To appear: Proceedings of PLIP/ALP'98

Logic Programming and Model Checking[⋆]

Baoqiu Cui, Yifei Dong, Xiaoqun Du, K. Narayan Kumar, C. R. Ramakrishnan, I. V. Ramakrishnan, Abhik Roychoudhury, Scott A. Smolka, David S. Warren

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794–4400, USA
`http://www.cs.sunysb.edu/~lmc`

Abstract. We report on the current status of the LMC project, which seeks to deploy the latest developments in logic-programming technology to advance the state of the art of system specification and verification. In particular, the XMC model checker for value-passing CCS and the modal mu-calculus is discussed, as well as the XSB tabled logic programming system, on which XMC is based. Additionally, several ongoing efforts aimed at extending the LMC approach beyond traditional finite-state model checking are considered, including compositional model checking, the use of explicit induction techniques to model check parameterized systems, and the model checking of real-time systems. Finally, after a brief conclusion, future research directions are identified.

1 Introduction

In the summer of 1997, C.R. Ramakrishnan, I.V. Ramakrishnan, Smolka, and Warren were awarded a four-year NSF Experimental Software Systems (ESS) grant[1] to investigate the idea of combining the latest developments in concurrency research and in logic programming to advance the state-of-the art of system specification and verification. This was the first year of the ESS program at NSF, and its goal is to support experimental investigations by research teams dedicated to making fundamental progress in software and software engineering. The ESS program director is Dr. William W. Agresti.

The current primary focus of our ESS grant is *model checking* [CE81, QS82, CES86], the problem of determining whether a system specification possesses a property expressed as a temporal logic formula. Model checking has enjoyed wide success in verifying, or finding design errors in, real-life systems. An interesting account of a number of these success stories can be found in [CW96b].

We call our approach to model checking *logic-programming-based model checking*, or LMC for short, and it is centered on two large software systems developed independently at SUNY Stony Brook: the Concurrency Factory [CLSS96] and

[⋆] Research supported in part by NSF grants CDA–9303181, CCR–9404921, CCR–9505562, CDA–9504275, CCR–9705998, CCR–9711386 and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087.

[1] There are two additional co-Principal Investigators on the grant who are no longer at Stony Brook: Y.S. Ramakrishna of Sun Microsystems and Terrance Swift located at the University of Maryland, College Park.

XSB [XSB98]. The Concurrency Factory is a specification and verification environment supporting integrated graphical/textual specification and simulation, and model checking in the modal mu-calculus [Koz83] temporal logic. XSB is a logic programming system that extends Prolog-style SLD resolution with *tabled resolution*. The principal merits of this extension are that XSB terminates on programs having finite models, avoids redundant subcomputations, and computes the well-founded model of normal logic programs.

Verification systems equipped with model checkers abound. For example, `http://www.csr.ncl.ac.uk:80/projects/FME/InfRes/tools` lists over 50 specification and verification toolkits, most of which support some form of model checking. Although these tools use different system-specification languages and property-specification logics, the semantics of these logics and languages are typically specified via structural recursion as (least, greatest, alternating) fixed points of certain types of functionals.

It is therefore interesting to note that the semantics of negation-free logic programs are given in terms of minimal models, and Logic Programming (LP) systems attempt to compute these models. The minimal model of a set of Horn clauses is equivalent to the least fixed point of the clauses viewed as equations over sets of atoms. Hence, model checking problems involving least fixed points can be naturally and concisely cast in terms of logic programs. Problems involving greatest fixed-point computations can be easily translated into least fixed-point computations via the use of logical negation.

However, Prolog-style resolution is incomplete, failing to find minimal models even for datalog (function-free) programs. Moreover, the implementation of negation in Prolog differs from the semantics of logical negation in the model theory. Consequently, traditional Prolog systems do not offer the needed support to directly implement model checkers. As alluded to above, evaluation strategies such as tabling [TS86, CW96a] overcome these limitations (see Section 2.2). Hence, tabled logic programming systems appear to offer a suitable platform for implementing model checkers. The pertinent question is whether one can construct a model checker using this approach that is efficient enough to be deemed practical.

The evidence we have accumulated during the first year of our LMC project indicates that the answer to this question is most definitely "yes." In particular, we have developed XMC [RRR$^+$97], a model checker for Milner's CCS [Mil89] and the alternation-free fragment [EL86] of the modal mu-calculus. The full *value-passing* version of CCS is supported, and a generalized prefix operator is used that allows arbitrary Prolog terms to appear as computational units in XMC system specifications. Full support for value-passing is essential in a specification language intended to deal with real-life systems such as telecommunications and security protocols.

XMC is written in approximately 200 lines of XSB tabled-logic-programming Prolog code, and is primarily intended for the model checking of finite-state systems, although it is capable of handling certain kinds of infinite-state systems, such as those exhibiting "data independence" [Wol86]. With regard to the efficiency issue, XMC is highly competitive with state-of-the-art model checkers hand-coded in C/C++, such as SPIN [HP96] and the Concurrency Factory [CLSS96]. This performance can be attributed in part to various aspects of the underlying XSB implementation, including its extensive support of tabling and the use of trie data structures to encode tables. In [LRS98] we describe how XMC can be extended to the full modal mu-calculus.

Buoyed by the success of XMC, we are currently investigating ways in which the LMC approach can be extended beyond traditional finite-state model checking. In particular, the following efforts are underway.

- An LMC-style specification of a model checker is given at the level of semantic equations, and is therefore not limited to any specific system-specification language or logic. For example, we have built a *compositional* model checker, simply by encoding the inference rules of the proof system as Horn clauses (Section 4.1).
- Traditionally, model checking has been viewed as an *algorithmic* technique, although there is a flurry of recent activity on combining model checking with *deductive* methods. Observe that (optimized) XSB meta-interpreters can be used to execute arbitrary deductive systems. Hence, the LMC approach offers a unique opportunity to fully and flexibly integrate algorithmic and deductive model checking, arguably the most interesting problem being currently researched by the verification community. To validate this claim, we have been experimenting with ways of augmenting XMC with the power of induction, with an eye toward the verification of *parameterized systems* (Section 4.2).
- By using constraints (as in Constraint LP [JL87]) to finitely represent infinite sets and tabled resolution to efficiently compute fixed points over these sets, we are looking at how tabled constraint LP can be used to verify *real-time systems* (Section 4.3).

