Logic Programming Optimizations for Faster Model Checking

Yifei Dong, C.R. Ramakrishnan

1 Introduction

Over the last three years, we have showed that logic programming with tabulation can be used to construct efficient model checkers [RRR+97, CDD+98, DDR+99]. In particular, we have developed XMC [RRS+00], a model checker which verifies properties written in the alternation-free fragment of the modal mu-calculus [Koz83] for systems specified in XL, an extension of value-passing CCS [Mil89]. The XMC system is available from `http://www.cs.sunysb.edu/~lmc`.

XMC is implemented atop the tabled logic programming system XSB [XSB00]. Its initial implementation [RRR+97] consisted of two predicates `trans/3`, which encoded the SOS semantics of XL terms, and `models/2`, which encoded the semantics of mu-calculus formulas. The predicate `trans/3` computes the transition relation of the automaton corresponding to the given XL specification. The `models/2` predicate determines whether a given state in the automaton models a given formula. In effect, model checking is done by query evaluation.

In a recent paper [DR99], we showed that XMC's performance can be significantly improved by *compiling* XL specifications into transition rules, which are an efficient representation of low-level automata. The compiler, at a very coarse level, can be viewed as partially evaluating the `trans/3` relation with respect to an XL specification, and then optimizing the resultant program.

In this abstract, we consider the orthogonal problem of compiling formulas. The compilation process is done in four steps. First, we transform the formula at mu-calculus level itself, to enable the subsequent transformations. We then partially evaluate `models/2` with respect to the transformed formula. We convert right recursion to left (under certain conditions) in the partially evaluated program. The specialization done in the second step can result in storing the same automaton state in multiple tables. The final step introduces a dictionary structure to ensure that state representations are shared.

We first describe the encoding of the `models/2` predicate and describe each step in the compilation. We present experimental results and discuss the effectiveness of optimizations performed in each of the steps.

The authors are at the Department of Computer Science, SUNY Stony Brook, Stony Brook, NY 11794, USA. Their emails addresses are: `ydong@cs.sunysb.edu`,`cram@cs.sunysb.edu`.

This work was support in part by NSF grants CCR-9711386, CCR-9876242 and EIA-9705998.

```
models(S, or(F1, F2)) :- models(S, F1); models(S, F2).
models(S, diamMinus(A, F)) :-
       trans(S, B, T), B \== A,models(T, F).
models(S, boxMinus(A, F)) :-
       findall(T, (trans(S, B, T), L), B \== A), L),
       all_models(L, F).
models(S, form(X)) :- rec_models(S, X).

all_models([], _).
all_models([T|Ts], F) :- models(T, F), all_models(Ts, F).

:- table rec_models/2.
rec_models(S, X) :- def(X, F), models(S, F).
```

Figure 1: Fragment of mu-calculus model checker `models/2`.

2 Compiling and Optimizing the Model Checker

Formulas to be verified are written in XL using an equational mu-calculus syntax; the abstract syntax of these formulas are represented using `def/2` facts. For instance the property of deadlock freedom, expressed in XL as "`df += [-]ff \/ <->df`" is represented in the abstract syntax as:

```
def(df, or(boxMinus(nil, ff), diamMinus(nil, form(df))))
```

The formula `ff` is false at all states, and `nil` is a transition label that does not occur in the automaton to be verified. A formula `diamMinus(`a`,`f`)` is true at state s if there is a non-a transition from s to t such that f is true at t. A formula of the form `boxMinus(`a`,`f`)` is true at s if for every non-a transition from s to t,f is true at t. We use the notation of the form "`df` ="... for least fixed point equations. This semantics is encoded in the definition of predicate `models/2`, a fragment of which is shown in Figure 1. Even in the initial encoding table only the rule that handles recursion; the rest of the rules are not tabled.

In the following, we describe the various optimization steps in formula compilation, namely, formula transformation, partial evaluation, recursion transformation, and dictionary creation. The effect of these optimizations in large XMC benchmarks is shown in Table 1. In the table, "iproto" refers to the i-Protocol model [DDR+99], "leader" refers a model of leader election algorithm adapted from SPIN's test suite [Hol97], and "metalock" refers to a model of the Java Metalocking algorithm [BSW00]. The i-Protocol benchmark verifies a livelock property on a faulty version (bug) and a corrected version, and with two window sizes (w=1, w=2). The "leader" benchmark verifies a liveness property for two instances: ring sizes 5 and 7. The "metalock" benchmark verifies a safety property (mutual exclusion) on an instance with 3 objects and 2 threads. All measurements were made on a Sun Enterprise 4000 with 2G memory.

