L)$ where S is the set of states, $\rightarrow \subseteq S \times S$ the transition relation, $s_0 \in S$ the initial state, and $L : S \rightarrow 2^{AP}$ the labeling function which maps each state to a subset of atomic propositions. For clarity, we write $s \rightarrow t$ whenever $(s, t) \in \rightarrow$. 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 $X \not\equiv Y$. Given the valuation ρ, the semantic function $[\![\bullet]\!]\rho$ that returns a set of states satisfying ϕ under the valuation ρ is defined as follows.

- $[\![X]\!]\rho = \rho(X)$;
- $[\![p]\!]\rho = \{s \in S : p \in L(s)\}$;
- $[\![\neg\phi]\!]\rho = S \setminus [\![\phi]\!]\rho$;
- $[\![\phi \wedge \psi]\!]\rho = [\![\phi]\!]\rho \cap [\![\psi]\!]\rho$;
- $[\![\Diamond\phi]\!]\rho = \{s \in S : \exists t \in S.s \to t \text{ and } t \in [\![\phi]\!]\rho\}$;
- $[\![\nu X.\phi]\!]\rho = \cup\{R \subseteq S : R \subseteq [\![\phi]\!](\rho[X \mapsto R])\}$.

Given a μ-calculus formula ϕ and a Kripke structure $K = (S, \to, s_0, L)$, we write $K, s_0 \models \phi$ when $s_0 \in [\![\phi]\!]\emptyset$. In other words, $K, s_0 \models \phi$ means s_0 of K satisfies the μ-calculus formula ϕ. The μ-calculus model checking problem is to check whether $K, s_0 \models \phi$.

Traditionally, μ-calculus model checking problem is solved by computing *all* states S_ϕ satisfying the given μ-calculus formula ϕ and checking whether $s_0 \in S_\phi$ [10,2]. Here, we use the local model checking technique to solve the problem. Instead of computing S_ϕ, local model checking tries to find a proof for the state s_0. This proof-theoretic approach explores the Kripke structure locally and is more suitable for logical frameworks like rewriting logic.

A local model checker consists of a set of proof rules [20,6,22,1]. It tries to search a complete proof tree for $K, s \vdash \phi$ according to these proof rules. In this work, we use a simple extension of the proof rules in [22]. In order to present his rules, Winskel introduces a new μ-calculus formula $\nu X\{\bar{r}\}\phi$ where $\{\bar{r}\} \subseteq S$. Its semantics is defined by:

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

Note that $\nu X.\phi$ is equivalent to $\nu X\{\}\phi$. The following proof rules can be found in [22]:

- $(K, s \vdash p) \Rightarrow$ **true** if $p \in L(s)$;
- $(K, s \vdash p) \Rightarrow$ **false** if $p \notin L(s)$;
- $(K, s \vdash T) \Rightarrow$ **true**;
- $(K, s \vdash F) \Rightarrow$ **false**;
- $(K, s \vdash \neg\phi) \Rightarrow \neg b$ if $(K, s \vdash \phi) \Rightarrow^* b$;
- $(K, s \vdash \phi \wedge \psi) \Rightarrow b_0 \wedge b_1$ if $(K, s \vdash \phi) \Rightarrow^* b_0$ and $(K, s \vdash \psi) \Rightarrow^* b_1$;
- $(K, s \vdash \phi \vee \psi) \Rightarrow b_0 \vee b_1$ if $(K, s \vdash \phi) \Rightarrow^* b_0$ or $(K, s \vdash \psi) \Rightarrow^* b_1$;
- $(K, s \vdash \Diamond\phi) \Rightarrow$ **true** if $(K, t \vdash \phi) \Rightarrow^*$ **true** for some t such that $s \to t$;
- $(K, s \vdash \nu X\{\bar{r}\}\phi) \Rightarrow$ **true** if $s \in \{\bar{r}\}$;
- $(K, s \vdash \nu X\{\bar{r}\}\phi) \Rightarrow (K, s \vdash \phi[\nu X\{s, \bar{r}\}\phi/X])$.

```
(fmod MU is
  sorts Variable Prop Formula .
  subsort Variable < Formula .
  subsort Prop < Formula .

*** primitive operators ***
  ops True False : -> Prop .
  op ~_          : Formula -> Formula [ prec 52 ] .
  op _/\_        : Formula Formula -> Formula [ comm prec 55 ] .
  op _\/_        : Formula Formula -> Formula [ comm prec 59 ] .
  op <>_         : Formula -> Formula [ prec 53 ] .
  op ‘[‘]_       : Formula -> Formula [ prec 53 ] .
  op Mu__        : Variable Formula -> Formula [ prec 61 ] .
  op Nu__        : Variable Formula -> Formula [ prec 61 ] .
endfm)
```

Fig. 1. Maude module MU

For any ν-calculus formula ϕ, it is shown that $(K, s_0 \vdash \phi) \Rightarrow^* \mathbf{true}$ if and only if $K, s \models \phi$ [22]. As a result, one can solve the model checking problem by specifying these proof rules as equations in Maude. However, there are a few issues to be addressed. The corresponding equations for μ-operators are needed for efficiency, as well as a mechanism to mimic variable substitution in proof rules.

