Efficient Model Checking Using Tabled Resolution*

Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan,
Scott A. Smolka, Terrance Swift, David S. Warren

Department of Computer Science
SUNY at Stony Brook
Stony Brook, NY 11794–4400, USA

Abstract. We demonstrate the feasibility of using the XSB tabled logic programming system as a programmable fixed-point engine for implementing efficient local model checkers. In particular, we present XMC, an XSB-based local model checker for a CCS-like value-passing language and the alternation-free fragment of the modal mu-calculus. XMC is written in under 200 lines of XSB code, which constitute a declarative specification of CCS and the modal mu-calculus at the level of semantic equations.
In order to gauge the performance of XMC as an algorithmic model checker, we conducted a series of benchmarking experiments designed to compare the performance of XMC with the local model checkers implemented in C/C++ in the Concurrency Factory and SPIN specification and verification environments. After applying certain newly developed logic-programming-based optimizations (along with some standard ones), XMC's performance became extremely competitive with that of the Factory and shows promise in its comparison with SPIN.

1 Introduction

Model checking [CE81, QS82, CES86] is a verification technique aimed at 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].

Model checking is the main verification technique deployed by the Concurrency Factory [CLSS96], NCSU Concurrency Workbench [CS96], SMV [CMCHG96], SPIN [HP96], and TempEst [JPO95] specification and verification environments. These tools use similar, but slightly different, system specification languages and property specification logics: the Concurrency Factory supports local model checking with partial order reductions in the alternation-depth-2 fragment of the modal mu-calculus for processes specified in a CCS-like value passing language; the NCSU Concurrency Workbench offers global model checking in the alternation-free modal mu-calculus for processes specified in pure CCS; SMV supports BDD-based symbolic model checking in CTL (with fairness) for a state-machine specification language;

* Research supported in part by NSF grants CDA–9303181, CCR–9404921, CCR–9505562, CDA–9504275, and AFOSR grants F49620-95-1-0508 and F49620-96-1-0087. Email correspondence: `sas@cs.sunysb.edu`

SPIN implements on-the-fly LTL model checking with partial order reductions for processes specified in Promela, a guarded-command language with buffered communication; and TempEst provides model checking of LTL with past operators for the Esterel synchronous language.

As is clear from this short list, there is a plethora of temporal logics and specification languages currently in use, and building a model checker for a particular combination involves having to confront the low-level computational details of the underlying model checking algorithm. One thing, however, these logics and process languages typically have in common is that their semantics are specified via structural recursion as fixed points of certain types of functionals.

To deal with the complexity of implementing model checkers, we would like ideally to focus only on declaratively specifying the semantics of the temporal logic and process language, while leaving the computational details to an efficient underlying engine. Such an ideal can be realized by the availability of engines that (i) compute fixed points, since from a computational viewpoint, model checking can be formulated in terms of computing fixed points, and (ii) are efficient enough to generate systems competitive with hand-crafted model checkers. One way to attain the second goal is for the fixed-point engine to provide *programmability*, so that optimizing program transformations can be made directly to the model checker specifications. Programmability also allows direct encoding of traditional model checking optimizations such as partial order reduction [HPP96].

Recent advances in *tabled resolution* [TS86, CW96a] offer significant promise towards achieving the above objective. At a high-level, tabled resolution augments Prolog-style SLDNF resolution for evaluating normal logic programs (with default negation). Tabled evaluation overcomes the three major limitations of Prolog, namely, weak termination, redundancy of computations, and weak semantics for negation. The XSB tabled logic programming system developed at SUNY Stony Brook is a practical embodiment of the power and enhanced functionality of tabled resolution.

When tabled resolution is used in XSB (by declaring particular predicates to be tabled), the system automatically maintains a table of predicate invocations and answers, using the table for all equivalent invocations after the first one. Many programs that would loop infinitely in Prolog will terminate in XSB because XSB calls a tabled predicate with the same arguments only once, whereas Prolog may call such a predicate infinitely often. For these terminating programs XSB efficiently computes the least model, which is the least fixed point of the program rules understood as "equations" over sets of atoms. More precisely, XSB is based on SLG resolution [CW96a], which computes queries to normal logic programs (containing default negation) according to the well-founded semantics.