The rest of the paper is structured follows. Section 2 shows how model checking can be essentially viewed as a problem of fixed-point computation, and how fixed points are computed in XSB using tabled resolution. Section 3's focus is our XMC model checker, and Section 4 describes ongoing work on extending the XMC technology beyond traditional finite-state model checking. After a brief conclusion, Section 5 identifies several promising directions for future research.

2 Preliminaries

In this section we describe the essential computational aspects of model checking and tabled logic programming. This side-by-side presentation exposes the primary rationale for the LMC project.

2.1 Model Checking

As remarked in the Introduction, model checking is essentially a problem of fixed-point computation. To substantiate this view, consider an example. Suppose we have a transition system T (transition systems are often used to model systems in the model-checking framework) and we wish to check if the start state of T satisfies the CTL branching-time temporal logic formula EFp. This will prove to be true just in case there is a run of the system in which a state satisfying the atomic proposition p is encountered.

Let S_0 be the set of states of T that satisfy EFp. If a state s satisfies p, written $s \vdash p$, then clearly it satisfies EFp. Further if s and t are states such that $t \vdash EFp$

and s has a transition to t, then $s \vdash EFp$ as well. In other words, if S is a set of states each of that satisfies EFp and $\mathbf{EFp} : 2^T \rightarrow 2^T$ is the function given by

$$\mathbf{EFp}(S) = \{s \mid s \vdash p\} \;\cup\; \{s \mid s \rightarrow t \;\wedge\; t \in S\}$$

then $\mathbf{EFp}(S) \subseteq S_0$. As a matter of fact, S_0 is the least fixed point of $\mathbf{EFp}$. Thus, one way to compute the set of states that satisfy EFp is to evaluate the least fixed point of the function $\mathbf{EFp}$. Assuming that T is finite-state, by the Knaster-Tarski theorem, it suffices to start with the empty set of states and repeatedly apply the function $\mathbf{EFp}$ till it converges.

There is a straightforward Horn-clause encoding of the defining equation of $\mathbf{EFp}$. Tabled LP systems can evaluate such programs efficiently, and hence yield an efficient algorithm for model checking EFp. This observation forms the basis for the XMC model checker (Section 3).

Other temporal logic properties of interest may involve the computation of greatest fixed points. For example, consider the CTL formula AGp asserting that p holds at all states and along all runs. The set of states satisfying this formula (which turns out to be the negation of the formula $EF\neg p$) is the greatest fixed point of the function $\mathbf{AGp}$ given by:

$$\mathbf{AGp}(S) = \{s \mid s \vdash p\} \;\cap\; \{s \mid s \rightarrow t \;\Rightarrow\; t \in S\}$$

Once again, using Knaster-Tarski, the set of states satisfying the property AGp may be computed by starting with the set of all states and repeatedly applying the function $\mathbf{AGp}$ till it converges.

More complicated properties involve the nesting of least and greatest fixed-point properties and their computation becomes a more complex nesting of iterations. The modal mu-calculus, the logic of choice for XMC, uses explicit least and greatest fixed-point operators and consequently subsumes virtually all other temporal logics in expressive power.

2.2 The XSB Tabled Logic Programming System

The fundamental theorem of logic programming [Llo84] is that, given a set of Horn clauses (i.e., a "pure" Prolog program), the set of facts derivable by SLD resolution (the computational mechanism of Prolog) is the same as the set of facts logically implied by the Horn clauses, which is the same as the set of facts in the least fixed point of the Horn clauses considered as set equations. From this result it might seem obvious that logic programming is well suited to solving general fixed-point problems and could be directly applied to model checking

This, however, is not the case. The foundational theorem on fixed points is weak in that it ensures only the existence of successful computations (SLD derivations); but there may be infinitely many computations that do not lead to success. This means that while a standard Prolog engine may be able to show that a particular fact is in the least fixed point of its rules, it can never show that a fact is not in the fixed point when there are infinitely many unsuccessful computation paths, which is the case for the fixed points needed for model checking. And even if the fact is in the fixed point, Prolog's search strategy may not find it. So even though the semantics of logic programming is a useful semantics for model checking, its computation mechanism of SLD is too weak to be practical.

The XSB system implements SLG resolution [CW96a], which to a first approximation can be understood as a tabled version of SLD resolution. This means that XSB can avoid rederiving a fact that it has previously tried to derive. (In the procedural interpretation of Horn clauses, this means that XSB will never make two calls to the same procedure passing the same arguments.) It is easy to see that for a system that has only finitely many possibly derivable facts, as for example in a finite-state model checking problem, XSB will always terminate.

To see how this works in practice, consider the following logic program:

```
reach(X,Y) :- trans(X,Y).
reach(X,Y) :- trans(X,Int), reach(Int,Y).
```

which defines reachability in a transition system. We assume that the predicate `trans(X,Y)` defines a transition relation, meaning that the system can make a direct transition from state X to state Y. Given a definition of the `trans` relation, these rules define `reach` to be true of a pair (X,Y) if the system can make a (nonempty) sequence of transitions starting in state X and ending in state Y. `trans` could be defined by any set of facts (and/or rules), but for our motivating example, we'll assume it is defined simply as:

```
trans(a,b).
trans(b,c).
trans(c,b).
```

Given a query of `reach(a,X)`, which asks for all states reachable starting from state `a`, Prolog (using SLD resolution) will search the tree indicated in Figure 1. The