Although μ-calculus is very expressive, it is sometimes difficult to interpret, even for experts. In order to help users to write specifications, we implement the translation from CTL to μ-calculus. A CTL formula α is built from the following constructs recursively [8,9]:

- atomic propositions: p, q, r;
- Boolean connectives: $\neg\alpha$ and $\alpha \wedge \beta$;
- existential next-step operator: $\mathsf{EX}\alpha$;
- existential always operator: $\mathsf{EG}\alpha$;
- existential until operator: $\mathsf{E}(\alpha\mathsf{U}\beta)$.

Other operators can be derived from them. For instance, $\mathsf{AX}\alpha(\equiv \neg\mathsf{EX}\neg\alpha)$, $\mathsf{EF}\alpha(\equiv \mathsf{E}(\mathbf{true}\mathsf{U}\alpha))$, $\mathsf{AG}\alpha(\equiv \neg\mathsf{EF}\neg\alpha)$ and $\mathsf{A}(\alpha\mathsf{U}\beta)(\equiv \neg\mathsf{E}(\neg\beta\mathsf{U}\neg\alpha \wedge \neg\beta) \wedge \neg\mathsf{EG}\neg\beta)$. CTL specifications are used in many formal verification tools such as SMV [14,15] and VIS [4]. Our translation would allow users familiar with these systems to adopt Maude as a verification platform easily.

3 Model Checking in Maude

We start with the equational theory for μ-calculus formulae (Figure 1). Three sorts are declared in the module `MU`: `Variable`, `Prop` and `Formula`. μ-calculus formulae are of sort `Formula`. Atomic propositions are of sort `Prop`. Finally, propositional variables are of sort `Variable`. Boolean connectives are defined as usual. For modal operators $\Diamond$ and $\Box$, we use `<>` and `[]` respectively.

```
(mod MUTEX is
  protecting MACHINE-INT .

  sorts Mode Proc Token Conf .
  subsorts Token Proc < Conf .

  op __ : Conf Conf -> Conf [ assoc comm ] .
  ops wait critical : -> Mode .
  op ‘[_‘,_‘] : MachineInt Mode -> Proc .
  op * : -> Token .

  vars m n : MachineInt .
  var C : Conf .

  crl [enter] : * [n, wait] [m, critical] =>
                [n - 1, wait] [m + 1, critical]   if n > 0 .
  crl [exit]  : [n, critical] [m, wait] =>
                * [n - 1, critical] [m + 1, wait] if n > 0 .
endm)
```

Fig. 2. Module `MUTEX`

The operators `Mu` and `Nu` represent least and greatest fixed point operators respectively. As an example, the μ-calculus formula $\nu Y.\Diamond(Y \wedge \mu Z.p \vee \Diamond Z)$ is written as

```
Nu Y (<> (Y /\ (Mu Z (p \/ <> Z)))).
```

For the specification of Kripke structures, we follow the infrastructure proposed in [11] and give a brief review here. The Kripke structure $K = (S, \rightarrow, s_0, L)$ is specified as a rewrite theory $\mathcal{R} = (\Sigma, E, R)$. Let $[s]$ be the equivalent class of the term s in $\mathcal{R}$. Then $[s] \Rightarrow [t]$ is a rewrite proof in the initial model of $\mathcal{R}$ if and only if $s \rightarrow t$ in K. Consider the module `MUTEX` in Figure 2 as an example, which is a simple extension of one found in [5]. The configuration `[`m`, wait]` represents m processes in mode `wait`, similarly for `[`n`, critical]`. Any process in mode `wait` may grab a token ('`*`') and enter mode `critical`. On the other hand, any process in mode `critical` may go back to mode `wait` by releasing a token. For instance, the configuration `* * * [5, wait] [2, critical]` can be rewritten to `* * [4, wait] [3, critical]` by applying the rule `enter`.

To define the labeling function in Kripke structure, we introduce the operator `|-` in module `ENTAILMENT`:

```
op __|-_ : Environment Term Formula -> Bool [ prec 85 ].
```

Let E be a variable environment (described later), $\bar{s}$ the meta-level term representing the state s, and ϕ a μ-calculus formula, then $E\ \bar{s} \vdash \phi$ represents that we would like to check whether s satisfies ϕ under the environment E. Notice that states are represented as meta-level terms, rather than elements of sort `State` as in [11]. Since successors of states are needed to prove modal operators $\Diamond$ and $\Box$, meta-level representation is used to compute all succes-

```
(fmod MUTEX-PREDS is
  protecting MACHINE-INT .
  protecting MU .
  protecting ENTAILMENT .

  op crit : MachineInt -> Prop .
  op wait : MachineInt -> Prop .
  op get-critical : TermList -> MachineInt .
  op get-wait : TermList -> MachineInt .

  var E : Environment .
  vars n : MachineInt .
  vars T : Term .

  ceq E T |- crit (n) = true if get-critical (T) >= n .
  ceq E T |- crit (n) = false if get-critical (T) < n .

  ceq E T |- wait (n) = true if get-wait (T) >= n .
  ceq E T |- wait (n) = false if get-wait (T) < n .

*** get-critical and get_wait are omitted here
endfm)
```