This paper shows that by using XSB as a programmable fixed-point engine, one can construct an efficient model checker in under 200 lines of code. In particular, we have specified the syntax and semantics of a CCS-like value passing language similar to one supported by the Concurrency Factory, along with the syntax and semantics of the alternation-free fragment of the modal mu-calculus. The specification is based on a parallel constant-time reduction from the alternation-free modal mu-calculus to Datalog with negation presented in [ZSS94]. Not surprisingly, the XSB specification directly reflects the structural operational semantics of CCS and

the fixed-point semantics of the modal mu-calculus. The direct execution of these declarative specifications yields a local (on-the-fly) model checker, which we refer to as XMC.

In order to gauge the performance of XMC as an algorithmic model checker, we conducted a number of benchmarking experiments designed to compare the performance of XMC with the local model checkers implemented in the Concurrency Factory and SPIN. The model checking benchmarks we considered include Milner's "scheduler of cyclers" [Mil89] and the leader election and sieve algorithms from the SPIN benchmark suite.[2] After applying certain newly developed logic-programming-based optimizations (along with some standard ones—see Section 3), XMC's performance became extremely competitive with that of the Factory and shows promise in its comparison with SPIN.

These results, discussed further in Section 4, are somewhat surprising since the Factory's and SPIN's model checkers are written in low-level languages (C/C++) with the express purpose of temporal logic model checking, while XSB is a general-purpose logic programming system. Our experimental results provide evidence that writing efficient model checkers for various process languages and logics in XSB can be a viable idea.

Concerning related work, XMC can be viewed as an algorithmic model checker in a deductive setting (XSB, after all, is a system to deduce theorems from normal logic programs). The recent literature contains a number of proposals for combining deductive methods with algorithmic model checking techniques in order to prove temporal properties of concurrent systems. For example, the STeP system [BBC+96] combines the deductive methods of [MP95] with decision procedures for automatically checking the validity of a large class of first-order and temporal formulas. [PS96] uses deduction to establish an invariant that is then used to constrain the state space exploration performed in model checking. [RSS95] embeds a symbolic model checking decision procedure into the PVS higher-order prover, and [SUM96] employs first-order linear temporal-logic formulas to construct an abstract representation of the state space to be explored, and deductive methods to successively refine this representation until an answer to the model checking problem can be ascertained.

In other related work, [SHIR96] also uses Horn logic to specify model checking (for a basic, non-value-passing process specification language) but reports no effort to implement or evaluate this approach. Toupie [Rau95] is a mu-calculus interpreter that utilizes a combination of constraint logic programming (over finite domains) and BDDs to perform model checking. Constraint logic programming is also used in [Ost91] for semi-automatic verification of possibly infinite-state systems. In [SCK+95], an efficient "fixpoint-analysis machine" (FAM) is presented which can be used on a variety of fixed point computation problems, including model checking.

The structure of the rest of this paper is as follows. Section 2 describes our encoding in XSB of a value-passing language and the alternation-free modal mu-calculus. The logic-based optimizations we performed on the XMC model checker are the subject of Section 3, and our expermental results are discussed in Section 4. Section 5 concludes and presents directions for future work.

[2] The XMC and benchmark sources can be found at `http://www.cs.sunysb.edu/~ysr/xmc`.

2 Value-Passing Language and Modal Mu-Calculus Encoding

As described in the Introduction, we encoded in XSB a model checker for a CCS-like value-passing language and the alternation-free modal mu-calculus. The syntax of processes in our value-passing language is the following:[3]

```
P  ::=   X   |  a?Y  |  a!Y  |  P o P  |  if X then P else P
     |  P # P  |  P || P |  P\L  |  P@f  |  C(Z1,...,Zn)
```

In the above, `X` is a Prolog formula. Thus, for example, the formula `length(Buf, Len)` is the process that binds `Len` to the length of list `Buf`. Note that this process performs no actual transitions. Process `a?Y` inputs a value over port `a` into `Y`, where `Y` is a Prolog term. Similarly, `a!Y` outputs the value of `Y` over port `a`. Operator `o` is generalized prefixing. The remaining operators are like their CCS counterparts (modulo occasional changes in syntax to avoid clashes with Prolog lexicon). E.g., process `if X then P else Q` behaves like `P` if `X` succeeds and otherwise like `Q`; `#` is nondeterministic choice; `P||Q` is the parallel composition of `P` and `Q`; `@` is relabeling, where `f` is a list of substitutions represented as Prolog terms; and `C(Z1,...,Zn)` is a process constant `C`, parameterized by `Z1,...,Zn` (`C` and the `Zi` are Prolog atoms). Recursion is provided by *defining equations* of the form `C(Z1,...,Zn) :== P`.

