Model Checking Using Tabled Rewriting*
(for IJCAR2004 Doctoral Programme)

Zhiyao Liang

Advisor: Rakesh M. Verma

Computer Science Department, The University of Houston, 501 PGH, Houston, Texas, 77204 – 3010, USA

{zliang,rmverma}@cs.uh.edu

Abstract. LRR [3] is a rewriting system developed at the Computer Science Department of University of Houston. LRR has two subsystems: Smaran (for tabled rewriting), and TGR (for untabled rewriting). It can utilize the history of computation to eliminate the redundant work in the process of reducing terms to their normalized forms. However the practicality of using LRR as a framework for implementing model checking has not been experimented before. We have implemented LTL and CTL model checking algorithms using LRR. The current result of this research shows that LRR can provide a convenient programming framework, and the model checker has already in some aspects achieved the efficiency comparable to those leading model checkers such as SPIN. The model checker also has the potential to be improved significantly.

1 Introduction

[Model Checking] Model checking [1] is a verification technique to verify whether a system has a property expressed as a temporal formula. Since many systems can be described as finite states models, model checking techniques can be applied in many areas. Model checking techniques are relatively new when compared with traditional formal verification techniques, such as theorem proving. Compared to other verification techniques, model checking has several advantages. For example, it is completely automatic, and special expertise is not required when model checkers are used. However, model checking techniques also need to handle several challenges such as the state explosion problem. In order to explore different model checking algorithms, it is a practical issue for programmers to find some convenient programming framework.

[Normalization Systems] Rewriting systems [5, 8] can provide an elegant framework for symbolic computation, theorem proving, equational reasoning and equational logic programming. These applications have the similar goal to find the normal forms of one or more terms. So to obtain efficient normalization algorithms is crucial in all these applications. The congruence closure based normalization algorithm (CCNA) [7] stores the history of its computations in a compact data

* Research partially supported by NSF grant CCF 036475

structure to eliminate repeating computations, and thus can do the normalization quickly. CCNA is implemented in LRR [3]

[Implementing Model Checking Using Rewriting] Since any computation can be implemented as normalization, and tabled normalization systems can reduce redundant computations, we think Model Checking algorithms can be implemented efficiently on normalization systems that have tabling.

Currently we are not aware of other works that use term rewriting and CCNA to implement model checking. Very recently we discovered that Maude [6] has implemented Büchi Automata based model checking using rewriting. Also we have noticed the work of [2] that uses the XSB interpreter and Prolog. The work of [2] uses different techniques of tabled resolution, as apposed to the CCNA tabling technique used in LRR.

It is interesting to implement model checking algorithms using LRR. To attack the state explosion problem of model checking, it is crucial to find out the redundant and repeated computation tasks. It is possible that the amount of repeated computation will be abundant in some model checking computation, since the same state will be visited repeatedly when the model checker is verifying some temporal logic formula. Therefore, we expect that by using the CCNA based normalization system, and letting it find out the repeated computation automatically, efficient and convenient model checkers can be built.

2 LRR

LRR [3] is a normalization system and its basic idea is the congruence closure based normalization algorithm. It has been developed at the Computer Science Department of University of Houston for several years, and has experienced considerable progress, and it is still evolving. LRR has different heuristics. Smaran is the normalization subsystem of LRR that use the full power of computation history, while TGR is the normalization subsystem without using the computation history.

Terms that need to be normalized can be given as expressions in prefix form. A programmer needs to specify the variables, constants, functions, and rewriting rules to the system. Currently, LRR has a modularized programming interface. A group of rules, variables, and functions can be defined together in a module file. A module file can export functions and variables that can be used by other modules. A module file can also import other modules. Every rule has LHS and RHS separated by =>. LHS must be a functional term.