Fig. 3. Module MUTEX-PREDS

sors. [1]

Using the module ENTAILMENT, we can define the labeling function. Instead of explicitly mapping each state to atomic propositions, it is more convenient to specify a state predicate indicating whether an atomic proposition holds at the state. In other words, we would like to reduce the term $E\ \bar{s} \vdash p$ to **true** whenever the atomic proposition p holds in state s. In Figure 3, the module MUTEX-PREDS defines two atomic propositions: crit(n) and wait(n). If n or more processes are in mode critical (wait, respectively) the proposition crit(n) (wait(n), respectively) evaluates to true. Recall that system states are represented at meta level. Hence meta-level functions get-critical and get-wait are used to extract the number of processes. [2] In contrast to [11], note that we need to define when the atomic propositions evaluate to **false**.

Figure 4 shows the module MUTEX-CHECK for model checking the specification MUTEX. The parameterized module LOCAL-MODEL-CHECK is our rewrite theory for μ-calculus model checker. It accepts modules of the interface theory KMODULE:

```
(fth KMODULE is
  protecting META-LEVEL .
  op KRIPKE : -> Module .
  op labels : -> QidList .
endfth)
```

To instantiate the module parameter, we first define a meta-level module

[1] It is unnecessary to use meta-level representation in Maude 2 since the function metaXapply can be used to compute successors.

[2] We need not work at meta level should the object-level representation is used.

```
(fmod MUTEX-CHECK is
  protecting MACHINE-INT .
  protecting MUTEX-PREDS .
  protecting LOCAL-MODEL-CHECK [KripkeMUTEX] .

  ops init : -> Term .
  eq init = up (MUTEX, * * * * * [100000, wait] [0, critical]) .

  ops X Y : -> Variable .

  ops prop prop0 prop1 prop2 prop3 prop4 : -> Bool .

  eq prop0 = {} init |- Nu X ~ (<> ~ X \/ (crit(6))) .
  eq prop1 = {} init |- Mu X (<> X \/ (crit(6))) .
  eq prop2 = {} init |- Nu X ([] X /\ crit(5)) .
  eq prop3 = {} init |- Nu X (<> True /\ [] X) .
endfm)
```

Fig. 4. Maude module `MUTEX-CHECK`

`META-MUTEX` as follows.

```
(mod META-MUTEX is
  protecting META-MODULE .
  op KRIPKE : -> Module .
  eq KRIPKE = up (MUTEX) .

  op labels : -> QidList .
  eq labels = ( 'enter 'exit ) .
endm)
```

We then define the view `KripkeMUTEX` that maps `KMODULE` to `META-MUTEX` as follows.

```
(view KripkeMUTEX from KMODULE to META-MUTEX is
  op KRIPKE to KRIPKE .
  op labels to labels .
endv)
```

For our exemplary Kripke structure, there are 100,000 processes in mode `wait` initially. `prop1` specifies whether there will be more than 5 processes in mode `critical` eventually along some computation path. `prop0` is its negation. `prop2` asks if there will be more than 4 processes waiting for all computation paths eventually. The most interesting one is `prop3` which specifies that the model is deadlock-free.

We can check `prop0` by reducing the entailment:

```
Maude> (red prop0 .)
rewrites: 5457 in 480ms cpu (510ms real) (11368 rewrites/second)
reduce in CHECK : prop0 .
result Bool : true
```

In this section, we review the interface of our Maude μ-calculus model checker. It is not unlike those proposed in [11]. Hence the expressive power of Maude can be used to model sophisticated systems, as in [11]. In contrast to the black-boxed implementation in [11], our model checking algorithm is specified by rewriting logic. Consequently, the algorithm can be further improved by any experienced Maude user without resorting to low-level C++ program-

```
(fmod ENVIRONMENT [K :: KMODULE] is
  protecting META-LEVEL .
  protecting MU .
  protecting ENTAILMENT .

  sort TermSet Definition .
  subsort Term < TermSet .
  subsort Definition < Environment .

  vars T T’ : Term .
  var TS : TermSet .

*** meta-level set
  op emptyTermSet : -> TermSet .
  op _U_ : TermSet TermSet -> TermSet [ assoc comm id: emptyTermSet ] .
  op _isIn_ : Term TermSet -> Bool .

  ceq T U T’ U TS = T U TS
    if meta-reduce (KRIPKE, ’_==_[T, T’]) == {’true}’Bool .

  eq T isIn emptyTermSet = false .
  ceq T isIn (T’ U TS) = true
    if meta-reduce (KRIPKE, ’_==_[T, T’]) == {’true}’Bool .
  ceq T isIn (T’ U TS) = T isIn TS
    if meta-reduce (KRIPKE, ’_=/=_[T, T’]) == {’true}’Bool .

*** Definition
  op _:=___ : Variable Bool TermSet Formula -> Definition [ prec 81 ] .

*** Environment
  op ‘{‘} : -> Environment .
  op _&_ : Environment Environment -> Environment
             [ ctor assoc comm id: {} prec 83 ] .
endfm)
```

Fig. 5. Module `ENVIRONMENT [K :: KMODULE]`

ming. In the following section, we discuss the implementation in detail.

