中央研究院
資訊科學研究所
Institute of Information Science, Academia Sinica • Taipei, Taiwan, ROC

TR-IIS-05-009

Automatic Verification of a Model Checker in Rewriting Logic

Bow-Yaw Wang

July 2005 || Technical Report No. TR-IIS-05-009

http://www.iis.sinica.edu.tw/LIB/TechReport/tr2005/tr05.html

Automatic Verification of a Model Checker in Rewriting Logic

Bow-Yaw Wang

Institute of Information Science
Academia Sinica
Taipei, Taiwan

Abstract. In this paper, we use the reflection of rewriting logic to analyze a bounded local model checker for infinite-state systems formally. We introduce three-valued logic in a local model checking algorithm to formalize aborted verification. To improve its efficiency, several optimizations are introduced in the algorithm. We show how to exploit the reflection of rewriting logic and model check our bounded local model checker in rewriting logic formally.

1 Introduction

Developing model checkers is a very complicated process. It requires sophisticated programming and algorithm-developing skills. But the complexity of a typical model checker does not allow developers to analyze the tool easily. One may wonder if there should be flaws in the design or implementation of these tools. If so, what strategies can be taken to detect the flaws in the development process. A naïve proposal is to perform computer-aided analysis on the model checkers themselves. If a model checker could be formally analyzed by computers, it would increase our confidence in the tool. Even though the current state of technology does not allow us to analyze a real-world model checker as is, it should be straightforward to verify high-level design in principle. Or is it?

Suppose we would like to verify a model checker by certain formal verification tool. We have to formalize the model checker in the verification tool first. Since the model checker takes a system and a property as inputs, we need specifications of the system description language and the property description language in the formalization. Additionally, system behavior and property semantics are formalized. The formalization of system behavior is then used in the formalization of the model checker. To analyze the model checker, we can show that the model checker yields results consistent with the property semantics in our formalization. If one considers variants of computation models and property specification logics, it is easy to see that a high-level formalization of model checker is by no mean a simple task. Unlike network protocols or hardware circuits, the obstacle in the analysis of model checkers arises even in their formal specifications.

The problem, in our view, is resulted from the lack of theoretical framework for developing model checkers. In the formalization of a typical model

checker, the syntax and semantics of property and system description languages are needed. The formalization of the model checker adopts the semantics of system description language and computes the result. Observe that system behavior and the computation of the model checker are different yet closely related. An ideal framework should be capable of unifying these components and distinguishing them at the same time. Although there have been formalisms for model and property specifications, there is little discussion about the formalism of the computation of model checker. As a result, formal analysis of model checkers is very complicated and hard to reproduce.

In this work, we use rewriting logic [1] as the formalism to verify a working model checker for infinite-state systems. Following the framework proposed in [2–4], we use rewrite theories as formal specifications of systems under verification. Properties are specified in a specification language defined by an equational theory. We develop a model checking algorithm for infinite-state systems and specify it in rewriting logic. Thanks to the reflection of rewriting logic, the simulation of model specification can be formalized by the universal rewrite theory. We are able to write down a formal specification of the new algorithm and execute it on Maude [5]. We verify the Bakery algorithm by our formally specified model checker as an example. Our model checker always terminates and reports conclusive verification results when properties can be proved or disproved locally. In comparison, the LTL model checker may not terminate in general [3]. The new model checker performs significantly better than those reported in [4] as well.

Although our extension and optimization in the model checking algorithm are intuitive, we have not provided any proof of correctness yet. In particular, we do not know if our modification may yield different results with different computation paths. Users would be surprised to have a proved property turning into abort by changing the order of rewriting. With the formal specification of our new model checking algorithm, we are able to perform its formal analysis in the same framework. Our key idea is to exploit the reflection of rewriting logic again. The specification of our model checking algorithm is nothing but another system under verification. Since there are finite number of "states" in our model checking algorithm, we can use any model checker for finite-state systems to formally verify our new algorithm. Specifically, the Maude LTL model checker proves that our new algorithm indeed yields unique result for the exemplary properties on the Bakery algorithm.

Model checking algorithms have been formally verified by proof assistants [6, 7]. In these works, 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 necessary due to the undecidability of the meta logic theory. Here, a working model checker is verified by another model checker. It only proves that our new model checker is correct under a very specific scenario. We are interested in our extensions and optimizations of the algorithm. The full scale analysis in [6, 7] would be an overkill.

An LTL model checker is also available in more recent releases of Maude [3]. The performance of the built-in LTL model checker is reported to be comparable to the model checker SPIN [8]. It may be difficult for verification tool developers to modify and improve the internal model checker, however. Additionally, only finite-state systems can be verified due to the limitation of the underlying model checking algorithm.

The inconvenience is resolved in [4] where a proof-theoretic μ-calculus model checking algorithm [9–11] 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. As a result, the efficiency of the model checker is disappointing. We extend the work in [4] and improve the performance significantly. Furthermore, checking infinite-state systems is only mentioned briefly in [4]. We address the problem explicitly in this paper.

The paper is organized as follows. Section 2 provides necessary technical backgrounds. We review the framework in Section 3. Our μ-calculus model checker is presented in Section 4. It is followed by the verification of the Bakery algorithm in Section 5. We verify our μ-calculus model checker in Section 6. In Section 7, we discuss future work and conclude the paper.