The formal semantics of our language is given using structural operational semantics. Due to space limitations, we present here the axioms and inference rules for 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(a?Y, a?Y, nil).
trans(a!Y, a!Y, nil).
trans(X, nil, nil) :- call(X).

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

trans(if X then P1 else P2, Act_a, Q) :-
              call(X) -> trans(P1, Act_a, Q) ; trans(P2, Act_a, Q).

trans( P || Q, Act_a, P1 || Q ) :- trans(P, Act_a, P1).
trans( P || Q, Act_a, P  || Q1) :- trans(Q, Act_a, Q1).
trans( P || Q, tau,   P1 || Q1) :- trans(P, Act_a, P1),
                               trans(Q, Act_b, Q1), comp(Act_a, Act_b).
```

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, Act_a, Q)` meaning that process `P` performs an

[3] To increase readability, we use here a slightly sugared version of the syntax actually interpreted by XMC.

Act_a transition to become process Q. The axiom for input says that a?Y can execute an a?Y 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 then P1 else 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 Act_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 a?Y and a!Y.

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

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

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

Our encoding of the modal mu-calculus uses the following syntax for formulas:

```
F  ::=  Z  | tt | ff | F \/ F | F /\ F | diam(Act_a,F) | box(Act_a,F)
```

In the above, Z, which is a Prolog atom, is a mu-calculus logical variable; tt and ff are propositional constants; \/ and /\ are standard logical connectives; and diam(Act_a,F) (possibly after action Act_a formula F holds) and box(Act_a,F) (necessarily after action Act_a formula F holds) are dual modal operators. Additionally, logical variables can be given defining equations of the form X :== mu(F) (least fixed point) or X :== nu(F) (greatest fixed point). For example, a basic property, the absence of deadlock, is expressed in this logic by the formula Z :== nu(box(-,Z) /\ diam(-,tt)), where '-' stands for any action. The formula states, essentially, that from every reachable state (box(-,Z)) a transition is possible (diam(-,tt)).

As in the case of our value-passing language, 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. For example, the semantics of \/, diam, and mu are encoded as follows:

```
State_s |= F_1 \/  _       :- State_s |= F_1.
State_s |=  _  \/ F_2      :- State_s |= F_2.

State_s |= diam(Act_a, F) :- trans(State_s, Act_a, State_t),
                             State_t |= F.

State_s |= Z              :- Z :== mu(F), State_s |= F.
```

Consider the rule for `diam`. It states that a state `State_s` (of a process) satisfies a formula of the form `diam(Act_a,F)` if `State_s` has an `Act_a` transition to some state `State_t` and `State_t` satisfies `F`. As for `mu`, the semantics of logic programs are based on minimal models, and accordingly XSB directly computes least fixed points. Thus, the encoding of `mu` is straightforward as shown above.

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(F) = ¬ mu(¬F)`. The "not models" predicate `|/=` performs this negation, and appears below for the same sampling of operators as given above.

```
State_s |/= F_1 \/ F_2        :- State_s |/= F_1, State_s |/= F_2.

State_s |/= diam(Act_a, F)    :-
         tfindall(State_t, trans(State_s, Act_a, State_t), State_ts),
         lnotModels(State_ts, F).

lnotModels([],_).
lnotModels([State|LState], F) :- State |/= F, lnotModels(LState, F).

State_s |/= Z                 :- Z :== mu(F), tnot(State_s |= F).
```

where `tfindall` finds all solutions of a predicate in a tabled environment. The semantics of `nu` is then defined by:

```
State_s |= Z   :- Z :== nu(F), tnot(State_s |/= F).    % Nu -- models

State_s |/= Z  :- Z :== nu(F), State_s |/= F.      % Nu -- not models
```

This encoding provides a sound method for model checking any modal mu-calculus formula that is alternation free [EL86]. In the alternation-free case, fixed points are computed "inside out," with an inner fixed point computed before an outer fixed point in whose scope it lies. The proof of correctness rests on showing that the XSB program for model checking an alternation-free formula is *dynamically stratified* with respect to negation and to `tfindall/3`, and has a two-valued minimal model. Dynamic stratification ensures that the program's dynamic dependency graph can be evaluated without loops through negation. 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 Logic-Based Optimization Techniques

XMC benefits from optimization techniques developed for deductive-database-style applications, such as literal reordering and clause resolution factoring, as well as

from source-code representational changes of process terms.

Literal Reordering Literal reordering (see, for example, [Ull88]) is a common technique for optimizing computation by changing the order in which literals on the right hand sides of clauses are selected for resolution. The selection strategy controls the search space needed to evaluate the rule regardless of whether the rule is evaluated top-down or bottom-up. Consider the computation of `tau` transitions using `trans`:

```
trans(par(P, Q), tau, par(P1, Q1)) :- trans(P, Act_a, P1),
                           trans(Q, Act_b, Q1), compAct(Act_a, Act_b).
```

In general, the number of solutions to the predicate `compAct(Act_a, Act_b)` is much smaller than the number of solutions to `trans(Q, Act_b, Q1)`, and hence it pays to rewrite the rule as:

```
trans(par(P, Q), tau, par(P1, Q1)) :-    trans(P, Act_a, P1),
                           compAct(Act_a, Act_b), trans(Q, Act_b, Q1).
```

Clause Resolution Factoring Clause resolution factoring, introduced in [DRRS95], is a newer optimization that is geared specifically to deductive databases having a tightly linked top-down and bottom-up evaluation strategy. Clause resolution factoring shares elementary match and unification operations across program and answer clause resolution steps. An important aspect of this optimization is that it enables individual clauses (instead of whole predicates) to be tabled, improving the space and time efficiency of programs. Consider the following fragment of the `trans` relation:

```
:- table trans/3.
trans(C, Act_a, Q) :-  C :== P, trans(P, Act_a, Q).
trans(Act_a o P, Act_a, P).
trans(P1 # P2, Act_a, Q) :- trans(P1,Act_a,Q) ; trans(P2,Act_a,Q).
```

Observe that the rules for `trans` have structural recursion, and to ensure termination, only the recursive case (i.e., the first rule) needs to be tabled. Applying clause resolution factoring with this information results in the following fragment:

```
:- table trans_rec/3.
trans_rec(P, Act_a, Q) :- trans(P, Act_a, Q).

trans(C, Act_a, Q) :-  C :== P, trans_rec(P, Act_a, Q).
trans(Act_a o P, Act_a, P).
trans(P1 # P2, Act_a, Q) :- trans(P1,Act_a,Q) ; trans(P2,Act_a,Q).
```

In the above fragment, only `trans_rec` is tabled; this results in considerable savings in table space as well as computation time because terms which do not need to be tabled do not incur the overhead of tabling. The model checking predicate `models`, which also has structural recursion, is also subject to this optimization.

Optimizing Representation of Process States As mentioned in Section 2, processes are represented in XMC using a CCS-like value passing language. For instance in the six-agent sieve benchmark, a generator process and six tester processes communicate along a linear chain. As an XSB term, the sieve process might be specified in source code as:

```
sieve :==
  (gen || chan0 || test1 || chan1 || test2 || chan2 ||  test3 || chan3
       || test4 || chan4 || test5 || chan5 ||  test6 || chan6 || cons)  \
 {gen_out(X), in1(X), in2(X), in3(X), in4(X), in4(X), in5(X), in6(X), in7(X),
  con_in(X), out1(X), out2(X), out3(X), out4(X), out5(X), out6(X), out7(X)}.
```

XMC runtime states directly reflect this specification. For instance, a runtime state might have the form:

```
   (generator1(4,31) @ 1  || chan_1([]) @ 10 || tester2(2,4) @ 2
  || chan_1([3]) @ 11 || test2 || chan2 || test3 || chan3 || test4
  || chan4 || test5 || chan5 || test6 || chan6 || cons) \ 1
```

Taken as Prolog terms, these runtime states are relatively large. This situation is inefficient not only in terms of memory, but also in terms of time, since each state encountered must be checked against the table and inserted if it is not there. To reduce state size source code representation can be "folded" as below:

```
sieve :== (gen || chan0 || sieve1) \ {gen_out(X), ...,  out7(X)}.