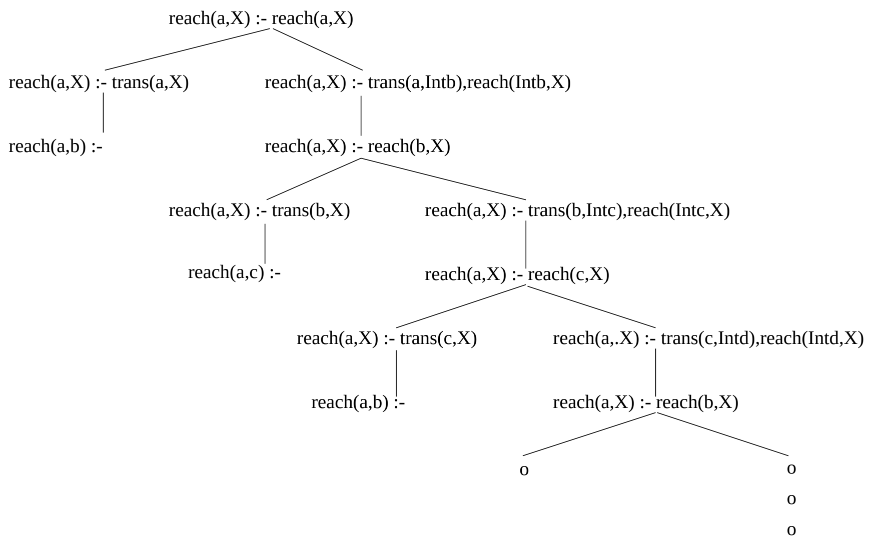

Fig. 1. Infinite SLD tree for reach(a,X)

atoms to the left of the `:-` symbols in the tree nodes capture the answers; the list of atoms to the right are waiting to be resolved away. Each path from the root to a leaf is a possible SLD derivation, or in the procedural interpretation of Prolog programs are computation paths through the nondeterministic Prolog program. Notice that the correct answers are obtained, in the three leaves. However, the point of more interest is that this is an infinite tree, branching infinitely to the lower right. Notice that the lower right node is identical to an ancestor four nodes above it. So the pattern will repeat infinitely, and the computation will never come back to say it has found all answers. A query such as `reach(c,a)` would go into an infinite loop, never returning to say that a is not reachable from c.

Now let us look at the same example executed using SLG resolution in XSB. The program is the same, but we add a directive `:- table reach/2.` to indicate that all subgoals of `reach` should be tabled. In this case during execution, an invocation of a `reach` subgoal creates a new subtree with that subgoal at its root if there is no such tree already. If there is such a tree, then the answers from that tree are used, and no new (duplicate) tree is created. Figure 2 shows the initial partial computation of the same query to the point where the subgoal `reach(b,X)` is about to be invoked, at the lower right node of that tree. Since there is no subtree for this subquery, a new

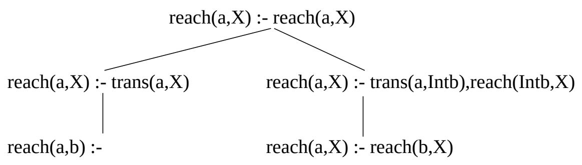

Fig. 2. Initial SLG subtree for reach(a,X)

one is created and computation continues with that one yielding another subtree, as shown in Figure 3. Now here again a new subgoal, `reach(c,X)`, is invoked, leading

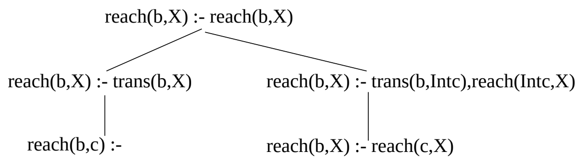

Fig. 3. Partial SLG subtree for reach(b,X)

to a new subtree, which is shown in Figure 4. Here again we have encountered a subgoal invocation, this time of `reach(b,X)`, and a tree for this subgoal already exists; it is in Figure 3. So no more trees are created (at least at this time.) Now we can use answers in the subtrees to answer the queries in the trees that generated them. For example we can use the answer `reach(c,b)` in Figure 4 to answer the query of `reach(c,X)` generated in the lower rightmost node of Figure 3. This results in another answer in Figure 3, `reach(b,b)`. Now the two answers in the tree for

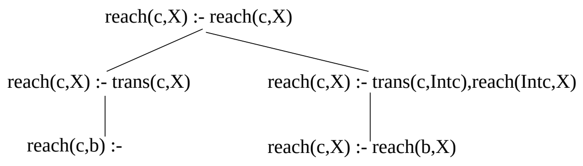

Fig. 4. Partial SLG subtree for reach(c,X)

`reach(b,X)` can be returned to the call that is the lower rightmost node of Figure 4, as well as to the lower rightmost node of Figure 2.

After all these answers have been returned, no new subgoals are generated, and the computation terminates, having reached a fixed point. The final state of the tree of Figure 2 is shown in Figure 5. The final forms of the other subtrees are similar.

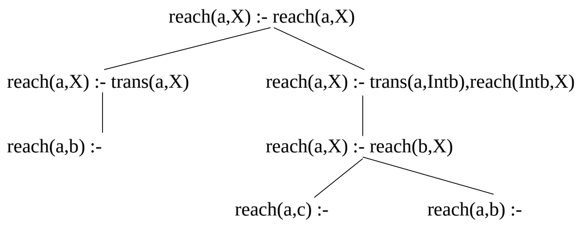

Fig. 5. The final SLG subtree for reach(a,X)

This very simple example shows how tabling in XSB terminates on computations that would be infinite in Prolog. All recursive definitions over finite sets will terminate in a similar way. Finite-state model checkers are essentially more complicated versions of this simple transitive closure example.

3 Model Checking of Finite-State Systems

In this section we present XMC, our XSB-based model checker for CCS and the modal mu-calculus. We first focus on the alternation-free fragment of the modal mu-calculus, to illustrate the strikingly direct encoding of its semantics as a tabled logic program. The full modal mu-calculus is treated next using a sophisticated semantics for negation developed within the logic-programming community. Finally, we show how the structural operational semantics of CCS, with full value-passing support, can also be naturally captured as a tabled logic program.