2 Preliminaries

We use μ-calculus for property specification [12]. 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: $\neg\varphi$, $\varphi \wedge \varphi'$;
- 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 $\varphi \vee \varphi' (\equiv \neg(\neg\varphi \wedge \neg\varphi'))$, $[\bar{\ell}]\varphi (\equiv \neg\langle\bar{\ell}\rangle\neg\varphi)$ and $\mu X.\varphi (\equiv \neg\nu X.\neg\varphi[\neg X/X])$. Furthermore, we will write $\Diamond\varphi$ and $\Box\varphi$ when all transition labels are allowed.

The semantics of φ is defined over a *Kripke structure* $K = (S, L, \rightarrow, s_0, P)$ where S is the set of states, L the set of transition labels, $\rightarrow \subseteq S \times L \times S$ 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 $[\![\bullet]\!]\rho$ for a μ-calculus formula φ computes the set of states satisfying φ under the valuation ρ:

- $[\![X]\!]\rho = \rho(X)$;

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

Given a μ-calculus formula φ and a Kripke structure $K = (S, L, \rightarrow, s_0, P)$, we write $K, s \models \varphi$ when $s \in [\![\varphi]\!]\emptyset$. The μ-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, [13]). In tableau-based local model checking algorithms [9, 10], the model checking problem is solved by constructing proofs of the judgment $K, s \vdash \varphi$. The tableau-based algorithm was then simplified to a set of reduction rules in [11]. We need the following extension to the greatest fixed point operator, $\nu X\{\bar{r}\}\varphi$ where $\bar{r}$ is a set of states [11]:

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

Note that $\nu X\{\}\varphi \equiv \nu X.\varphi$. Any fixed-point operator can be translated to its extended form syntactically. The extension reduces the side condition of tableau-based algorithm to membership checking and allows the proof search to be performed by rewriting. Given a Kripke structure $K = (S, L, \rightarrow, s_0, P)$ and a μ-calculus formula φ, the following rules reduce $K, s \vdash \varphi$ to truth values true or false [11]:

- $(K, s \vdash p) = \mathsf{true}$ if $p \in P(s)$;
- $(K, s \vdash p) = \mathsf{false}$ if $p \notin P(s)$;
- $(K, s \vdash \mathsf{true}) = \mathsf{true}$;
- $(K, s \vdash \mathsf{false}) = \mathsf{false}$;
- $(K, s \vdash \neg\varphi) = \neg b$ where $(K, s \vdash \varphi) = b$;
- $(K, s \vdash \varphi \wedge \varphi') = b_0 \wedge b_1$ where $(K, s \vdash \varphi) = b_0$ and $(K, s \vdash \varphi') = b_1$;
- $(K, s \vdash \varphi \vee \varphi') = b_0 \vee b_1$ 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 $(K, s_0 \vdash \varphi) = \mathsf{true}$ if and only if $K, s \models \varphi$ [11].

3 Rewriting Logic as Verification Framework

We use rewriting logic [1] as the unified framework to formalize our model checker. Since its introduction in [1], rewriting logic has been used as a unified formalism for modeling concurrency [1, 14, 15] and as a logical framework [16]. It

is not hard to see that rewriting logic is capable of property and model specification [2–4]. As for the model checking algorithm, we rely on the reflection property of rewriting logic. In the following, we will briefly review rewriting logic and its verification framework as proposed in [3]. For detailed exposition, the reader is referred to [17, 18, 3].

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*.

In rewriting logic, 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.

Two terms are equivalent if they can be reduced to the same normal form by the equations of a rewrite theory. Equations therefore define equivalence classes of terms. For any term t, we write $[t]$ for its equivalence class. 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 presented 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. By the universal theory $\mathcal{U}$, we can manipulate meta-level terms at object level. We call the feature that can represent meta-level objects at object level as *reflection*.

We now describe the framework for model checking algorithm specifications used in [3, 4]. In the framework, the Kripke structure is specified as a rewrite theory $\mathcal{K}$. The states are equivalence classes of terms defined in $\mathcal{K}$. The transitions of the Kripke structure correspond to rewriting rules in $\mathcal{K}$. Since the Kripke structure is specified as a rewrite theory and system configurations as equivalence classes of terms, the universal theory $\mathcal{U}$ can be used to explore successors of the current system configuration. Meanwhile, μ-calculus formulae can be represented by terms in rewriting logic with proper function symbols. Along with the universal rewrite theory, the essential components of model checkers from specification language, model representation to model checking algorithm can be formalized under the rewriting logic framework.

4 A Bounded μ-Calculus Model Checker

Based on the framework described in Section 3, Winskel's reduction rules can be formalized as an equational theory [4]. Since the reduction rules are known to be sound and complete for finite-state models, one simply need search all (finite) proof trees exhaustively. The μ-calculus model checker in [4] is implemented with the naïve search strategy. However, this simple idea may not work for infinite-state models.

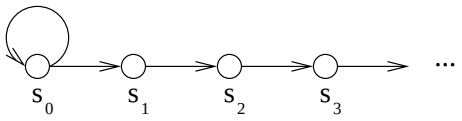

Fig. 1. An Infinite-State Model

Consider the infinite-state model in Figure 1, where the atomic proposition p holds universally. Suppose we would like to check whether the model satisfies $\nu X.p \wedge \Diamond X$. There are two possible derivations. One derivation chases the infinite sequence of states and is unable to conclude the proof; the other chooses the self-loop and concludes the proof in one step. As can be seen from the example, different strategies for choosing successors have impact on the effectiveness of the model checker. One may wonder if a universal strategy which works for all models may exist. However, since it is easy to reduce Post's Correspondence Problem to an instance of μ-calculus model checking problem in rewriting logic, such a universal strategy does not exist. One can only hope for heuristics in practice.

