Model Checking Linear Temporal Logic Using Tabled Logic Programming

L. Robert Pokorny, C.R. Ramakrishnan

1 Introduction

Logic Programming (LP) has been used in the last few years to construct model checkers [RRR+97, DP99, LM99]. These model checkers verify whether a system of interest satisfies properties expressed as formulas in some temporal logic. They are constructed by directly encoding the semantics of the temporal logic as a logic program. This program is then evaluated for the system and formula of interest using tabled resolution or bottom-up techniques. The system to be verified is given in terms of a state transition relation, either written directly or compiled from a process algebra notation. Efficient tabled LP systems such as XSB [XSB00] permit us to construct practical model checkers using this approach.

The works described in [RRR+97, DP99, LM99] implement model checkers for modal mu-calculus [Koz83] and CTL. The model checking problem in these logics reduces to solving set equations with least or greatest fixed points. Such equations can be solved by using the least model computation of tabled resolution. The XMC model checker [RRS+00] has shown that the simplicity of implementation does not come at the cost of performance: XMC performs as well as hand-crafted model checkers [CDD+98]. The XMC system can be obtained (with full sources) from `http://www.cs.sunysb.edu/~lmc`.

Model checking linear temporal logic (LTL) and CTL* involve path-based computations: in particular, detecting whether cycles of a particular form are reachable from the start state of the transition system. In this abstract, we describe how a model checker for LTL is constructed using tabled LP.

The solution presented in the abstract contains several points of interest to logic programmers. First of all, we formulate the proof rules for LTL by using a dual of the tableau construction method proposed by Bhatt, Cleaveland and Grumberg [BCG95] (see Section 2). This formulation relies on tabulation and the ability to handle stratified negation. Secondly, we present a programming abstraction of "inflationary negation" (`inot`) which is used to encode the strongly connected component (SCC) algorithm due to Kosaraju and Sharir described in [CLR89] (see Section 3). This illustrates combining procedural programming with tabled resolution.

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

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

Thirdly, we present optimizations which ensure that the computations associated with cycle detection are performed only where needed (see Section 4). This highlights how tabled LP promotes the programming style where correct programs are successively refined to obtain efficient implementations. Finally, we show the flexibility of this formulation by extending it to LTL with actions as proposed in [BCG00] (see Section 4). The performance of our model checker is currently being evaluated.

2 LTL Model Checking

LTL is a logic for specifying temporal properties of finite-state systems. In this abstract, we consider propositional LTL with the following syntax (P is the finite set of propositions):

$$
\begin{array}{lll}
\Psi & \rightarrow & \mathtt{A}\Phi \\
\Phi & \rightarrow & p \mid \neg p \mid \Phi \wedge \Phi \mid \Phi \vee \Phi \mid \Phi \ \mathtt{U} \ \Phi \mid \Phi \ \mathtt{R} \ \Phi \mid \mathtt{X}\Phi \qquad\qquad p \in P
\end{array}
$$

The semantics of LTL operators can be described informally as follows (see, e.g., [BCG95] for a formal treatment). We consider Kripke structures, which are finite directed graphs with nodes representing states, and edges representing transitions between states. Each state is labeled with a set of propositions (a subset of P) that are true at that state.

The semantics of LTL is traditionally given in terms of infinite paths (called *runs*) of a Kripke structure. We say that an LTL formula $\mathtt{A}\phi$ is true at a state s iff it is true for every run starting at s. The operator $\mathtt{A}$ is called the universal path quantifier. The operators $\mathtt{U}$, $\mathtt{R}$ and $\mathtt{X}$ are the until, release, and next-state modalities respectively. A base proposition $p \in P$ is true for a run $\pi = s_0, s_1, \ldots$ iff p is true at s_0. The treatment of negated propositions and boolean connectives $\wedge$ and $\vee$ are standard. A formula $\mathtt{A}(\phi_1 \ \mathtt{U} \ \phi_2)$ is true at s_0 iff for every run $\pi = s_0, s_1, \ldots$ there is a $k \geq 0$ such that $\mathtt{A}\phi_2$ is true at s_k and $\forall 0 \leq i < k$ $\mathtt{A}\phi_1$ is true at s_i. A formula $\mathtt{A}(\phi_1 \ \mathtt{R} \ \phi_2)$ is true at s_0 iff for every run $\pi = s_0, s_1, \ldots$, either $\mathtt{A}\phi_2$ holds at all s_i, or there is a $k \geq 0$ such that $\mathtt{A}\phi_1$ holds at s_k and $\forall 0 \leq i \leq k$ $\mathtt{A}\phi_2$ holds at s_i. A formula $\mathtt{AX}\phi$ is true at s_0 iff ϕ is true at s_1 whenever there is a transition from s_0 to s_1.