3.1 Model Checking the Alternation-Free Modal Mu-Calculus

The modal mu-calculus [Koz83] is an expressive temporal logic whose semantics is usually described over sets of states of labeled transition systems. We encode the logic in an equational form, the syntax of which is given by the following grammar:

$$\begin{aligned} F &\longrightarrow Z \mid \texttt{tt} \mid \texttt{ff} \mid\ F \vee F\ \mid\ F \wedge F\ \mid \texttt{diam(}A\texttt{,}\ F\texttt{)} \mid \texttt{box(}A\texttt{,}\ F\texttt{)} \\ D &\longrightarrow Z \texttt{ += } F \quad \text{(least fixed point)} \\ &\qquad \mid Z \texttt{ -= } F \quad \text{(greatest fixed point)} \end{aligned}$$

In the above, Z is a set of formula variables (encoded as Prolog atoms) and A is a set of actions; `tt` and `ff` are propositional constants; $\vee$ and $\wedge$ are standard logical connectives; and `diam(`A`, `F`)` (possibly after action A formula F holds) and `box(`A`, `F`)` (necessarily after action A formula F holds) are dual modal operators. For example, a basic property, the absence of deadlock, is expressed in this logic by a formula variable `deadlock_free` defined as:

```
deadlock_free -= box(-, deadlock_free) /\ diam(-, tt)
```

where the '`-`' in `box` and `diam` formulas stand for any action. The formula states, essentially, that from every reachable state (`box(-,deadlock_free)`) a transition is possible (`diam(-,tt)`).

We assume that the labeled transition system corresponding to the process specification is given in terms of a set of facts `trans(`*Src*`, `*Act*`, `*Dest*`)`, where *Src*, *Act*, and *Dest* are the source state, label and target state, respectively, of each transition. The semantics of the modal mu-calculus is specified declaratively in XSB by providing a set of rules for each of the operators of the logic, as follows:

```
models(State_S, tt).

models(State_S, (F1 \/ F2))  :- models(State_S, F1).
models(State_S, (F1 \/ F2))  :- models(State_S, F2).

models(State_S, (F1 /\ F2))  :- models(State_S, F1), models(State_S, F2).

models(State_S, diam(A, F))  :- trans(State_S, A, State_T),
                                models(State_T, F).

models(State_S, box(A, F))   :- findall(T, trans(State_S, A, T), States_L),
                                map_models(States_L, F).
```

Consider the rule for `diam`. It declares that a state `State_S` (of a process) satisfies a formula of the form `diam(A, F)` if `State_S` has an `A` transition to some state `State_T` and `State_T` satisfies `F`.

The semantics of logic programs are based on minimal models, and accordingly XSB directly computes least fixed points. Hence, the semantics of the modal mu-calculus's least fixed-point operator can be directly encoded as:

```
models(State_S, Z)           :- Z += F, models(State_S, F).
```

To compute greatest fixed points in XSB, we exploit its capability to handle normal logic programs: programs with rules whose right-hand side literals may be negated using XSB's `tnot`, which performs negation by failure in a tabled environment. In particular, we make use of the duality $\nu X.F(X) = \neg\mu X.\neg F(\neg X)$, and encode the semantics of greatest fixed-point operator as:

```
models(State_S, Z)           :- Z -= F, negate(F, NF),
                                tnot(models(State_S, NF)).
```

The auxiliary predicate `negate(F, NF)` is defined such that NF is a positive formula equivalent to (¬F).

For alternation-free formulas, the encoding yields dynamically stratified programs (i.e., a program whose evaluation does not involve traversing loops with negation), and has a two-valued minimal model. In [SSW96] it was shown that the evaluation method underlying XSB correctly computes this class of programs. Tabling ensures that each explored system state is visited only once in the evaluation of a modal mu-calculus formula. Consequently, the XSB program will terminate under XSB's tabling method when there are a finite number of states in the transition system.

3.2 Model Checking the Full Modal Mu-Calculus

Intuitively, the *alternation depth* of a modal mu-calculus formula [EL86] `f` is the level of nontrivial nesting of fixed points in `f` with adjacent fixed points being of different type. When this level is 1, `f` is said to be "alternation-free". When this level is greater than 1, `f` is said to "contain alternation." The full modal mu-calculus refers to the class of formulas of all possible alternation depths.

In contrast to the alternation-free fragment of the modal mu-calculus, when a formula contains alternation, the resultant XSB program is not dynamically stratified, and hence the well-founded model may contain literals with unknown values [vRS91]. For such formulas, we need to evaluate one of the stable models of the resultant program [GL88], and the choice of the stable model itself depends on the structure of alternation in the formula. Such a model can be computed by extending the well-founded model. When the well-founded model has unknown values, XSB constructs a *residual program* which captures the dependencies between the predicates with unknown values.

We compute the values of these remaining literals in the preferred stable model by invoking the stable model generator *smodels* [NS96] on the residual program. The algorithm used in *smodels* recursively assigns truth values to literals until all literals have been assigned values, or an assignment is inconsistent with the program rules. When an inconsistency is detected, it backtracks and tries alternate truth assignments for previously encountered literals. By appropriately choosing the order in which literals are assigned values, and the default values, we obtain an algorithm that correctly computes alternating fixed points.

Initial experiments indicate that XMC computes alternating fixed points very efficiently using the above strategy, even outperforming existing model checkers crafted to carry out the same kind of computation. Details appear in [LRS98].

3.3 On-the-Fly Construction of Labeled Transition Systems

The above encoding assumes that processes are given as labeled transition systems. For processes specified using a process algebra such as CCS [Mil89], we can construct the labeled transition system *on the fly*, using CCS's structural operational semantics. In effect, we can treat `trans` as a computed (IDB) relation instead of as a static (EDB) relation, without changing the definition of `models`. Below, we sketch how the `trans` relation can be obtained for processes specified in XL (a syntactically sugared version of value-passing CCS), the process specification language of XMC.