For systems with infinite sequence of successive states, the depth-first search strategy may not be effective (as in the example). Similarly, the breadth-first search strategy may not be effective for infinite-branching systems. Our main idea is thus to perform the search within certain bounds of depth and width. If a complete proof can be found within these bounds, the model checker reports the result; otherwise, it reports abort.

Figure 2 defines the function symbols for the three-valued logic that we are using. The sort LMCResult specifies the result of model checker. The subsort declaration Bool $\prec$ LMCResult specifies that Bool is a subsort of LMCResult. Therefore both false and true are of sort LMCResult as well. The constant symbol abort of sort LMCResult is also defined. It represents the situation when the model checker cannot conclude the verification. The function symbols !, &&, and || extend Boolean operators $\neg$, $\wedge$, and $\vee$ respectively. The underlines (_) in the declarations denotes the positions of the parameters. The keywords **comm** and **assoc** declare that the function symbols && and || are commutative and associative. The intuition is to exploit dominating values in Boolean disjunction. For instance, the result of the disjunction can be determined if one of the operands is known to be true, regardless of the value of the other operand.

sort LMCResult
subsort Bool $\prec$ LMCResult
op abort : $\rightharpoonup$ LMCResult

op !_ : LMCResult $\rightharpoonup$ LMCResult
op _&&_ : LMCResult LMCResult $\rightharpoonup$ LMCResult [**comm assoc**]
op _||_ : LMCResult LMCResult $\rightharpoonup$ LMCResult [**comm assoc**]

eq ! abort = abort	**eq** ! $b = \neg\, b$
eq false && a = false	**eq** false $\|\|$ $a = a$
eq true && $a = a$	**eq** true $\|\|$ a = true
eq abort && abort = abort	**eq** abort $\|\|$ abort = abort

Fig. 2. Equational Rules for Three-Valued Logic

We specify Winskel's reduction rules as a set of rewriting rules for entailment terms.[1] To define entailment terms, we first define the entailment $\vdash$ as a function symbol. Let $\mathcal{K}$ be a rewrite theory, L a set of labels, $\underline{s}$ a meta-level term representing a state, d and w two natural numbers, and φ a μ-calculus formula. The entailment term $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi$ is of sort LMCResult. The idea is to provide a set of rules to rewrite the entailment term to false, true, or abort for any Kripke structure specified by the rewrite theory $\mathcal{K}$. It is crucial to use a meta-level term $\underline{s}$ for the state representation. The transitions of the Kripke structure $\mathcal{K}$ would rewrite the state s had we used the object-level term s in the entailment term.

With the definition of entailment term in place, we can now define the reduction rules for our μ-calculus model checker. The following rules are used for Boolean constant and negation.

rl $[tt] : \mathcal{K}\ L\ \underline{s}\ d\ w \vdash \mathsf{true} \Rightarrow \mathsf{true}$
rl $[neg] : \mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg\, p \Rightarrow\ !\ (\mathcal{K}\ L\ \underline{s}\ d\ w \vdash p)$

The rule tt rewrites the entailment term $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \mathsf{true}$ to true. For atomic propositions p, the user needs to provide his or her own rewriting rules to determine whether p holds in the state s. For atomic propositions of the form $\neg p$, the model checker first rewrites $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash p$, then uses the three-valued negation (!) to get the final result. Since the entailment term may rewrite to abort, the Boolean operators do not consider the inconclusive value and would incur irreducible terms. Hence it is incorrect to use the Boolean negation. Similarly, the conjunction and disjunction are achieved by invoking the corresponding operators in three-valued logic.

rl $[conj] : \mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi \wedge \varphi' \Rightarrow (\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi)$ && $(\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi')$
rl $[negconj] : \mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg(\varphi \wedge \varphi') \Rightarrow (\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg\, \varphi)\ \|\ (\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg\, \varphi')$

The rules *conj* and *negconj* are very inefficient, though. To verify $\varphi \wedge \varphi'$, for instance, the model checker first verifies both φ and φ'. It then uses the results in rule *conj*. If φ is not satisfied, it is easy to see that the proof search of φ' is redundant. We therefore use the following optimized rule instead:

[1] Equational theory would suffice for the specification, but we need rule labels for its formal verification.

rl [*conj*] : $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi \wedge \varphi' \Rightarrow$ and-wrapper $(\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi, \mathcal{K}, L, \underline{s}, d, w, \varphi')$
eq and-wrapper (false, $\mathcal{K}$, L, $\underline{s}$, d, w, φ') = false
eq and-wrapper (true, $\mathcal{K}$, L, $\underline{s}$, d, w, φ') = $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi'$
eq and-wrapper (abort, $\mathcal{K}$, L, $\underline{s}$, d, w, φ') = abort && $(\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi')$

rl [*negconj*] : $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg\ (\varphi \wedge \varphi') \Rightarrow$
or-wrapper $(\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg\ \varphi, \mathcal{K}, L, \underline{s}, d, w, \neg\ \varphi')$
eq or-wrapper (false, $\mathcal{K}$, L, $\underline{s}$, d, w, φ') = $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi'$
eq or-wrapper (true, $\mathcal{K}$, L, $\underline{s}$, d, w, φ') = true
eq or-wrapper (abort, $\mathcal{K}$, L, $\underline{s}$, d, w, φ') = abort && $(\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi')$