For example, the following is the module file that defines the rules to compute fibonacci numbers.

```
module fib
  rem  (the comments) compute fibonacci numbers ;
  import ;
  export fib;
  var x ;
  func fib(1),f(2);
  rule
    fib(x)     => f(>(x,1),x) ;
    f(true,x)  => +(fib(-(x,1)),fib(-(x,2)));
    f(false,x) => 1 ;
end module fib
```

```
The following is a sample term file.

        fib(100)
        fib(25)
```

There are no type checking issues in LRR. LRR has its own strategies to handle the computation history automatically. Given a term to be normalized, a signature will be computed for it. The signatures of all equivalent terms will be grouped together. LRR will quickly find whether there is some equivalent term already computed in the history, and will avoid any possible repeated computation. Once the simple module interface is understood, programmers can implement algorithms quickly without worrying about language issues.

3 Model Checker

The model is interpreted as a state graph. The states and edges are specified in a module file, model.m. The current model checker has implemented the model checking algorithms with bottom-up style. Its goal is to find all of the states in the model that can satisfy the given temporal logic formula.

[CTL Model Checker] The implemented CTL module checker is based on the fixed-points algorithms [1]. The code is very compact, about 50 lines, and runs very efficiently. The computation of a fixed point will start with an empty set of states or the full set of states, and the set will change along with the steps of the computation. When the set can not change any more, it is the set of states that can satisfy the CTL formula. The complexity of the CTL checking algorithm chosen from the book [1] is $O(|f| (|S| + |R|))$, where $|f|$ is the length of the formula, $|S|$ is the number of states, and $|R|$ is the size of the transition relation.

[LTL Model Checker] The LTL model checker is implemented using the tableau based algorithms [1]. Given a model and a LTL formula, the algorithm will first construct a corresponding graph, called the atom graph. Then the LTL model checking task is reduced to finding all the states that have a corresponding path to a self-fulfilling strongly connected component (SCC) in the atom graph. A self-fulfilling SCC mean that, if a U formula, like f U g, appears in an atom in the SCC, there must exist an atom in the SCC, such that g appears in that atom.

The implemented SCC algorithm chosen from book [1] is based on the traditional depth-first-search algorithm. The LTL model checking algorithm has the complexity of $O((|S| + |R|) 2^{O(|f|)}$, which is obviously more time consuming than the CTL model checker.

[CTL* Model Checker] The temporal formulas that need to be verified can be expressed as CTL* formulas. CTL* has more expressive power than CTL and LTL. CTL* formulas include LTL and CTL formulas. Given a temporal formula, the model checker will automatically identify its category, and the LTL model checker or the CTL model checker will be called accordingly. The task to check a long CTL* formula can be divided into checking its sub-formulas. Eventually all CTL* model checking tasks will be handled by the LTL model checker or the CTL model checker. The CTL* model checker works correctly in all experiments.

So far the implemented model checking algorithms are in the preliminary stage, and do not include the more advanced techniques such as partial order reduction,

symbolic representation, Büchi Automata, and on-the-fly LTL model checking. These techniques are implemented in SPIN. We attribute the reasonable performance of our model checker in some of the experiments comparing to SPIN to the power of LRR.

[The Automatic Model File Generator] In order to do model checking for large models, it is necessary to generate the model files automatically, since it becomes impractical to do it manually. For example, as one of our experiments has showed, given two simple concurrent processes for the mutual exclusion problem, there are 8 states in the model. But with three such processes, the number of states quickly grows to 32.

A C program was written to generate model files automatically. The user only need to specify the code of the processes, then the corresponding model file will be generated by the C program. By doing this, the process description code expressed in other languages, such as Promela that is the protocol specification language used in SPIN, can be easily translated into the model files of this model checker.

4 Performances

We have compared the performances of our current model checker with the famous model checker SPIN version 4.1.2 [4]. When the temporal formula can be expressed using CTL formulas, the performance of our model checker is very close to SPIN. The following chart shows the running time of SPIN 4.1.2 and TGR (the rewriting system without using computation history) and Smaran. The experiment is to check four different CTL* formulas (the first three are expressible in both CTL and LTL, but the last one can only be expressed in LTL) with a small model of 8 states (two parallel process), and a bigger model with 32 states (three parallel processes, the model is automatically generated by the automatic model file generator), for the mutual exclusion problem. The experiment was performed on a desktop using Redhat Linux 9.0, with Pentium4 2.4GHZ CPU and 512MB memory. In Tables 1, 2, timings are in seconds.