The syntax of processes in our value-passing language is described by the following grammar:

$$\begin{aligned} E &\longrightarrow PN \mid \texttt{in(}A\texttt{)} \mid \texttt{out(}A\texttt{)} \mid \texttt{code(}C\texttt{)} \mid \texttt{if(}C\texttt{, } E\texttt{, } E\texttt{)} \\ & \quad\; E \texttt{ o } E \mid E \texttt{ '||' } E \mid E \texttt{ \# } E \mid E \setminus L \mid E \texttt{ @ } F \\ \mathit{Def} &\longrightarrow (\texttt{PN ::= E})^* \end{aligned}$$

In the above, E is a process expression; PN is (parameterized) process name, represented as a Prolog term; C is a computation, (e.g., `X is Y+1`); Process `in(a(`t`))` inputs a value over port `a` and unifies it with term t; `out(a(`t`))` outputs term t over port `a`. Process `if(`C`, `E_1`, `E_2`)` behaves like E_1 if computation C succeeds and otherwise like E_2. Operator `o` is generalized prefixing. The remaining operators are like their CCS counterparts (modulo occasional changes in syntax to avoid clashes with Prolog lexicon). For example, `#` is nondeterministic choice; `'||'` is parallel composition; `@` is relabeling, where F is a list of substitutions; and '\' is restriction, where L is a list of port names. Recursion is provided by a set of *defining equations*, *Def*, of the form PN `::=` E.

The formal semantics of our language is given using structural operational semantics, closely paralleling that of CCS [Mil89]. Due to space limitations, we present here the axioms and inference rules for only a few key constructs. In order to emphasize the highly declarative nature of our encoding, these are presented exactly as they are encoded in the Prolog syntax of XSB.

```
trans(in(A), in(A), nil).
trans(out(A), out(A), nil).
trans(code(X), _, code) :- call(X).

trans(P1 o P2, A, Q) :- trans(P1, A, Q1),
                        (Q1 == code -> trans(P2, A, Q);
                            (Q1 == nil -> Q = P2 ; Q = Q1 o P2))).

trans(if(X, P1, P2), A, Q) :- call(X) -> trans(P1, A, Q) ; trans(P2, A, Q).

trans( P '||' Q,  A,  P1 '||' Q ) :- trans(P, A, P1).
trans( P '||' Q,  A,  P  '||' Q1) :- trans(Q, A, Q1).
trans( P '||' Q, tau, P1 '||' Q1) :- trans(P, A, P1),
                                     trans(Q, B, Q1), comp(A, B).

comp(in(A), out(A)).
comp(out(A), in(A)).

trans(P, A, Q) :-  P ::= R, trans(R, A, Q).
```

In the above, `A -> B ; C` is Prolog syntax for `if A then B else C`. The `trans` predicate is of the form `trans(P, A, Q)` meaning that process `P` performs an `A` transition to become process Q. The axiom for input says that in(A) can execute an in(A) transition and then terminate; similarly for the output axiom. The axiom for internal computation forces the evaluation of X and then terminates (without exercising any transition). The rule for generalized prefix states that P1 o P2 behaves like P1 until P1 terminates; at that point it behaves as P2. The conditional process if(X, P1, P2) behaves like P1 if evaluation of X succeeds, and like P2 otherwise. Finally, the rules for parallel composition state that P '||' Q can perform an autonomous A transition if either P or Q can (the first two rules), and P '||' Q can perform a synchronizing tau transition if P and Q can perform "complementary" actions (the last rule); i.e., actions of the form in(A) and out(A). The final rule above captures recursion: a process P behaves as the process expression R used in its definition.

To illustrate the syntax and semantics of XL, our value-passing language, consider the following specification of a channel chan (with input port get and output port give) implemented as a bounded buffer of size N.

```
chan(N, Buf) ::= code(length(Buf, Len)) o
                  if( (Len == 0)
                    , receive_only(N, Buf)
                    , if( (Len == N)
                        , send_only(N, Buf)
                        , receive_only(N, Buf) # send_only(N, Buf)
                  )).

receive_only(N, Buf) ::= in(get(Msg)) o chan(N, [Msg|Buf]).
send_only(N,Buf) ::= code(rm_last(Buf,Msg,RBuf)) o
                        out(give(Msg)) o chan(N,RBuf).
```

In the above definition rm_last(Buf,Msg,RBuf) is a computation, defined in Prolog, that removes the last message Msg from Buf, resulting in a new (smaller) buffer RBuf.

3.4 Implementation and Performance

The implementation of the XMC system consists of the definition of two predicates models/2 and trans/3; in addition, it contains a compiler to translate input XL representation to one with smaller terms that is more appropriate for efficient runtime processing. Overall the system consists of less than 200 lines of well-documented tabled Prolog code.

Preliminary experiments show that the ease of implementation does not penalize the performance of the model checker. In fact, XMC has been shown (see [RRR$^+$97]) to consistently outperform the Concurrency Factory's model checker [CLSS96] and virtually match the performance of SPIN [HP96] on a well-known set of benchmarks.

We recently obtained results from XMC on the i-protocol, a sophisticated sliding-window protocol used for file transfers over serial lines, such as telephone lines. The i-protocol is part of the protocol stack of the GNU UUCP package available from the Free Software Foundation, and consists of about 300 lines of C code.

Table 1 contains the execution-time and memory-usage requirements for XMC, SPIN, COSPAN [HHK96], and SMV [CMCHG96] applied to the i-protocol to detect a non-trivial livelock error that can occur under certain message-loss conditions. This livelock error was first detected using the Concurrency Factory.

Version	Tool	Completed?	Memory (MB)	Time (min:sec)
W=1 ˜fixed	COSPAN	Yes	4.9	0:41
	SMV	Yes	4.0	41:52
	SPIN	Yes	749	0:10
	XMC	Yes	18.4	0:03
W=1 fixed	COSPAN	Yes	116	24:21
	SMV	Yes	5.3	74:43
	SPIN	Yes	820	1:02
	XMC	Yes	128	0:46
W=2 ˜fixed	COSPAN	Yes	13	1:45
	SMV	No	28	>35 hrs
	SPIN	Yes	751	0:12
	XMC	Yes	68	0:11
W=2 fixed	COSPAN	Yes	906	619:49
	SMV	No	—	—
	SPIN	Yes	1789	6:23
	XMC	Yes	688	3:48

Table 1. i-protocol model-checking results.

Run-time statistics are given for window sizes $W = 1$ and $W = 2$, with the livelock error present (˜fixed) and not present (fixed). All results were obtained on an SGI IP25 Challenge machine with 16 MIPS R10000 processors and 3GB of main memory. Each individual execution of a verification tool, however, was carried out on a single processor with 1.8GB of available main memory.