4 Equational Theory for μ-Calculus Model Checking

Let us begin with the sort `Environment` defined in the parameterized module `ENVIRONMENT` (Figure 5). An environment consists of a set of definitions. Each definition in turn contains a variable, a Boolean value, a set of states (represented by meta-level terms) and a formula. To understand why definitions are defined as such, recall the proof rules for greatest fixed point operator:

- $(K, s \vdash \nu X\{\bar{r}\}\phi) \Rightarrow$ **true** if $s \in \{\bar{r}\}$; and
- $(K, s \vdash \nu X\{\bar{r}\}\phi) \Rightarrow (K, s \vdash \phi[\nu X\{s, \bar{r}\}\phi/X])$ otherwise.

When formula $\nu X\{\bar{r}\}\phi$ is encountered, the proof rule first checks whether the state s belongs to the set of states $\{\bar{r}\}$ associated with the formula. If so, we are done. Otherwise, ϕ is unrolled once with s added to $\{\bar{r}\}$. It is now easy to see why we define `Definition` as in Figure 5. The variable and formula are used for substitution. The Boolean value is used to distinguish greatest from least fixed point operators. And the meta-level term set is used to simulate the set operations (such as element addition and membership) at object level. Strictly speaking, our definition of environment is not the most general one. It cannot be used to check μ-calculus formulae with multiple occurrences of a propositional variable. Consider the formula $\nu Z.Z \vee \Diamond\Diamond Z$. Since the environment does not differentiate different occurrences of the variable Z, our model checker will yield incorrect results. The problem can be resolved by keeping an environment for each occurrence of propositional variable. For simplicity, we do not consider such formulae.

Note that states are represented by meta-level terms. The aforementioned set operations need be performed at meta level. Additionally, the underlying equational theory of the Kripke structure should be applied. For this reason, we let module `ENVIRONMENT` be parameterized by the Kripke structure and use `meta-reduce` in related set operators.

Now we can present the equational theory for proof rules. For Boolean connectives, it is straightforward to write corresponding equations. For instance, the equation for conjunction rule is:

```
eq E s |- f /\ g = if (E s |- f) then (E s |- g) else false fi .
```

For each μ-calculus operator, we have an equation for its negative form. This gives us a more direct proof of termination for full μ-calculus [20]. The negative equation has the same form as its logically equivalent formula. The corresponding equation for conjunction is:

```
eq E s |- ~ (f /\ g) =
  if (E s |- ~ f) then true else (E s |- ~ g) fi .
```

For the modal next operator, recall

$$(K, s \vdash \Diamond\phi) \Rightarrow \textbf{true} \text{ if } (K, t \vdash \phi) \Rightarrow^* \textbf{true} \text{ for some } t \text{ such that } s \rightarrow t.$$

Thus the term $E\ \bar{s} \vdash \Diamond\phi$ is reduced to the disjunction of $E\ \bar{t} \vdash \phi$ ranging over all successors t of s. Here we use meta-level theory to find all successors of the state s. This explains why we use meta-level representation in definitions. In Figure 6, we show a modified version of the meta-level function `allRew` in [21] to compute successors of a given term. Interested readers are referred to [21] for details. [3]

With function `successors` in place, it is easy to define the equation for the next modal operator:

```
eq E s |- <> f = OR (s, E, f, 0) .
```

where

```
eq OR (s, E, f, n) =
  if (successor (s, labels, n)) == error* then false
  else if (E (successor (s, labels, n)) |- f) then true
```

[3] In Maude 2, we can simply invoke `metaXapply` and get corresponding terms of the successors. Hence it is not necessary to use meta-level theory.

```
(fmod SUCCESSOR [K :: KMODULE] is
  protecting QID-LIST .
  protecting MACHINE-INT .

  *** variable declaration here
  op ~ : -> TermList .
  op successor : Term QidList MachineInt -> Term .
  op lowerRew : Term Qid MachineInt -> Term .
  op rewArgs : Qid TermList TermList Qid MachineInt -> Term .
  op rebuild : Qid TermList Term TermList -> Term .
  op meta-apply' : Term Qid MachineInt -> Term .
  op get-t : ResultPair -> Term .

  eq successor (T, nil, n) = error* .
  eq successor (T, L LS, n) =
    if meta-apply' (T, L, n) == error*
      then if lowerRew (T, L, n) == error*
             then successor (T, LS, n)
             else lowerRew (T, L, n)
           fi
      else meta-apply' (KRIPKE, T, n)
    fi .

  eq get-t ({T, SB}) = T .
  eq meta-apply' (T, L, n) =
    get-t (meta-apply (KRIPKE, T, L, none, n)) .

  eq lowerRew ({C}S, L, n) = error* .
  eq lowerRew (OP[TL], L, n) = rewArgs (OP, ~, TL, L, n) .

  eq rewArgs (OP, Now, T, L, n) =
    if successor (T, L, n) == error*
      then error*
      else rebuild (OP, Now, successor (T, L, n), ~)
    fi .

  eq rewArgs (OP, Now, (T, After), L, n) =
    if successor (T, L, n) == error*
      then rewArgs (OP, (Now, T), After, L, n)
      else rebuild (OP, Now, successor (T, L, n), After)
    fi .

  eq rebuild (OP, Now, T, After) =
    meta-reduce (KRIPKE, OP[Now, T, After]) .
endfm)
```