model/method	time	space			
		total	permanent	stack	table
iproto w=1 bug					
original	0.09	2.4M	618K	34K	67K
after partial evaluation	0.07	2.2M	419K	31K	75K
after left recursion	0.90	2.7M	419K	57K	567K
after interning	1.00	2.6M	668K	28K	215K
iproto w=2 bug					
original	0.45	2.7M	618K	143K	307K
after partial evaluation	0.45	2.5M	419K	138K	315K
after left recursion	4.32	4.1M	419K	116K	1956K
after interning	5.08	4.0M	1452K	63K	777K
iproto w=1 fix					
original	14.92	31.5M	421K	14.7M	5.4M
after partial evaluation	14.84	32.7M	420K	13.6M	6.5M
after left recursion	13.12	7.7M	420K	77K	5.5M
after interning	13.84	6.3M	2105K	37K	2.5M
iproto w=2 fix					
original	228.26	256.7M	421K	161.9M	50.7M
after partial evaluation	227.20	266.7M	420K	150.6M	60.7M
after left recursion	144.37	53.7M	420K	133K	51.6M
after interning	152.15	43.1M	17.3M	68K	24.2M
leader5 one-leader					
original	0.75	3.6M	400K	1104K	722K
after partial evaluation	0.70	3.6M	400K	1123K	706K
after left recursion	not applicable				
after interning	0.75	2.8M	976K	371K	129K
leader7 one-leader					
original	16.06	45.3M	411K	26.2M	12.9M
after partial evaluation	15.93	45.1M	411K	26.4M	12.6M
after left recursion	not applicable				
after interning	15.34	21.6M	11.3M	6.0M	1.8M
metalock 3x2 nomutex					
original	23.31	39.7M	583K	22.9M	12.8M
after partial evaluation	22.29	41.0M	392K	19.0M	14.3M
after left recursion	18.87	14.3M	394K	848K	11.9M
after interning	21.53	12.0M	7.9M	1266K	2.2M

Table 1: Effect of different stages of formula compilation

Formula Transformation: The first step in compilation is to transform the formulas themselves. We transform box modalities to their dual diamond modalities as long as the transformation does not introduce cycles with negation in the corresponding evaluation. In particular, we transform `boxMinus(`a`,`f`)` in the definition of a formula g into `neg(diamMinus(`a`, neg(`f`)))` whenever the definitions of f are g are not mutually recursive. For instance, the deadlock freedom formula is transformed to:

```
def(df, or(neg(diamMinus(nil, neg(ff))),
           diamMinus(nil, form(df))))
```

This transformation necessitates the addition of the following rules to `models/2`:

```
models(S, neg(F)) :- tnot(tab_models(S, F)).
:- table tab_models/2.
tab_models(S, F) :- models(S, F).
```

The predicate `tab_models/2` is simply the tabled version of `models/2`. The additional table can be removed in the subsequent partial evaluation phase, as described below.

Partial Evaluation: The next step in optimization partially evaluates `models/2` with respect to the given formula. Partial evaluation involves traditional specialization and a few simple transformations that are specific to the model checker.

For example, the deadlock freedom property yields a specialized model checker of the form:

```
:- table rec_models_df/1.
rec_models_df(S) :- tnot(tab_models_diam(S)) ;
                    trans(S,B,T), B \== nil, rec_models_df(T).

:-table tab_models_diam/1.
tab_models_diam(S) :-
                    trans(S,B,T), B \== nil,
                    tnot(tab_models_notff(S)).

:-table tab_models_notff/1.
tab_models_notff(S).
```

We can eliminate `tab_models_notff/1` using a trivial partial evaluation step. Observe that `tab_models_diam/1` need not be tabled and does not invoke a tabled predicate; hence the tabled negation `tnot` can be replaced with Prolog negation. Finally, since `nil` does not occur as a label on any transition, `B\==nil` always succeeds. This results in the following definition of `rec_models_df/1`:

```
:- table rec_models_df/1.
rec_models_df(S) :- not(trans(S,_,_)) ;
                    trans(S,_,T), rec_models_df(T).
```

Partial evaluation, by itself, has little effect on performance. For instance, for verifying the deadlock freedom property on the "metalock" benchmark (not shown in table). Partial evaluation alone speeds up model checking by about 8% compared to XMC without formula compilation. Partial evaluation *after* formula transformation results in a speedup of over 60%. The formula transformation step does not apply to any of the benchmarks shown in Table 1 and hence we observe a speedup of 5% or less on those examples. More significantly, this step makes the recursive structure of the formula explicit in the model checker, and thereby enables the recursion transformation.