As can be observed from Table 1, XMC performs exceptionally well on this demanding benchmark. This can be attributed to the power of the underlying Prolog data structuring facility (the i-protocol makes use of non-trivial data structures such as arrays and records), and the fact that data structures in XSB are evaluated lazily.

4 Beyond Finite-State Model Checking

In Section 3 we provided evidence that finite-state model checking can be efficiently realized using tabled logic programming. We argue here that tabled LP is also a powerful and versatile vehicle for verifying infinite-state systems. In particular, three applications of tabled logic programming to infinite-state model checking are considered. In Section 4.1, we show how an XMC-style model checker can be extended with compositional techniques. Compositional reasoning can be used to establish properties of infinite-state systems that depend only on some finite subpart. Section 4.2 treats parameterized systems. Finally, in Section 4.3, the application of tabled LP to real-time systems is discussed.

4.1 Compositional Model Checking

Consider the XL process `A o nil`. Clearly it satisfies the property `diam(A,tt)`. We can use this fact to establish that `(A o nil) # T` also satisfies `diam(A,tt)`, without consideration of `T`. This observation forms the basis for *compositional model checking*, which seeks to solve the model checking problem for complex systems in a modular fashion. Essentially, this is accomplished by examining the syntactic structure of processes rather than the transition relation. (Recall, that the XMC model checker, presented in Section 3, does exactly the latter in its computation of the predicate `trans/3`.) Besides yielding potentially significant efficiency gains, the compositional approach is intriguing also when one considers that the `T` in our example may well have been infinite-state or even undefined.

Andersen, Stirling and Winskel [ASW94] present an elegant compositional proof system for the verification of processes defined in a synchronous process calculus similar to Milner's SCCS [Mil83] with respect to properties expressed in the modal mu-calculus. A useful feature of their system is the algorithmic nature of the rules. The only nondeterminism in the choice of the next rule to apply arises from the disjunction operator in the logic and the choice of action in the process. Both of these sources of nondeterminism are unavoidable. In this respect, it differs from many systems reported in literature which require a clever choice of intermediate assertions to guide the choice of rules.

Andersen et al. in [ASW94] also present an encoding of CCS into their synchronous process calculus and consequently it is possible to use their proof system to verify CCS processes. This encoding, however, has two disadvantages. First, the size of the alphabet increases exponentially with the number of parallel components, and, secondly, the translation of the CCS parallel composition operator is achieved via a complex nesting of synchronous parallel, renaming, and restriction operators.

To mitigate the problems with their proof system in the context of CCS, we have adapted it to work directly for CCS processes under the restriction that relabeling operators use only injective functions. Our system retains the algorithmic nature of their system, yet incorporates the CCS parallel composition operator and avoids the costly alphabet blowup.

This adaptation is achieved by providing rules at three levels as opposed to two in [ASW94]. The first level deals with processes that are not in the scope of a parallel composition operator, the second for processes in the scope of a parallel composition operator, and the third for processes appearing in the scope of relabeling and parallel composition operators.

A Level-1 Rule:

```
models(P1 # P2, box(A,F)) :- models(P1, box(A,F)),
                             models(P2, box(A,F)).
```

A Level-2 Rule:

```
models((P1 # P2) '||'  P3, box(A,F)) :-
                       models(P1 '||' P3, box(A, F)),
                       models(P2 '||' P3, box(A, F)).
```

A Level-3 Rule:

```
models((B o P1) @ R '||' P2, box(A,F)) :-
                maps(R,B,C), models(C o (P1 @ R) '||' P2, box(A,F)).
```

Our system is sound for arbitrary processes and complete for finite-state processes. It has been implemented using XSB in the same declarative fashion as our XMC model checker. The compositional system is expected to improve on XMC's space efficiency by avoiding the calculation of intermediate states and by reusing subproofs, though worst-case behavior is unchanged. Performance evaluation is ongoing.

Our compositional system can indeed provide proofs for properties of partially defined processes as illustrated by the following example from [ASW94]. Let `p ::= (tau o p) # T` and `q ::= (tau o q) # T` where `T` is an unspecified process. The formula `x += box(tau, x)` expresses the *impossibility* of divergence. The following is a proof that `p '||' q` may possibly diverge.

```
models(p '||' q, x)
   ?- models(p '||' q, box(tau, x))
      ?- models((tau o p) # T) '||' q, box(tau, x))
         ?- models((tau o p) '||' q, box(tau, x))
            ?- models(p '||' q, x)
               ?- fail.
```

4.2 Model Checking Parameterized Systems using Induction

We have thus far described how inference rules for a variety of verification systems can be encoded and evaluated using XSB. These inference systems specify procedures that are primarily intended for model checking finite-state systems. We now sketch how more powerful deductive schemes offer (albeit incomplete) ways to verify properties of *parameterized systems*. A parameterized system represents an infinite family of finite-state systems; examples include an n-bit shift register and a token ring containing an arbitrary number of processes.

An infinite family of shift registers can be specified in XL as follows:

```
reg(0) :== bit
reg(s(N)) :== (bit @ [get/temp] || reg(N) @ [give/temp]) \ {temp}
bit :== in(get)  o out(give)  o bit
```

In the above specification, natural numbers are represented using successor notation (`0`, `s(0)`, `s(s(0))`, ...) and `reg(`n`)` defines a shift register with $n+1$ bits.

Now consider what happens when the query `models(reg(M),` ϕ`)`, for some nontrivial property ϕ and `M` unbound (thereby representing an arbitrary instance of the family), is submitted to an XMC-style model checker. Tabled evaluation of this query will attempt to enumerate all values of `M` such that `reg(M)` models the formula ϕ. Assuming there are an infinite number of values of `M` for which this is the case, the query will not terminate. Hence, instead of attempting to enumerate the set of parameters for which the query is true, we need a mechanism to derive answers that compactly *describe* this set.