Fig. 3. Optimized *negconj*

The trick is to postpone the proof search of φ' by not forming the entailment term $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi'$. Along with the proof search result of the term $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi$, parameters in the term $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \varphi'$ are passed to the function and-wrapper instead. The wrapper function then performs the proof search of φ' if necessary. The same trick is used for the rule *negconj* as well (Figure 3). Another option to delay the evaluation of entailment terms is by user-defined strategies [5]. But we find auxiliary functions are more efficient in this case and use them here.

We now give the definitions of depth and width bounds of a proof. The *depth* of a proof is the maximal number of unrolling applied to a fixed point operator. The *width* of a proof is defined by the maximal number of successors explored by each $\langle\bar{\ell}\rangle$- and $[\bar{\ell}]$-rules. For the modal temporal operator $\langle\bar{\ell}\rangle$, we have the following rules:

rl [*ex*] : $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \Diamond\ \varphi \Rightarrow$ exists $(\mathcal{K}, L, \underline{s}, \varphi, L, 0, d, w, w)$
rl [*negex*] : $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg\ \Diamond\ \varphi \Rightarrow$! exists $(\mathcal{K}, L, \underline{s}, \varphi, L, 0, d, w, w)$

rl [*exx*] : $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \langle L'\rangle\ \varphi \Rightarrow$ exists $(\mathcal{K}, L, \underline{s}, \varphi, L', 0, d, w, w)$
rl [*negexx*] : $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash \neg\ \langle L'\rangle\ \varphi \Rightarrow$! exists $(\mathcal{K}, L, \underline{s}, \varphi, L', 0, d, w, w)$

The function exists $(\mathcal{K}, L, \underline{s}, \varphi, L', n, d, w, W)$ checks if there exists a proof of φ within depth d and width w at an L'-successor where n and W serve as counters. It applies the rewriting rules on φ recursively and keeps the width of proof search within the bound. If the search exceeds the bound, it reports abort:

eq exists $(\mathcal{K}, L, \underline{s}, \varphi, L', n, d, w, 0)$ = abort

If the proof search bound is not exceeded but there is no transition label, it degenerates to false:

eq exists $(\mathcal{K}, L, \underline{s}, \varphi, \emptyset, n, d, w,$ s $W)$ = false

where the symbol s is the successor symbol defined in the natural number theory. Hence the pattern (s W) matches any natural number greater than zero. Notice the semantics differ from those in [3]. Our semantics do not have implicit self-loops, which is quite common in branching-time temporal logics.

If the bound is exceeded and there are transition labels, the function exists $(\mathcal{K}, L, \underline{s}, \varphi, L', n, d, w,$ s $W)$ uses the universal theory to find successors of s in the rewrite theory $\mathcal{K}$. The reduction rule checks whether there is a proof of

the subformula φ from the successor within depth d and width w, or a proof for the next successor within depth d and width W recursively. On the other hand, if there is no successor of the current label, it looks for a successor of the next label.

eq exists $(\mathcal{K},\ L,\ \underline{s},\ \varphi,\ l'\ L',\ n,\ d,\ w,\ \mathsf{s}\ W) =$
  if $\mathcal{U} \vdash_{l',n} [\underline{\mathcal{K}},\ \underline{s}] \rightarrow [\underline{\mathcal{K}},\ \underline{t}]$ **then**
    $(\mathcal{K}\ L\ \underline{t}\ d\ w \vdash \varphi)\ \|\ (\mathsf{exists}\ (\mathcal{K},\ L,\ \underline{s},\ \varphi,\ l'\ L',\ n+1,\ d,\ w,\ W))$
  else
    exists $(\mathcal{K},\ L,\ \underline{s},\ \varphi,\ L',\ 0,\ d,\ w,\ \mathsf{s}\ W)$
  fi

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 allows us to specify the algorithm clearly. Similar to the rules *conj* and *negconj*, the three-valued disjunction in the function exists can be optimized as well (Figure 4).

eq exists-wrapper (false, $\mathcal{K}$, L, $\underline{s}$, φ, L', n, d, w, W) =
  exists $(\mathcal{K},\ L,\ \underline{s},\ \varphi,\ L',\ n,\ d,\ w,\ W)$
eq exists-wrapper (true, $\mathcal{K}$, L, $\underline{s}$, φ, L', n, d, w, W) = true
eq exists-wrapper (abort, $\mathcal{K}$, L, $\underline{s}$, φ, L', n, d, w, W) =
  abort $\|$ exists $(\mathcal{K},\ L,\ \underline{s},\ \varphi,\ L',\ n,\ d,\ w,\ W)$
eq exists $(\mathcal{K},\ L,\ \underline{s},\ \varphi,\ l'L',\ n,\ d,\ w,\ \mathsf{s}\ W) =$
  if $\mathcal{U} \vdash_{l',n} [\underline{\mathcal{K}},\underline{s}] \rightarrow [\underline{\mathcal{K}},\underline{t}]$ **then**
    exists-wrapper $(\mathcal{K}\ L\ \underline{t}\ d\ w \vdash \varphi,\ \mathcal{K},\ L,\ \underline{s},\ \varphi,\ l'L',\ n+1,\ d,\ w,\ W)$
  else
    exists $(\mathcal{K},\ L,\ \underline{s},\ \varphi,\ L',\ n,\ d,\ w,\ \mathsf{s}\ W)$
  fi