Our syntax can be seen as describing LTL formulas in negation normal form. Similar to $\wedge$ and $\vee$, $\mathtt{U}$ and $\mathtt{R}$ are duals (i.e., $\neg(\phi_1 \ \mathtt{U} \ \phi_2) = \neg\phi_1 \ \mathtt{R} \ \neg\phi_2$), and $\mathtt{X}$ is its own dual (i.e., $\neg\mathtt{X}\phi = \mathtt{X}\neg\phi$). It is useful to consider the existential path quantifier $\mathtt{E}$, which is a dual of $\mathtt{A}$ (i.e., $\neg\mathtt{A}\psi = \mathtt{E}\neg\psi$). The semantics of the extended LTL formulas with $\mathtt{E}$ can be derived from the above semantics using the duality.

Figure 1 gives a tableau based proof system for LTL with $\mathtt{E}$ quantifier, closely following the system for LTL with $\mathtt{A}$ quantifier presented by Bhat, Cleaveland, and Grumberg [BCG95]. Each node in the tableau is an assertion of the form $s \vdash \mathtt{E}(\Phi)$ where Φ is a set of (unquantified) formulas. Semantically, $s \vdash \mathtt{E}\Phi$ where $\Phi = \{\phi_1, \ldots \phi_n\}$ is equivalent to $s \vdash \mathtt{E}(\phi_1 \wedge \cdots \wedge \phi_n)$. We write $\mathtt{E}(\Phi, \phi_1, \ldots, \phi_k)$ to denote $\mathtt{E}(\Phi \cup \{\phi_1, \ldots, \phi_k\})$.

$$
\begin{array}{llllll}
(1) & \dfrac{s \vdash \mathtt{E}(\Phi, p)}{s \vdash \mathtt{E}(\Phi)} \quad p \text{ is true in } s & (2) & \dfrac{s \vdash \mathtt{E}(\{\})}{\mathit{true}} & (3) & \dfrac{s \vdash \mathtt{E}(\Phi, \phi_1 \wedge \phi_2)}{s \vdash \mathtt{E}(\Phi, \phi_1, \phi_2)}
\end{array}
$$

$$
(4) \quad \frac{s \vdash \mathtt{E}(\Phi, \phi_1 \vee \phi_2)}{s \vdash \mathtt{E}(\Phi, \phi_1)} \qquad (4') \quad \frac{s \vdash \mathtt{E}(\Phi, \phi_1 \vee \phi_2)}{s \vdash \mathtt{E}(\Phi, \phi_2)}
$$

$$
(5) \quad \frac{s \vdash \mathtt{E}(\Phi, \phi_1 \;\mathtt{U}\; \phi_2)}{s \vdash \mathtt{E}(\Phi, \phi_2)} \qquad (5') \quad \frac{s \vdash \mathtt{E}(\Phi, \phi_1 \;\mathtt{U}\; \phi_2)}{s \vdash \mathtt{E}(\Phi, \phi_1, \mathtt{X}(\phi_1 \;\mathtt{U}\; \phi_2))}
$$

$$
(6) \quad \frac{s \vdash \mathtt{E}(\Phi, \phi_1 \;\mathtt{R}\; \phi_2)}{s \vdash \mathtt{E}(\Phi, \phi_1, \phi_2)} \qquad (6') \quad \frac{s \vdash \mathtt{E}(\Phi, \phi_1 \;\mathtt{R}\; \phi_2)}{s \vdash \mathtt{E}(\Phi, \phi_2, \mathtt{X}(\phi_1 \;\mathtt{R}\; \phi_2))}
$$

$$
(7) \quad \frac{s \vdash \mathtt{E}(\mathtt{X}\phi_1, \ldots, \mathtt{X}\phi_n)}{t \vdash \mathtt{E}(\phi_1, \ldots, \phi_n)} \quad (s, t) \in T
$$

Figure 1: Tableau Rules for LTL Model Checking

These rules can be used to construct a proof graph that represents the dependence between the various assertions. Each assertion can be viewed as a proof obligation, and the children of the assertion as the different ways to dispense that obligation. The proof succeeds if we can reach either (1) the leaf *true*, or (2) a "good" cycle, defined as follows. A cycle in the proof graph is deemed good if every until obligation in the cycle is satisfied in that cycle: in other words, until obligations are not postponed indefinitely. Finally, we can show that $s \vdash \mathtt{A}\phi$ iff $s \not\vdash \mathtt{E}\overline{\phi}$ where $\overline{\phi}$ is the negation normal form of $\neg\phi$.