For this purpose, we exploit XSB's capability to derive *conditional* answers: answers whose truth has not yet been (or cannot be) independently established. This mechanism is used already in XSB for handling programs with non-stratified negation under well-founded semantics. For instance, for the program fragment `p :- q. q :- tnot r. r :- tnot q.` XSB generates three answers: one for `p` that is conditional on the truth of `q`, and one each for `q` and `r`, both conditional on the falsity of the other. Now, if `r` can be proven false independently, conditional answers for `q` and `p` can be *simplified*: both can be marked as unconditionally true.

Our approach to model checking parameterized systems is to implement a scheme to uncover the inductive structure of a verification proof based on the above mechanism for marking and simplifying conditional answers. Consider, again, the query `models(reg(M),` ϕ`)`. When resolving this query, we will encounter two cases, corresponding to the definition of the `reg` family: (i) `M = 0`, and (ii) `M = s(N)`, corresponding to the base case and the recursive case, respectively. For the base case, the model checking problem is finite-state and the query `models(reg(0),` ϕ`)` can be completely evaluated by tabled resolution.

For the recursive case, we will *eventually* encounter a subgoal of the form `models(reg(N),` ϕ'`)`, where ϕ' is some subformula of ϕ. For simplicity of exposition, consider the case in which $\phi = \phi'$. Under tabled resolution, since this subgoal is a variant of one seen before, we will begin resolving answers to `models(reg(N),` ϕ`)` from the table of `models(reg(M),` ϕ`)`, and eventually add new answers to `models(reg(M),` ϕ`)`. This naive strategy leads to an infinite computation. However, instead of generating (enumerating) answers for `models(reg(N),` ϕ`)` for adding answers to `models(reg(M),` ϕ`)`, we can generate one conditional answer, of the form:

```
models(reg(M), φ) :- M = s(N), models(reg(N), φ).
```

which captures the dependency between the two sets of answers. In effect, we have evaluated away the finite parts of the proof, "skipping over" the infinite parts while retaining their structure. For instance, in the above example, we skipped over `models(reg(N),` ϕ`)` (*i.e.*, did not attempt to establish its truth independently), and retained the structure of the infinite part by marking the truth of `models(reg(s(M)),` ϕ`)` as conditional on the truth of `models(reg(N),` ϕ`)`. Using this mechanism, we are left with a *residual program*, a set of conditional answers that reflects the structure of the inductive proof. The residual program, in fact, computes exactly the set of instances of the family for which the property holds, and hence compactly represents a potentially infinite set.

Resolution as sketched above, by replacing instances of heads (left-hand sides) of rules by the corresponding bodies (right-hand sides) *unfolds* the recursive structure of the specification. In order to make the structure of induction explicit, it is often necessary to perform *folding* steps, where instances of rule bodies are replaced by the corresponding heads. In [RRRS98] we describe how tabled resolution's ability to compute conditional answers, and folding mechanisms can be combined to reveal the structure of induction.

It should be noted that although we have a representation for the (infinite) set of instances for which the property holds, the proof is not yet complete; we still need to show that the set of instances we have computed covers the entire family. What we have done is to simply evaluate away the finite parts, leaving behind the core induction problem, which can then be possibly solved using theorem provers. In many cases, however, the core induction problem is simple enough that the proof can be completed by heuristic methods that attempt to find a counter example, i.e., an instance of the family that is not generated by the residual program. For example, we have successfully used this counter-example technique to verify certain liveness properties of token rings and chains (shift registers), and soundness properties of carry look-ahead adders.

4.3 Model Checking Real-Time Systems

Another kind of infinite-state system we are interested in is *real-time systems*. Most embedded systems such as avionics boxes and medical devices are required to satisfy certain timing constraints, and real-time system specifications allow a system designer to explicitly capture these constraints.

Real-time systems are often specified as *timed automata* [AD94]. A timed automaton consists of a set of *locations* (analogous to states in a finite automaton), and a set of edges between locations denoting transitions. Locations as well as transitions may be decorated with constraints on *clocks*. An example of a timed automaton appears in Figure 6. A *state* of a timed automaton comprises a location and an assignment of values to clock variables. Clearly, since clocks range over infinite domains, timed automata are infinite-state automata.

Real-time extensions to temporal logics, such as timed extensions of the modal mu-calculus [ACD93, HNSY94, SS95], are used to specify the properties of interest.

Traditional model-checking algorithms do not directly work in the case of real-time systems since the underlying state-space is infinite. The key then is to consider only finitely many *regions* of the state space. In [AD94] it is shown that when the constraints on clocks are restricted to those of the form $X < Y + c$ where X and Y are clock variables and c is a constant, the state space of a timed automaton can be partitioned into finitely many *stable* regions—sets of indistinguishable elements of the state space.

For example, in the automaton of Figure 6, states $\langle L_0, t = 3\rangle$ and $\langle L_0, t = 4\rangle$ are indistinguishable. States $\langle L_0, t = 4\rangle$ and $\langle L_0, t = 6\rangle$, however, can be distinguished, since only from the latter can we make a transition to $\langle L_1, t = 6\rangle$, where an a-transition is enabled.

In [SS95], we presented a *local* algorithm for model checking real-time systems, where the finite discretization of the state space is done on demand, and only to the extent needed to prove or disprove the formula at hand. We can encode the essence of this algorithm with three predicates: `models/2`, which expresses when a region satisfies a timed temporal formula, `refinesto/2`, which relates a region with its partitions (obtained during finite discretization), and `edge/3`, which captures the transitions between regions.

Refinement adds a new inference rule to models:

$$\frac{R\ \texttt{refinesto}\ \{R_i\}, \forall i \quad R_i \vdash F}{R \vdash F}$$

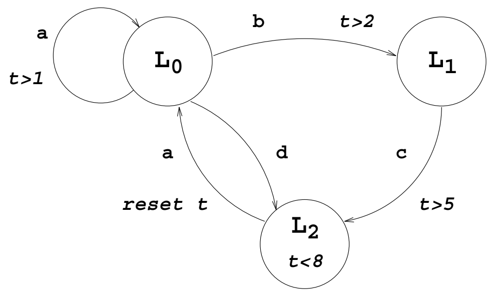

Fig. 6. Timed Automaton

which is captured by the following Horn clause:

```
models(R, F) :- refinesto(R, Refinements), map_models(Refinements, F).
```

Refinement creates new regions, and hence introduces new edges. The presence or absence of such edges may force further refinement. Therefore, `refinesto` and `edge` are mutually recursive predicates. Regions themselves are represented as a set of linear constraints, and operations on regions (such as splitting, which is needed when two points in a region can be distinguished) manipulate these constraints. Thus the resultant program is a *tabled* constraint logic program. Such programs can be evaluated in XSB using a meta interpreter, without modifying XSB's SLG-resolution engine. For better performance, however, we plan to directly augment the engine with a constraint-handling facility.