Fig. 6. Module `SUCCESSOR [K :: KMODULE]`

```
    else OR (s, E, f, n + 1) fi fi .
```

For the greatest fixed point formulae of the form $\nu X.\phi$, the definition $X :=$ **true** $\bar{s}$ ϕ is added to the environment when the formula is encountered for the first time. Hence the entailment `E s |- Nu X f` rewrites to `E & (X := true s f) s |- f` where `X := true s f` records the definition of function `f`, the Boolean value `true` for the greatest fixed point operator, and the visited state `s`.

```
  eq {} s |- Nu X f = (X := true s f) s |- f .
  ceq E & (Y := b S g) s |- Nu X f =
    E & (Y := b S g) & (X := true s f) s |- f if X =/= Y .
```

If, on the other hand, the formula $\nu X.\phi$ has been added to the environment, we need determine whether the formula should be unfolded. As in [22], there are two cases. If the formula is encountered by the same state again, it rewrites to `true`. Otherwise, the state is stored in the definition and the formula is unfolded.

```
ceq E & (X := true S f) s |- X = true if s isIn S .
ceq E & (X := true S f) s |- X =
  E & (X := true (s U S) f) s |- f if not (s isIn S) .
```

The function `isIn` checks whether the meta-level term `s` belongs to the set `S` of meta-level terms. Hence the first equation reduces to `true` if `s` has been visited. The expression `s U S` evaluates to a new set by adding `s` to the set `S`. The second equation records the current state if it has not been visited.

For formulae with least fixed point operators, we could rewrite them to equivalent formulae with only greatest fixed point operators by applying logical equivalence $\mu X.\phi \Leftrightarrow \neg\nu X.\neg\phi[\neg X/X]$ recursively. Here, we would like to take a more direct approach. Observe

$$\begin{aligned} K, s \vdash \neg\nu X\{\bar{r}\}\neg\phi[\neg X/X] &\Leftrightarrow K, s \vdash \neg(\neg\phi[\neg X/X][\nu X\{s, \bar{r}\}\neg\phi[\neg X/X]/X]) \\ &\Leftrightarrow K, s \vdash \phi[\neg\nu X\{s, \bar{r}\}\neg\phi[\neg X/X]/X] \end{aligned}$$

Therefore, we may define $\mu X\{\bar{r}\}\phi$ to be $\neg\nu X\{\bar{r}\}\neg\phi[\neg X/X]$ and obtain

$$\begin{aligned} K, s \vdash \mu X\{\bar{r}\}\phi &\Leftrightarrow K, s \vdash \neg\nu X\{\bar{r}\}\neg\phi[\neg X/X] \\ &\Leftrightarrow K, s \vdash \phi[\neg\nu X\{s, \bar{r}\}\neg\phi[\neg X/X]/X] \\ &\Leftrightarrow K, s \vdash \phi[\mu X\{s, \bar{r}\}/X] \end{aligned}$$

For the terminating condition, consider $K, s \vdash \mu X\{s\}\phi$. We have

$$\begin{aligned} K, s \vdash \mu X\{s\}\phi &\Leftrightarrow K, s \vdash \neg\nu X\{s\}\neg\phi[\neg X/X] \\ &\Leftrightarrow \mathbf{not}(K, s \vdash \nu X\{s\}\neg\phi[\neg X/X]) \\ &\Leftrightarrow \mathbf{false} \end{aligned}$$

Thus, the equations for least fixed point formulae are

```
eq {} s |- Mu X f = (X := false s f) s |- f .
ceq (E & (Y := b S g)) s |- Mu X f =
  E & (Y := b S g) & (X := false s f) s |- f if X =/= Y .

ceq E & (X := false S f) s |- X = false if s isIn S .
ceq E & (X := false S f) s |- X =
  E & (X := false (s U S) f) s |- f if not (s isIn S) .
```