Note that the implementation of this proof system needs cycle checking. It turns out that this check can be accomplished by inspecting the nodes in the strongly connected components of the proof graph. In this case, the number of checks is proportional to the size of the proof graph, whereas checking all cycles can be exponential in the size of the proof graph. We now describe how the proof system in Figure 1 is encoded as a tabled logic program.

3 Implementing the Proof System

Our implementation represents a Kripke structure by using three types of relations (EDB). These are:

```
base(S,P)           % Proposition P holds in state S
transition(S1,S2)   % State S1 has a transition to state S2
startstate(S)       % S is the start state of interest
```

An assertion of the form `S⊢F` is represented by a term `models(S,F)`. An edge in the proof graph is computed as follows. Given an state `S` and a formula `F`, we first separate the proof obligations that can be checked in the current state from those that can only be checked in the next state. This is done by a predicate `separate(F, T, NF)` which, given a formula `F`, splits the obligation into this-state obligations T and next-state obligations NF. separate/3 is a direct encoding of the tableau rules in Figure 1. Note that all this-state obligations will be base propositions or their negation. We determine if the set of this-state obligations are met, by checking against base/2 for each member of the set (predicate all_true/2 below). If there are any next-state obligations, then there is an edge from models(S,F) to models(T,NF) such that S and T are related via transition/2. If there are no next-state obligations, then there is an edge from models(S,F) to true. This is encoded in XSB as follows:

```
edge(models(S,F),A) :-
        separate(F,T,NF),
        all_true(S,T).
        (NF = []
            -> A = true
            ;  transition(S,S1),
               A = models(S1,NF)
        ).
```

The path relation is simply the transitive closure of edge relation. The predicates path/2, edge/2, and separate/3 are all tabled: path/2 to ensure termination, and the other two for efficiency.

Recall that a formula F is true if there is a path to true or to a good cycle in the proof graph. Let leader(A,C,B) be a predicate such that B is the leader of a SCC reachable from A, and C is a node in that SCC. Then, the presence of a proof for assertion A can be encoded by predicate valid_proof/1 defined as:

```
valid_proof(A) :-
    path(A,X), X=true.
valid_proof(A) :-
        bagof(C, C^ (leader(A,C,B)), L),
                    % L contains all vertices in an SCC
                    % with leader node B
        check_scc(L).
```

In the above, check_scc(L) determines if the until-obligations of all assertions in L are satisfied in L. Let $\{u_1, \ldots u_n\}$ be a collection of sets where u_i is the set of until obligations of an assertion a_i in L. The until-obligations of all assertions in L are satisfied in L itself iff the collection $\{u_1, \ldots u_n\}$ has a null intersection.

The efficiency (and asymptotic complexity) of the encoding hinges on the implementation of leader/3. SCC detection can be done in linear-time using Tarjan's algorithm [Tar72]; however, this requires maintaining updateable "lowlinks" at each node in the graph, and hence is highly procedural.

We use the algorithm due to Kosaraju and Sharir (see [CLR89, p.488–493]) since it is amenable to a much cleaner implementation. Given a graph G and a root node in G this algorithm first assigns post-order numbers to all nodes in G. Then, it constructs a graph G^T by reversing the directions in the edges in G, and builds a forest of depth-first-search (DFS) trees in G^T starting from the node with the highest post-order number. Each tree is a SCC of G, and the root of the trees are the leaders.

Using this algorithm, `leader/3` is implemented as follows:

```
:- table leader/3.

leader(Root,Node,Leader) :-
  fwd_reach(Root,Leader),      % Returns reachable nodes in
                               %    reverse post order
  inot(leader(Root,Leader,_)), % The selected node is not in
                               %    previously found SCC
  (Node = Leader;              % The SCC consists of the leader
   back_reach(Leader,Node)).   %    and nodes backward reachable
                               %    from the leader
```

The predicate `inot(A)` is "inflationary not" and succeeds iff an answer `A` has not been already computed. This serves as a useful abstraction to mark nodes in a graph as long as the marks are monotonic (i.e., there is no corresponding unmark). The monotonicity of marking makes this algorithm amenable to direct encoding compared to the DFS-based algorithm.