5 Conclusions

We have surveyed the current state of the LMC project, which seeks to use the latest developments in logic-programming technology to advance the state of the art of system specification and verification. In particular, the XMC model checker was discussed as well as several directions in which we are extending the LMC approach beyond traditional finite-state model checking.

Additional efficiency and ease-of-use issues are worthy of future investigation. First, since model checkers are specified at the level of semantic equations, equations of *abstract* semantics [CC77] can be encoded with equal ease. These can be used to incorporate process and formula abstractions, which have been used successfully to ameliorate state explosion in model checking [Dam96], into an LMC-style model checker. Secondly, the programmability of an LP system allows for direct encoding of traditional model-checking optimizations, such as partial order reduction [HPP96]. Finally, the high level at which model checking is specified correspondingly elevates the level at which erroneous system specifications can be diagnosed and debugged.

References

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *Information and Computation*, 104(1):2–34, 1993.

[AD94] R. Alur and D. Dill. The theory of timed automata. *Theoretical Computer Science*, 126(2), 1994.

[AH96] R. Alur and T. A. Henzinger, editors. *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[ASW94] H. R. Andersen, C. Stirling, and G. Winskel. A compositional proof system for the modal mu-calculus. In *Proceedings of the Ninth Annual IEEE Symposium on Logic in Computer Science*, pages 144–153, Paris, France, July 1994.

[CC77] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[CE81] E. M. Clarke and E. A. Emerson. Design and synthesis of synchronization skeletons using branching-time temporal logic. In D. Kozen, editor, *Proceedings of the Workshop on Logic of Programs,* Yorktown Heights, volume 131 of *Lecture Notes in Computer Science*, pages 52–71. Springer-Verlag, 1981.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CLSS96] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In Alur and Henzinger [AH96], pages 398–401.

[CMCHG96] E. M. Clarke, K. McMillan, S. Campos, and V. Hartonas-GarmHausen. Symbolic model checking. In Alur and Henzinger [AH96], pages 419–422.

[CW96a] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *Journal of the ACM*, 43(1):20–74, January 1996.

[CW96b] E. M. Clarke and J. M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys*, 28(4), December 1996.

[Dam96] D. Dams. *Abstract Interpretation and Partition Refinement for Model Checking*. PhD thesis, Eindhoven University of Technology, 1996.

[EL86] E. A. Emerson and C.-L. Lei. Efficient model checking in fragments of the propositional mu-calculus. In *Proceedings of the First Annual Symposium on Logic in Computer Science*, pages 267–278, 1986.

[GL88] M. Gelfond and V. Lifschitz. The stable model semantics for logic programming. In *International Conference on Logic Programming*, pages 1070–1080, 1988.

[HHK96] R. H. Hardin, Z. Har'El, and R. P. Kurshan. COSPAN. In Alur and Henzinger [AH96], pages 423–427.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model checking for real-time systems. *Information and Computation*, 111(2), 1994.

[HP96] G. J. Holzmann and D. Peled. The state of SPIN. In Alur and Henzinger [AH96], pages 385–389.

[HPP96] G. Holzmann, D. Peled, and V. Pratt, editors. *Partial-Order Methods in Verification (POMIV '96)*, DIMACS Series in Discrete Mathematics and Theoretical Computer Science, New Brunswick, New Jersey, July 1996. American Mathematical Society.

[JL87] J. Jaffar and J.-L. Lassez. Constraint logic programming. In *ACM Symposium on Principles of Programming Languages*, pages 111–119, 1987.

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

[Llo84] J. W. Lloyd. *Foundations of Logic Programming.* Springer, 1984.

[LRS98] X. Liu, C. R. Ramakrishnan, and S. A. Smolka. Fully local and efficient evaluation of alternating fixed points. In *Proceedings of the Fourth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '98), Lecture Notes in Computer Science.* Springer-Verlag, 1998.

[Mil83] R. Milner. Calculi for synchrony and asynchrony. *Theoretical Computer Science*, 25:267–310, 1983.

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

[NS96] I. Niemela and P. Simons. Efficient implementation of the well-founded and stable model semantics. In *Joint International Conference and Symposium on Logic Programming*, pages 289–303, 1996.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *Proceedings of the International Symposium in Programming*, volume 137 of *Lecture Notes in Computer Science*, Berlin, 1982. Springer-Verlag.

[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.

[RRRS98] A. Roychoudhury, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Tabulation-based induction proofs with applications to automated verification. In *Workshop on Tabulation in Parsing and Deduction*, 1998.

[SS95] O. Sokolsky and S. A. Smolka. Local model checking for real-time systems. In *Proceedings of the 7th International Conference on Computer-Aided Verification.* American Mathematical Society, 1995.

[SSW96] K. Sagonas, T. Swift, and D. S. Warren. An abstract machine to compute fixed-order dynamically stratified programs. In *International Conference on Automated Deduction (CADE)*, 1996.

[TS86] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *International Conference on Logic Programming*, pages 84–98. MIT Press, 1986.

[vRS91] A. van Gelder, K. A. Ross, and J. S. Schlipf. The well-founded semantics for general logic programs. *Journal of the ACM*, 38(3), 1991.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. 13th ACM Symp. on Principles of Programming*, pages 184–192, St. Petersburgh, January 1986.

[XSB98] XSB. The XSB logic programming system v1.8, 1998. Available from `http://www.cs.sunysb.edu/~sbprolog`.

This article was processed using the LaTeX macro package with LLNCS style