Fig. 4. Optimized *exists*

For the fixed-point operators, we need a term to represent the formula $\sigma X\{\bar{r}\}\varphi$ where σ is either of the fixed-point operators [11]. There are several choices for the representation of the state set $\{\bar{r}\}$. We could define an equational theory for state sets and use it in the representation of the fixed-point formula. This approach would, however, induce unintended rewrites in the μ-calculus formulae by rewriting rules of the Kripke structure. To avoid unintended rewriting, we use a meta-level term set S instead. A term of the form $\sigma XS\varphi$ represents the formula $\sigma X\{\bar{r}\}\varphi$ when $S = \{\underline{\bar{r}}\}$. Our μ-calculus theory depends on the user-defined meta-level term set theory consequently.

To formally define the rules for fixed-point formulae, we need the substitution function. For μ-calculus formulae φ, ϵ and the proposition variable X, the substitution function subst $(\varphi,\ X,\ \epsilon)$ returns the formula $\varphi[\epsilon/X]$ (Figure 5). It is straightforward to add bound checking to Winskel's rules:

rl $[nu] : \mathcal{K}\ L\ \underline{s}\ 0\ w \vdash \nu XS\varphi \Rightarrow$ abort
rl $[nu] : \mathcal{K}\ L\ \underline{s}\ (\mathsf{s}\ d)\ w \vdash \nu XS\varphi \Rightarrow$
  if $\underline{s} \in S$ **then** true **else** $\mathcal{K}\ L\ \underline{s}\ d\ w \vdash$ subst $(\varphi,\ X,\ \nu X(\underline{s} \cup S)\varphi)$ **fi**

eq subst (true, Z, ϵ) = true
eq subst (false, Z, ϵ) = false
ceq subst (X, Z, ϵ) = X **if** $X \neq Z$
ceq subst (X, Z, ϵ) = ϵ **if** $X = Z$
eq subst (p, Z, ϵ) = p
eq subst ($\neg\varphi$, Z, ϵ) = $\neg$ subst (φ, Z, ϵ)
eq subst ($\varphi \wedge \psi$, Z, ϵ) = subst (φ, Z, ϵ) $\wedge$ subst (ψ, Z, ϵ)
eq subst ($\varphi \vee \psi$), Z, ϵ) = subst (φ, Z, ϵ) $\vee$ subst (ψ, Z, ϵ)
eq subst ($\Diamond\varphi$, Z, ϵ) = $\Diamond$ subst (φ, Z, ϵ)
eq subst ($\langle\bar{\ell}\rangle\varphi$, Z, ϵ) = $\langle\bar{\ell}\rangle$ subst (φ, Z, ϵ)
eq subst ($\Box\varphi$, Z, ϵ) = $\Box$ subst (φ, Z, ϵ)
eq subst ($[\ell]\varphi$, Z, ϵ) = $[\ell]$ subst (φ, Z, ϵ)
ceq subst ($\mu\ X\ S\ \varphi$, Z, ϵ) = $\mu\ X\ S$ (subst (φ, Z, ϵ)) **if** $X \neq Z$
ceq subst ($\mu\ X\ S\ \varphi$, Z, ϵ) = $\mu\ X\ S\ \varphi$ **if** $X = Z$
ceq subst ($\nu\ X\ S\ \varphi$, Z, ϵ) = $\nu\ X\ S$ (subst (φ, Z, ϵ)) **if** $X \neq Z$
ceq subst ($\nu\ X\ S\ \varphi$, Z, ϵ) = $\nu\ X\ S\ \varphi$ **if** $X = Z$

Fig. 5. Definition of *subst*

rl [*choosing*] : $\lhd\ i$, choose, $n \rhd \lhd\ i'$, M', $n' \rhd \Rightarrow$
$\lhd\ i$, wait-choose $((i+1)\ \%\ 2)$, max$(n, n') + 1 \rhd \lhd\ i'$, M', $n' \rhd$
crl [*waiting*] : $\lhd\ i$, wait-choose (i'), $n \rhd \lhd\ i'$, M', $n' \rhd \Rightarrow$
$\lhd\ i$, wait-turn (i'), $n \rhd \lhd\ i'$, M', $n' \rhd$
if $M' \neq$ choose
crl [*waiting*] : $\lhd\ i$, wait-turn (i'), $n \rhd \lhd\ i'$, M', $n' \rhd \Rightarrow$
$\lhd\ i$, wait-choose $((i'+1)\ \%\ 2)$, $n \rhd \lhd\ i'$, M', $n' \rhd$
if $((i'+1)\ \%\ 2) \neq i \wedge \neg(n' \neq 0 \wedge (n' < n \vee (n' = n \wedge i' < i)))$
crl [*entering*] : $\lhd\ i$, wait-turn (i'), $n \rhd \lhd\ i'$, M', $n' \rhd \Rightarrow$
$\lhd\ i$, critical, $n \rhd \lhd\ i'$, M', $n' \rhd$
if $((i'+1)\ \%\ 2) = i \wedge \neg(n' \neq 0 \wedge (n' < n \vee (n' = n \wedge i' < i)))$
rl [*leaving*] : $\lhd\ i$, critical, $n \rhd \Rightarrow \lhd\ i$, choose, $0 \rhd$

Fig. 6. Bakery Algorithm

The first *nu* rule rewrites $\nu XS\varphi$ to abort if the depth bound is exceeded. Otherwise, the second *nu* rule checks whether the current state $\underline{s}$ has been visited or not. If so, it rewrites the entailment term to true. Otherwise, the current state is added to the set S and the new set is used in the unfolding of the fixed-point formula. Note that the set membership and union in the rules are defined over meta-level terms.