The first two equations add the least fixed point definition to the environment. If the state has not been encountered, the fourth equation records it and unfolds the propositional variable. Except the definition, they are the same as the equations for greatest fixed points. The third equation, however, reduces to `false` if the state has been visited.

```
(fmod LOCAL-MODEL-CHECK [K :: KMODULE] is
  protecting META-LEVEL .
  protecting MU .
  protecting ENTAILMENT .
  protecting ENVIRONMENT [K] .
  protecting SUCCESSOR [K] .

*** variable declaration here
  ops OR AND : Term Environment Formula MachineInt -> Bool .

  eq E s |- True = true .
  eq E s |- False = false .
  eq E s |- ~ prop = not (E s |- prop) .

  eq E s |- ~ ~ f = E s |- f .

  eq E s |- f /\ g = if (E s |- f) then (E s |- g) else false fi .
  eq E s |- f \/ g = if (E s |- f) then true else (E s |- g) fi .

  eq E s |- <> f = OR (s, E, f, 0) .
  eq OR (s, E, f, n) =
    if (successor (s, labels, n)) == error* then false
    else if (E (successor (s, labels, n)) |- f) then true
    else OR (s, E, f, n + 1) fi fi .

  eq E s |- [] f = AND (s, E, f, 0) .
  eq AND (s, E, f, n) =
    if (successor (s, labels, n)) == error* then true
    else if (E (successor (s, labels, n)) |- f) then
      AND (s, E, f, n + 1)
    else false fi fi .

  eq {} s |- Mu X f = (X := false s f) s |- f .
  ceq (E & (Y := b S g)) s |- Mu X f =
    E & (Y := b S g) & (X := false s f) s |- f if X =/= Y .

  eq {} s |- Nu X f = (X := true s f) s |- f .
  ceq E & (Y := b S g) s |- Nu X f =
    E & (Y := b S g) & (X := true s f) s |- f if X =/= Y .

  ceq E & (X := b S f) s |- X = b if s isIn S .
  ceq E & (X := b S f) s |- X =
    E & (X := b (s U S) f) s |- f if not (s isIn S) .

*** negated equations are omitted
endfm)
```

Fig. 7. Maude Module LOCAL-MODEL-CHECKER [K :: KMODULE]

It is worth noting that all the fixed point proof rules have their semantic foundation [22,1]. From semantic point of view, our equations are essentially the same as those in [20,6,22,1].

We can actually reduce the number of equations by introducing a Boolean variable. For instance, the terminating equations for both greatest and least fixed point formulae can be reduced to the following equation:

```
  ceq E & (X := b S f) s |- X = b if s isIn S .
```

Similarly, we can merge two unfolding equations as one. The full theory for our μ-calculus local model checker is shown in Figure 7.

```
(fmod CTL is
  protecting MU .
  sort CTLFormula .
  subsort Prop < CTLFormula .

*** primitive operators ***
  op !_         : CTLFormula -> CTLFormula [ prec 53 ] .
  ops _&&_ _||_  : CTLFormula CTLFormula -> CTLFormula
                   [ comm prec 55 ] .
  ops EX_    : CTLFormula -> CTLFormula [ prec 53 ] .
  ops EG_    : CTLFormula -> CTLFormula [ prec 63 ] .
  ops E_U_   : CTLFormula CTLFormula -> CTLFormula
                   [ prec 63 ] .

*** derived operators ***
  ops AX_         : CTLFormula -> CTLFormula [ prec 53 ] .
  ops EF_ AF_ AG_ : CTLFormula -> CTLFormula [ prec 63 ] .
  ops A_U_   : CTLFormula CTLFormula -> CTLFormula
                   [ prec 63 ] .

*** derived equations are omitted
endfm)
```

Fig. 8. Maude module `CTL`

We present a rewriting theory for model checking μ-calculus formulae. In contrast to [11], our model checker is implemented in rewriting logic. Readers can examine and improve the model checking algorithm by modifying the Maude module `LOCAL-MODEL-CHECK`. It is possible to improve the proof strategy by a meta-level Maude theory. For instance, the conjunction and disjunction equations have been modified to use short-cut evaluation in `LOCAL-MODEL-CHECK`. These are desirable features missing from the black-boxed approach in [11].

5 Model Checking CTL Formula

There is, however, an issue of acceptability in μ-calculus model checking. μ-calculus formulae do look arcane to untrained eyes. Take the specification of deadlock freedom as an example. The μ-calculus formula $\nu X.\Diamond\mathbf{true} \wedge \Box X$ is less obvious than the corresponding CTL formula $\mathsf{AGEX}\mathbf{true}$. If the formula contains mutual fixed points, it would be more difficult to interpret. Fortunately, it is easy to translate any CTL formula to its corresponding μ-calculus formula. In this section, we shall present a Maude equational theory to help users write CTL specifications.

As mentioned in Section 2, there are only three primitive temporal operators in CTL: EX, EG and EU. The derived operators AX, EF, AG, AF and AU are defined as usual (Figure 8).