Table 1. Performances of SPIN and TGR and Smaran with a model of 8 states

SPIN4.1.2	TGR		Smaran	
Real time	Normalizing time	Number of reductions	Normalizing time	Number of reductions
Formula 1 0.005	0.000	1378	0.010	402
Formula 2 0.006	0.340	37250	0.010	2620
Formula 3 0.006	0.180	14785	0.000	3316
Formula 4 0.006	1430.470	396578	273.410	136444

Table 2, Performances of SPIN, TGR, and Smaran with a model of 32 states

SPIN4.1.2	TGR		Smaran	
Real time	Normalizing time	Number of reductions	Normalizing time	Number of reductions
Formula1 0.006	0.030	6998	0.000	1019
Formula2 0.006	120.300	1342537	0.210	24464
Formula3 0.006	53.810	1504170	0.008	31560
Formula4 0.051	System limit reached	> 1906070	3626.36	1247044

From the above tables, we can see that our CTL model checker has comparable performance with SPIN. Smaran has a considerable performance improvement over TGR. By using tabling, the number of reductions can be reduced significantly. Notice the 0.000 times recorded by system means the normalization time is too short or the computation can be found in history. But the performance of our LTL model checker is not comparable to SPIN, due to the highly advanced algorithms implemented in SPIN, which can take the direct shortcut of the computation, and which has evolved for more than a decade.

Several hundred rules are implemented in a relatively short time, which is the first experience of the author with LRR. Comparing to the programming experience using other languages, we think Smaran can help the programmers improve their productivity considerably.

5 Future Work

We plan to improve the performance of LRR. More features will be built into LRR. The programming interface will be enriched and improved.

We will implement more efficient model checking algorithms. The current version of our model checker has the bottom-up style. We are also interested in the top-down style. We also want to research about the algorithms to handle models with infinite size. Since Smaran can provide a unified programming environment, it is possible to also implement the theorem proving algorithms together with model checker. When model checkers and theorem provers work together within Smaran, we expect a more powerful and convenient automatic verification tool.

The author plans to contribute in the areas of theory and implementation of model checking, and applications of LRR in his PhD thesis.

References

1. E. Clark, O. Grumburg, D.A. Peled: *Model Checking*, MIT Press, Cambridge, 1999
2. Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren: Efficient Model Checking by Tabled Resolution. Proceedings of the *Ninth International Conference on Computer Aided Verification (CAV'97)*, Haifa, Israel, Lecture Notes in Computer Science, Vol. 1243, Springer-Verlag (July 1997)
3. Rakesh M. Verma, S.A. Senanayake: *LRR: A Laboratory for Rapid Term Rewriting.* Proceedings of the International Conference on Rewriting Techniques and Applications (RTA), Springer-Verlag LNCS (July 1999)
4. SPIN web page: http://spinroot.com/spin/whatispin.html
5. Paul Chew: *An improved algorithm for computing with equations*. Proceedings of the Twenty-first Annual Symposium on Foundations of Computer Science, 1980, pages 108–117
6. Steven Eker, José Meseguer, Ambarish Sridharanarayanan: *The Maude LTL Model Checker and Its Implementation*. SPIN 2003, Proceedings. Lecture Notes in Computer Science 2648 Springer 2003, pages 230–234
7. R. M. Verma: *A theory of using history for equational systems with application*. J. ACM, 1995, Vol. 42, pages 984–1020
8. Leo Bachmair and Ramakrishnan, C. R. and Ramakrishnan, I. V. and Ashish Tiwari: *Normalization via rewrite closure*, Proceedings of the Eleventh International Conference on Rewriting, Vol. 1631, 1999, pages 190–204