5 Verification of the Bakery Algorithm

We use our bounded local model checker to verify the Bakery algorithm with two processes as an example. The mutual exclusion algorithm is finitely branching but admits infinite number of distinct states along computation paths. Figure 6 shows the rewrite theory for the Bakery algorithm.

Each process in our model is represented by the triple $\lhd id, mode, ticket \rhd$. The natural number *id* specifies the process identifier. There are several different *modes* in the model: choose, wait-choose (i), wait-turn (i), and critical. The *ticket* field is the ticket number owned by the process.

In the choose mode, the process computes its ticket number by incrementing the maximal ticket number of all processes. After choosing its ticket number, the process i enters the mode wait-choose $((i+1)\ \%\ 2)$. This is specified by the rules *choosing*. The process compares its ticket number with other processes by the *waiting* rules. The comparison begins at the next process and iterates through all processes in a round-ribbon manner. For process i', the current process i enters the mode wait-turn (i') after process i' leaves the mode choose. In the mode wait-turn (i'), process i compares its ticket number with process i''s. If process i has higher priority, it moves to the mode wait-choose $((i'+1)\ \%\ 2)$ and compares with the next process's ticket number. However, if there is no more ticket number to be compared, process i enters the mode critical by rule *entering*. When leaving the mode critical, the process resets its ticket number back to zero and goes back to mode choose (rule *leaving*). Since the ticket number may be incremented indefinitely, the number of states is infinite.

The atomic proposition CS (i) is defined as follows.

$$\textbf{rl}\ [AP] : \mathcal{K}\ L\ \underline{s}\ d\ w \vdash \mathsf{CS}\ (i) \Rightarrow \text{in-critical}\ (s,\ i)$$
$$\textbf{eq}\ \text{in-critical}\ (\lhd\ 0,\ M_0,\ n_0\ \rhd\ C,\ 0) = (M_0 = \text{critical})$$
$$\textbf{eq}\ \text{in-critical}\ (\lhd\ 1,\ M_1,\ n_1\ \rhd\ C,\ 1) = (M_1 = \text{critical})$$

The proposition CS (i) holds at the state s if the process i of state s is in the mode critical. For convenience, we define labels and init as follows.

$$\textbf{eq}\ \text{labels} = \textit{choosing waiting entering leaving}$$
$$\textbf{eq}\ \text{init} = \lhd\ 0,\ \text{choose},\ 0\ \rhd\ \lhd\ 1,\ \text{choose},\ 0\ \rhd$$

Let $\mathcal{B}$ be the rewrite theory for the Bakery algorithm. We can check whether process 0 will enter the critical mode inevitably. This can be formulated as the following entailment term:

$$\textbf{eq}\ \text{prop0} = \mathcal{B}\ \text{labels}\ \underline{\text{init}}\ 10\ 3 \vdash \mu\ X\ \emptyset\ (\mathsf{CS}\ (0) \vee \Box\ X)$$

Secondly, we would like to know if process 1 does not enter the critical section until process 0 does. That is, process 0 always enters the critical section first. The property is specified by

$$\textbf{eq}\ \text{prop1} = \mathcal{B}\ \text{labels}\ \underline{\text{init}}\ 5\ 3 \vdash \mu\ X\ \emptyset\ (\mathsf{CS}\ (0) \vee (\neg\ \mathsf{CS}\ (1) \wedge \Box\ X))$$

Finally, we would like to check if process 0 can enter the critical section infinitely often:

$$\textbf{eq}\ \text{prop2} = \mathcal{B}\ \text{labels}\ \underline{\text{init}}\ 3\ 3 \vdash \nu\ X\ \emptyset\ \mu\ Y\ \emptyset\ \Diamond\ ((\mathsf{CS}\ (0) \wedge X) \vee Y)$$

We conduct our experiments in Maude [5]. Maude is a rewriting system based on rewriting logic. The property, model and algorithm specifications presented in this paper are translated to Maude modules. The entailment terms prop0, prop1, and prop2 rewrite to true, false, and abort in Maude respectively. Moreover, the results are obtained by performing 3698, 1495, and 275 rewrites respectively. Maude is able to produce the results almost immediately on a 2.8GHz Pentium 4 Linux computer.

In comparison, the built-in LTL model checker in Maude can only prove that process 0 will enter the critical section inevitably (prop0). It fails to terminate on

the remaining properties. For prop0, the Bakery algorithm lets process 0 enter the critical section inevitably. The global model checking algorithm implemented in Maude cannot find any counterexample by its default exploration strategy. It therefore concludes the verification successfully. For prop1, the default strategy in Maude LTL model checker ends up exploring the infinite state space. But our bounded local model checker gives up the infinite computation paths and successfully disproves the property by a finite path. For prop2, there is no finite path to verify the property. The Maude LTL model checker will explore the infinite state space surely. But our model checker stops after the bounds are exceeded.