Since Boolean connectives can be mapped trivially, it remains to translate primitive temporal operators to their corresponding μ-calculus formulae. Hence we define the translation function τ over the primitive temporal operators:

- $\tau(\mathbf{true}, c) = \mathbf{true}$;
- $\tau(p, c) = p$;
- $\tau(\neg\alpha, c) = \neg\tau(\alpha, c)$;
- $\tau(\alpha \wedge \beta, c) = \tau(\alpha, c) \wedge \tau(\beta, c + \theta(\alpha))$;
- $\tau(\mathsf{EX}\alpha, c) = \Diamond\tau(\alpha, c)$;
- $\tau(\mathsf{E}(\alpha\mathsf{U}\beta), c) = \mu X_c.\tau(\beta, c + 1) \vee (\tau(\alpha, c + \theta(\beta) + 1) \wedge \Diamond X_c)$;
- $\tau(\mathsf{EG}\alpha, c) = \nu X_c.\tau(\alpha, c + 1) \wedge \Diamond X_c$.

where $\theta(\alpha)$ computes the number of fixed point operations required in α:

- $\theta(\mathbf{true}) = 0$;
- $\theta(p) = 0$;
- $\theta(\neg f) = \theta(f)$;
- $\theta(f \wedge g) = \theta(f) + \theta(g)$;
- $\theta(\mathsf{EX}f) = \theta(f)$;
- $\theta(\mathsf{EG}f) = \theta(f) + 1$;
- $\theta(\mathsf{E}f\mathsf{U}g) = \theta(f) + \theta(g) + 1$;

In our translation, we increment the index c to make a fresh propositional variable. The function $\tau(\alpha, c)$ returns an equivalent μ-calculus formula which uses propositional variables starting with index c. As an example, let us compute $\tau(\mathsf{AGEX}\mathbf{true}, 0)$:

$$\begin{aligned}
&\tau(\mathsf{AGEX}\mathbf{true}, 0)\\
&= \tau(\neg\mathsf{EF}\neg(\mathsf{EX}\mathbf{true}), 0)\\
&= \tau(\neg\mathsf{E}(\mathbf{true}\mathsf{U}(\neg(\mathsf{EX}\mathbf{true}))), 0)\\
&= \neg\tau(\mathsf{E}(\mathbf{true}\mathsf{U}(\neg(\mathsf{EX}\mathbf{true}))), 0)\\
&= \neg(\mu X_0.\tau(\neg(\mathsf{EX}\mathbf{true}), 1) \vee (\mathbf{true} \wedge \Diamond X_0))\\
&= \neg(\mu X_0.\neg\tau(\mathsf{EX}\mathbf{true}, 1) \vee \Diamond X_0)\\
&= \neg(\mu X_0.\neg(\Diamond\mathbf{true}) \vee \Diamond X_0).
\end{aligned}$$

Notice that $\neg(\mu X_0.\neg(\Diamond\mathbf{true}) \vee \Diamond X_0)$ is equivalent to $\nu X_0.\Diamond\mathbf{true} \wedge \Box X_0$ as desired.

Let $K = (S, \rightarrow, s_0, L)$ be a Kripke structure and α a CTL formula, it is easy to show that $K, s_0 \models \alpha$ if and only if $K, s_0 \models \tau(\alpha, 0)$. We define the module `CTL2MU` based on the translation function τ to help the user to write

CTL specifications.[4] For instance, the deadlock freedom specification

```
eq prop3 = empty init |- Nu X (<> True /\ [] X) .
```

can also be written as

```
eq prop3 = empty init |- tau (AG EX True, 0) .
```

An equational theory to translate CTL formulae is presented in this section. Since the translation of CTL does not require mutual fixed point operations, our solution simply ignores all propositional variables used previously. Note that translations of fair CTL [9] and LTL require mutual fixed points. Currently, we do not know how to perform LTL translation in rewriting logic.

6 Future Work and Conclusion

The present work demonstrates the expressive power of rewriting logic. It shows that Maude can be used as a general framework for model specification, property specification, and model checking algorithm implementation. However, our work is by no mean complete. In order to compete with formal verification tools, many issues still need be improved.

Firstly, an LTL translation would be very useful. Since CTL and LTL are incomparable in terms of expressive power, an LTL translator can help users to specify more properties in practice. In [7], a translation from CTL* to μ-calculus is proposed. However, it is unclear how to implement the translation in rewriting logic.

Secondly, we would like to explore the possibility of verifying infinite-state systems. By infinite-state systems, we mean the number of reachable states is infinite. Since both explicit- and implicit-state algorithms require the model to be finite, it is necessary to reduce any infinite-state system to a finite one. For traditional model checking algorithms, infinite-state verification cannot be done without proper abstraction. Local model checking, on the other hand, does not restrict to finite models. It is possible to prove properties of infinite systems locally. Of course, the choice of successors plays an important role in this context. Fortunately, the user can use meta-level theory to try different strategies. Since model checking infinite-state systems is undecidable in general, we cannot hope a single strategy to solve the problem entirely. However, heuristics can be developed and tried on the infinite system first. The verifier may perform the abstraction after heuristics fail.

Lastly, it would be very interesting to combine the present work with those in [11]. The efficiency of the model checker in [11] is comparable to SPIN [12]. However, the user has less control over the model checking algorithm. Once

[4] Due to the lack of space, the listing is omitted.

it gets started, the user can only wait for the result. On the other hand, the efficiency of our model checker is somewhat disappointing. Developing a hybrid approach would be very useful to real world applications.