Recursion Transformation: It is well-known that bottom-up parsing algorithms (such as LR(1)) perform better while parsing left-recursive grammars, due to fewer symbols stored on the shift-reduce stack. A similar performance improvement can be seen in a tabling environment when right-recursive predicates are converted to left recursive ones. For instance, the right-recursive `rec_models_df/1` predicate can be transformed into `models_df_left` defined as:

```
models_df_left(S) :- reach(S, T), not(trans(T,_,_)).
:- table reach/2.
reach(S, S).
reach(S, T) :- reach(S, U), trans(S, _, T).
```

Note that `models_df_left` is not tabled.

This transformation does not affect the complexity of evaluation as long as there is only one query to `models_df_left/1`. If the predicate being transformed results from an inner fixed point, then there may be multiple queries to the predicate. Since multi-source reachability is quadratic, the transformation may produce unacceptable slow downs. Hence we apply this transformation only on the predicate derived from the outermost fixed point in the mu-calculus formula. The outermost fixed point of the formula "one_leader" used in the leader benchmark cannot be made left recursive (due to a box modality). We see that the space and time requirements of all benchmarks except the faulty "iproto" are significantly improved by recursion transformation.

The recursion transformation can change the order in which the model checking search is done. Hence, for formulas that do not require complete exploration of the state space, the transformation can cause dramatic slow downs or speedups, depending on how the search order was changed. This is apparent the order-of-magnitude slowdown observed for the faulty version of "iproto".

State Dictionary: Partial evaluation of model checking with respect to a set of formulas may have adverse effect on the table space. When the original `models/2` is used, the trie representation of calls and answers in tables [RRS+95] shares the states across multiple formulas, since the trie linearizes terms by a preorder traversal. Specialization destroys this sharing opportunity. We use a dictionary of states to ameliorate this loss.

Each state is stored in a separate structure, using XSB's `intern` library. The system assigns to each "interned" term an unique index which can be used as a handle for lookup. The states are then represented only by their intern indices. Using such a dictionary, any state sharing that was destroyed by specialization can be restored. However, the dictionary lookup at each step (to check for transitions from a state) increases model checking time.

Note that state dictionary is not useful for a single (non-nested) fixed point formula, since there is no loss in sharing due to specialization. All formulas used in the benchmarks shown in Table 1 have nested fixed points. For such formulas, we see space savings in the range from 4% to over 200%. The use of dictionary increases run time by up to 15%.

3 Conclusions

We presented several simple LP-based techniques to improve the space and time performance of a model checker. We find that these techniques can yield considerable savings in time (up to 40%) and space (by a factor of two or more) when exhaustive state-space search is necessary. When only a portion of the state space needs to be explored, the techniques may change the search order, consequently worsening the performance. Hence it seems appropriate to use these optimizations when exhaustive exploration is expected: especially checking for "bugs" in the final stages of a design. We observe that further improvements in the performance are unlikely if the techniques are based solely on the properties being verified. Techniques to reduce the state space of the system based on the given formula hold significant promise and are currently being studied.

References

[BSW00] S. Basu, S. A. Smolka, and O. R. Ward. Model checking the Java Meta-Locking algorithm. In *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2000)*, Edinburgh, Scotland, April 2000.

[CDD+98] B. Cui, Y. Dong, X. Du, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, A. Roychoudhury, S. A. Smolka, and D. S. Warren. Logic programming and model checking. In *Static Analysis Symposium*. Springer Verlag, 1998.

[DDR+99] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I.V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-Protocol: A comparative study of verification tools. In *Tools and Algorithms for the Construction and Analysis of Algorithms (TACAS '99)*, Lecture Notes in Computer Science, Amsterdam, March 1999. Springer Verlag.

[DR99] Y. Dong and C. R. Ramakrishnan. An optimizing compiler for efficient model checking. In *Proceedings of FORTE/PSTV '99*, 1999.

[Hol97] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[Koz83] D. Kozen. Results on the propositional μ-calculus. *Theoretical Computer Science*, 27:333–354, 1983.

[Mil89] R. Milner. *Communication and Concurrency*. International Series in Computer Science. Prentice Hall, 1989.

[RRR+97] Y. S. Ramakrishna, C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, T. W. Swift, and D. S. Warren. Efficient model checking using tabled resolution. In *Proceedings of the 9th International Conference on Computer-Aided Verification (CAV '97)*, Haifa, Israel, July 1997. Springer-Verlag.

[RRS+95] I. V. Ramakrishnan, P. Rao, K. Sagonas, T. Swift, and D. S. Warren. Efficient tabling mechanisms for logic programs. In *International Conference on Logic Programming*, 1995.

[RRS+00] C. R. Ramakrishnan, I. V. Ramakrishnan, S. A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V. N. Venkatakrishnan. XMC: A logic-programming-based verification toolset. In *Computer Aided Verification (CAV)*, 2000.

[XSB00] The XSB Group. The XSB logic programming system v2.1, 2000. Available from `http://www.cs.sunysb.edu/~sbprolog`.