Inflationary-not is implemented using XSB builtins that inspect table states: `get_calls` which finds calls that are currently stored in the table, and `get_returns` which finds answers currently stored in a given call table.

```
inot(C) :- \+ ( get_calls(C,Cs,Rs), get_returns(Cs,Rs)).
```

This implementation of `leader/3` to return nodes in strongly connected components runs in linear time. Hence the logic program implementation of the LTL proof system runs in time linear in the size of the proof graph. The proof graph size is $(O(N \times 2^k))$ where N is the the number of states in the system being checked, and k is the length of the LTL formula.

4 Conclusions and Future Work

The LTL model checker has been implemented in XSB[1]. At the time of this writing, we have tested the implementation on a suite of small examples, but have only preliminary data on performance.

Optimizations: Our experience with building such systems is that reachability properties can be checked very efficiently in XSB since they involve only a transitive closure operation. The interesting question is whether the more general LTL model checker can be optimized such that its performance is close to that of reachability checkers for these specialized queries. It turned out that the LTL model checker, as described above, was slower by a factor of 6 to 8 for reachability properties on

[1]Our implementation can be obtained from `http://www.cs.sunysb.edu/~lmc/expt/ltl`

a large benchmark (deadlock detection in Java Metalock algorithm [BSW00]). We have since cut this factor to less than 2, mainly by ensuring that the cost of cycle-checking is incurred only where necessary. We found two main sources of inefficiency. First, the formulation as outlined above will perform SCC computation whenever a `true` assertion cannot be reached. However, if the formula has no release operators, and `true` is not reachable, then there are no good cycles. Hence, there is no need to compute SCCs in such cases. We ensure this by checking for release operators in the formula in the second clause of `good_path/1` in the above encoding. Second, recall that the SCC computation needs information on back edges. This was originally computed and stored when the forward edges are computed using `edge/2`, which is wasteful if SCC computation is not used. Hence, the computation of back edges is now done only in the post-order numbering phase of `leader/3`. We believe that a general LTL model checker that is only two times slower than a special-purpose reachability checker is definitely within bounds of acceptability. It will be interesting to see what other optimizations will be possible that will lower this performance difference even further.

Extensions: The base propositions in LTL formulas refer to properties of states. It is often convenient to include properties on actions (i.e., transitions) as well. Labeled transition systems (LTSs) are used to represent action-based operational behavior of processes. Action-based LTLs have been proposed to express properties of LTSs. Most proposals add labels to modalities and path quantifiers; extending our encoding to deal with these labels is trivial. One of the more interesting proposals for an action-based logics is GCTL* [BCG00], where sets of labels are treated as path formulas of CTL*, while base propositions are state formulas. We extended our LTL model checker to the linear-time fragment of GCTL* (i.e., path quantifiers appear at the outermost level). The extension involved changing (i) the syntax of formulas to accommodate label sets (denoted by $label(L)$, where L is a set of labels) (ii) the binary transition relation to a ternary one to by adding transition labels, and (iii) rule 7 in Figure 1 to:

$$\frac{s \vdash \mathtt{E}(\mathtt{X}\phi_1, \ldots, \mathtt{X}\phi_n, action(L_1), \ldots, action(L_k))}{t \vdash \mathtt{E}(\phi_1, \ldots, \phi_n)} \quad (s, a, t) \in T, a \in \cap_{1 \leq i \leq k} L_i$$

The last change involved simple modifications to `separate/3` and `edge/2`; the rest of the encoding is unchanged. We are currently investigating extending the LTL model checker to the entire class of GCTL*.

References

[BCG95] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *IEEE Symposium on Logic in Computer Science*. IEEE Press, 1995.

[BCG00] G. Bhat, R. Cleaveland, and A. Groce. Efficient model checking via Büchi tableau automata. Technical report, Department of Computer Science, SUNY, Stony Brook, 2000.

[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.

[CLR89] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. MIT Press, 1989.

[DP99] G. Delzanno and A. Podelski. Model checking in CLP. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, pages 223–239, 1999.

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

[LM99] M. Leuschel and T. Massart. Infinite state model checking by abstract interpretation and program specialisation. In *Proceedings of LOPSTR'99*, pages 63–82. Springer Verlag, 1999.

[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+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.

[Tar72] R. Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1(2):146–160, 1972.

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