6 Verification of Model Checking Algorithm

The verification of Bakery algorithm only shows that our infinite-state model checking algorithm may rewrite entailment terms to one of false, true, or abort. But we have yet to provide a formal analysis of our extension and optimization to Winskel's rewriting rules. In particular, if our algorithm would yield contradictory results by different rewriting sequences, users would be very confused.

The advocated theoretical framework solves the problem nicely. Formally, it only takes another application of reflection. Entailment terms and the model checking rewrite theory are lifted to the meta level. The behavior of our model checker can be explored by the universal theory $\mathcal{U}$. Hence, another model checker can verify our model checker at meta level formally. Notice that our model checker can generate only but finitely many LMCResult terms for each entailment term.

We begin with the meta-level entailment. Let $\mathcal{M}$ be our model checking theory, L the rule labels, e the entailment term, and φ a μ-calculus formula. Define the meta-level entailment term $\mathcal{M}\ L\ \underline{e} \Vdash \varphi$ of sort Bool. It is easy to define the equational rules for the meta-level atomic propositions is-false, is-true, and is-abort:

$$\begin{aligned}
&\mathbf{eq} : \mathcal{M}\ L\ \underline{e} \Vdash \text{is-false} = (\underline{e} = \underline{\text{false}})\\
&\mathbf{eq} : \mathcal{M}\ L\ \underline{e} \Vdash \text{is-true} = (\underline{e} = \underline{\text{true}})\\
&\mathbf{eq} : \mathcal{M}\ L\ \underline{e} \Vdash \text{is-abort} = (\underline{e} = \underline{\text{abort}})
\end{aligned}$$

They check whether the current entailment term corresponds to the LMCResult false, true, and abort respectively. Define lmc-labels to be:

$$\mathbf{eq}\ \text{lmc-labels} = \mathit{AP\ tt\ neg\ conj\ negconj\ ex\ negex\ exx\ negexx\ mu\ negmu\ nu\ negnu}$$

It is straightforward to specify the original local model checking algorithm (Figure 7). However, specifying properties of $\mathcal{M}$ in branching-time temporal logics exposes a subtle semantic issue. If our model checker always rewrites entailment terms to one of false, true, or abort after a number of rewrites, entailment terms will have no successors. Subsequently, the property $\Box\ \varphi$ will be true for all φ eventually. In particular, the entailment prop2 will be irreducible inevitably. It therefore satisfies $\mu\ X\ \emptyset$ is-true $\vee \Box\ X$, even though prop2 does not rewrite to true after a number of unrolling.

Our solution is to add implicit self-loops to irreducible terms. This can be done by modifying the meta-exists function:

```
eq M L e ⊨ true = true
eq M L e ⊨ ¬ p = ¬ (M L e ⊨ p)
eq M L e ⊨ φ ∧ φ' = (M L e ⊨ φ) and-also (M L e ⊨ φ')
eq M L e ⊨ ¬ (φ ∧ φ') = (M L e ⊨ ¬ φ) or-else (M L e ⊨ ¬ φ')
eq M L e ⊨ ◇ φ = meta-exists (M, L, e, φ, L, 0, true)
eq M L e ⊨ ¬ (◇ φ) = ¬ meta-exists (M, L, e, φ, L, 0, true)
eq M L e ⊨ ⟨ L' ⟩ φ = meta-exists (M, L, e, φ, L', 0, true)
eq M L e ⊨ ¬ (⟨ L' ⟩ φ) = ¬ meta-exists (M, L, e, φ, L', 0, true)
ceq M L e ⊨ μ X E φ =
   if e ∈ E then false else M L e ⊨ subst (φ, X, μ X (e ∪ E) φ) fi
eq M L e ⊨ ¬ μ X E φ =
   if e ∈ E then true else M L e ⊨ subst (¬ φ, X, μ X (e ∪ E) φ) fi
eq M L e ⊨ ν X E φ =
   if e ∈ E then true else M L e ⊨ subst (φ, X, ν X (e ∪ E) φ) fi
eq M L e ⊨ ¬ ν X E φ =
   if e ∈ E then false else M L e ⊨ subst (¬ φ, X, ν X (e ∪ E) φ) fi
```

Fig. 7. Meta-Level Model Checker

```
eq meta-exists (M, L, e, φ, ∅, n, deadend) =
     if deadend then M L e ⊨ φ else false fi
eq meta-exists (M, L, e, φ, l' L', deadend) =
   if U ⊢_{l',n} [M, e] → [M, f] then
      (M L f ⊨ φ) ∨ meta-exists (M, L, e, l' L', n + 1, false)
   else
      meta-exists (M, L, e, φ, L', deadend)
   fi
```

If e is an irreducible term ($deadend$ = true), e is the only successor of itself. Otherwise, meta-exists works like exists but at the meta meta-level.