Acknowledgment

The author would like to thank anonymous reviewers for their valuable suggestions and comments.

References

[1] Henrik Reif Andersen, Colin Stirling, and Glynn Winskel. A compositional proof system for the modal μ-calculus. In *Proceedings, Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 144–153, Paris, France, 4–7 July 1994. IEEE Computer Society Press.

[2] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, and L. J. Hwang. Symbolic model checking: 10^{20} states and beyond. *Information and Computation*, 98(2):142–170, June 1992.

[3] David Basin, Manuel Clavel, and José Meseguer. Rewriting logic as a metalogical framework. *Lecture Notes in Computer Science*, 1974:55–80, 2000.

[4] R. Brayton, G. Hachtel, A. Sangiovanni-Vincentell, F. Somenzi, A. Aziz, S. Cheng, S. Edwards, S. Khatri, Y. Kukimoto, A. Pardo, S. Qadeer, R. Ranjan, S. Sarwary, T. Shiple, G. Swamy, and T. Villa. VIS: A system for verification and synthesis. In *Proceedings of the Eighth International Conference on Computer Aided Verification*, LNCS 1102, pages 428–432. Springer-Verlag, 1996.

[5] Manuel Clavel, Francisco Durán, Steven Eker, Patrick Lincoln, Narciso Martí-Oliet, José Meseguer, and Carolyn Talcott. *Maude 2.0 Manuel*, version 1.0 edition, June 2003.

[6] Rance Cleaveland. Tableau-based model checking in the propositional mu-calculus. *Acta Informatica*, 27(8):725–747, 1989.

[7] Mads Dam. CTL$\star$ and ECTL$\star$ as fragments of the modal μ-calculus. *Theoretical Computer Science*, 126(1):77–96, 1994.

[8] E.A. Emerson and E.M. Clarke. Using branching-time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[9] E.A. Emerson and C.L. Lei. Modalities for model-checking: Branching time logic strikes back. In *Proceedings of the 12th ACM Symposium on Principles of Programming Languages*, pages 84–96, 1985.

[10] E. Allen Emerson and Chin-Laung Lei. Efficient model-checking in fragments of the propositional mu-calculus. In *Proceedings 1st Annual IEEE Symp. on Logic in Computer Science, LICS'86, Cambridge, MA, USA, 16–18 June 1986*, pages 267–278. IEEE Computer Society Press, Los Alamitos, CA, 1986.

[11] Steven Eker, José Meseguer, and Ambarish Sridharanarayanan. 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, 2002.

[12] G.J. Holzmann. The model checker SPIN. *IEEE Trans. on Software Engineering*, 23(5):279–295, 1997.

[13] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[14] K. L. McMillan. The SMV system, symbolic model checking - an approach. Technical Report CMU-CS-92-131, Carnegie Mellon University, 1992.

[15] K. McMillan. *Symbolic model checking: an approach to the state explosion problem.* Kluwer Academic Publishers, 1993.

[16] J. Meseguer. Conditional rewriting logic as a unified model of concurrency. *Theoretical Computer Science*, 96(1):73–155, April 1992.

[17] José Meseguer. Rewriting logic as a semantic framework for concurrency: A progress report. In Ugo Montanari and Vladimiro Sassone, editors, *CONCUR '96: Concurrency Theory, 7th International Conference*, volume 1119 of *Lecture Notes in Computer Science*, pages 331–372, Pisa, Italy, 26–29 August 1996. Springer-Verlag.

[18] José Meseguer. Rewriting logic and Maude: Concepts and applications. In L. Bachmair, editor, *Rewriting Techniques and Applications, 11th International Conference, RTA 2000, Norwich, UK, July 10–12, 2000, Proceedings*, volume 1833 of *Lecture Notes in Computer Science*, pages 1–26. Springer-Verlag, 2000.

[19] Narciso Martí-Oliet and José Meseguer. Rewriting logic: roadmap and bibliography. *Theoretical Computer Science*, 285(2):121–154, August 2002.

[20] Colin Stirling and David Walker. Local model checking in the modal mu-calculus. In J. Díaz and F. Orejas, editors, *Proceedings Int. Joint Conf. on Theory and Practice of Software Development, TAPSOFT'89, Barcelona, Spain, 13–17 March 1989, Volume 1*, volume 351 of *Lecture Notes in Computer Science*, pages 369–383. Springer-Verlag, Berlin, 1989.

[21] Alberto Verdejo and Narciso Martí-Oliet. Executing and verifying CCS in Maude. Technical report, October 2002. Submitted for publication.

[22] Winskel. A note on model checking the model nu-calculus. *TCS: Theoretical Computer Science*, 83:157–167, 1991.

[23] Bow-Yaw Wang, José Meseguer, and Carl A. Gunter. Specification and formal analysis of a PLAN algorithm in Maude. In Pao-Ann Hsiung, editor, *Proceedings International Workshop on Distributed System Validation and Verification, Taipei, Taiwan*, pages 49–56, April 2000.