We can check whether our infinite-state model checking algorithm $\mathcal{M}$ rewrites the entailment term prop0, prop1, prop2 to true, false, abort inevitably by reducing the following three entailment terms respectively:

```
eq inevitably-true = M lmc-labels prop0 ⊨ μ X ∅ is-true ∨ □ X
eq inevitably-false = M lmc-labels prop1 ⊨ μ X ∅ is-false ∨ □ X
eq inevitably-abort = M lmc-labels prop2 ⊨ μ X ∅ is-abort ∨ □ X
```

We can therefore reduce the meta-level entailment term inevitably-abort. Maude reports true in 158,190 rewrites (about 32 seconds). However, we are unable to verify the other meta-level properties within a reasonably amount of time. We need a more efficient model checker for the task.

Recall that the bounds imposed on the proof search essentially forbid infinite rewrites on any entailment term, even for infinite-state models like the Bakery algorithm. Since entailment terms are the meta-level states, the Maude LTL model checker should not have difficulty in verifying properties on our model checker. Indeed, define the LTL atomic propositions on entailment terms:

```
eq e |= is-false = (e = true)
eq e |= is-true = (e = false)
eq e |= is-abort = (e = abort)
```

We can verify inevitably-true, inevitably-false, and inevitably-abort by reducing the following three terms:

eq eventually-true = modelCheck (prop0, ◊ is-true)
eq eventually-false = modelCheck (prop1, ◊ is-false)
eq eventually-abort = modelCheck (prop2, ◊ is-abort)

The Maude LTL model checker successfully proves eventually-true, eventually-false and eventually-abort in 300, 51 and 1 seconds respectively.

7 Conclusion and Future Work

In this paper, we introduce three-valued logic to model aborted computation in local model checking algorithms. The uncertain value allows the proof search to be redirected to other branches of the computation. We demonstrate how straightforward it is to formalize and adopt the idea in developing an infinite-state model checking algorithm. To illustrate the effectiveness of our new algorithm, we verify properties on the Bakery algorithm and compare it with the Maude LTL model checker. In our example, the new algorithm is able to verify local properties and terminate on all properties of interest. The global model checking algorithm implemented in Maude, on the other hand, proves only one property and fails to terminate on the others.

Using the same verification framework, we show how to verify properties on our new model checker by reflection. Although it is possible to perform formal verification by the original local model checker in theory, the meta-level model checker is unable to verify all the properties efficiently. The Maude LTL model checker can be deployed and successfully verifies that our new model checker indeed yields unique result.

Currently, we are interested in applying our technique in other model checking algorithms. Particularly, the analysis of binary decision diagram-based algorithms would be more useful to model checking community. We are investigating the theory developed in [19] and specifying a BDD-based algorithms in rewriting logic as the first step. Secondly, we would like to investigate on how to present counterexamples. At present, our model checker can only report confirmation, refutation and abort of the property. Reporting counterexamples of disproved properties to users would be very useful. Finally, we would like to improve the efficiency of our model checker further.

References

1. Meseguer, J.: Conditional rewriting logic as a unified model of concurrency. Theoretical Computer Science **96** (1992) 73–155
2. 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, Taipei, Taiwan. (2000) 49–56

3. 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 (2002)
4. Wang, B.Y.: μ-calculus model checking in maude. In: 5th International Workshop on Rewriting Logic and its Applications, Barcelona, Spain. March 27-28. (2004)
5. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude 2.0 Manuel. version 1.0 edn. (2003)
6. 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
7. Manolios, P. In: Mu-Calculus Model-Checking. Kluwer Academic Publishers (2000) 93–111
8. Holzmann, G.: The model checker SPIN. IEEE Trans. on Software Engineering **23** (1997) 279–295
9. Cleaveland, R.: Tableau-based model checking in the propositional mu-calculus. Acta Informatica **27** (1989) 725–747
10. 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, TAPSOFT'89, Barcelona, Spain, 13–17 March 1989. Volume 351 of LNCS. Springer-Verlag, Berlin (1989) 369–383
11. Winskel, G.: A note on model checking the modal nu-calculus. Theoretical Computer Science **83** (1991) 157–167
12. Kozen, D.: Results on the propositional μ-calculus. Theoretical Computer Science **27** (1983) 333–354
13. Clarke, E.M., Grumberg, O., Peled, D.A.: Model Checking. The MIT Press, Cambridge, Massachusetts (1999)
14. 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 Lecture Notes in Computer Science., Pisa, Italy, Springer-Verlag (1996) 331–372
15. Martí-Oliet, N., Meseguer, J.: Rewriting logic: roadmap and bibliography. Theoretical Computer Science **285** (2002) 121–154
16. Basin, D., Clavel, M., Meseguer, J.: Rewriting logic as a metalogical framework. Lecture Notes in Computer Science **1974** (2000) 55–80
17. Clavel, M., Meseguer, J.: Reflection and strategies in rewriting logic. In Meseguer, J., ed.: Proceedings First International Workshop on Rewriting Logic and its Applications, WRLA'96, Asilomar, California, September 3–6, 1996. Volume 4 of Electronic Notes in Theoretical Computer Science., Elsevier (1996) 125–147
18. 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, WRLA'98, Pont-à-Mousson, France, September 1–4, 1998. Volume 15 of Electronic Notes in Theoretical Computer Science., Elsevier (1998) 317–328
19. van de Pol, J.C., Zantema, H.: Binary decision diagrams by shared rewriting. In: 108. Centrum voor Wiskunde en Informatica (CWI), ISSN 1386-369X (2000) 14