sieve1 :==  (test1 || sieve2).   ...   sieve6 :==  (test6 || chan6).
```

This representation leads to smaller runtime terms such as

```
   (generator1(4,31) @ 1  || chan_1([]) @ 10 || tester2(2,4) @ 2
  || chan_1([3]) @ 11 || sieve2) \ 1
```

Optimizing process states leads to a 50% reduction in memory required to represent states in the leader and sieve examples of Section 4.

4 Experimental Results

In this section, we compare the performance of XMC with that of the Concurrency Factory and SPIN in terms of time and memory. Figure 1 shows the space and time used by XMC and the Factory on Milner's scheduler for the formula `Z :== nu(box(-,Z) /\ diam(-,tt))`, for a scheduler of n cyclers, $4 \leq n \leq 10$. The formula, which asserts the absence of deadlock, forces exploration of the entire state-space of the system, thus allowing us to assess the scalability of the two implementations. As is clear from the figure, XMC performs better than the Factory in terms of speed, and is quite competitive in terms of space.

The example of Milner's scheduler does not involve value-passing. For examples involving value-passing, we compared the performance of XMC with SPIN, rather than with the Factory, since the latter does not yet have efficient support for value-passing processes. (This problem is expected to be remedied in the next release of the Factory, slated for Fall '97.)

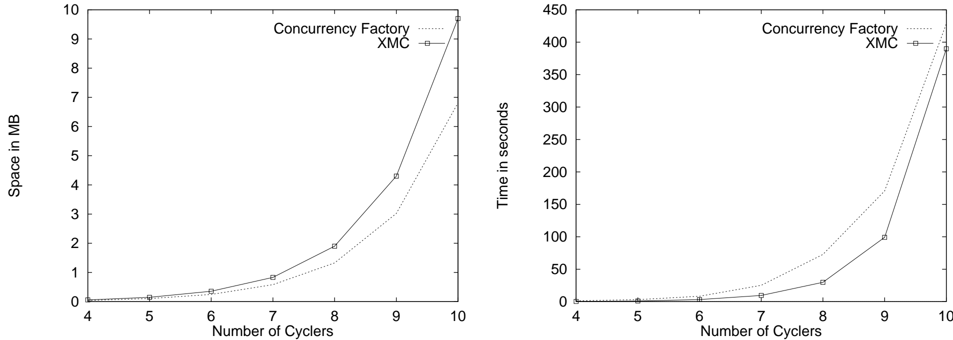

Fig. 1. Performance comparison with the Concurrency Factory on Milner's scheduler of cyclers.

To assess XMC's ability to model check in a value-passing language, and to assess the effects of the optimizations described in the previous section, we used the leader election example from the SPIN benchmark suite. As in the case of the scheduler, we benchmarked the system for several different ring sizes. The "leader5" system corresponds to the system used in the SPIN suite. Table 1 gives the space and time figures for two different formulas, `F1` being a least fixed point formula stating that in every run of the system a leader is eventually elected, and `F2` being a nested fixed point formula stating that in every run of the system at most one leader is elected. In this table, for a system of given size, the first line indicates the space and time figures with the naive encoding without any of the optimizations of the previous section, and the second line gives the corresponding figures with all the optimizations in place.

To compare XMC to SPIN we also implemented in XMC, a simple transitive closure algorithm to search and store all the reachable states of leader5 as well as sieve6, also from the SPIN benchmark suite. The results in Table 2 indicate that XMC has good memory usage as compared to SPIN, but that the speed of XMC appears uneven.[4] Two features account for the good memory usage of XMC. First, tabled terms in the underlying XSB engine are stored using a trie-like data structure that provides good structure sharing for variant subterms. Second, the scheduling strategy of XSB allows left-linear transitive closure to be performed using a minimum of runtime stack space.

An important feature of SPIN is that it combines on-the-fly model checking with *partial order reduction*, a technique for combating the combinatorial explosion that results from interleaving concurrent independent transitions in all possible orders. Roughly speaking, partial order reduction partitions the state space into equivalent search paths; (dis)proving a given property then requires exploring only one

[4] The sieve benchmark of Table 2 was run on an SGI challenge for both XMC and SPIN; SPIN results are from [GKPP97]. All other figures for XMC and the Concurrency Factory were performed on a sparc10 with about 500 MB available main memory; the leader benchmark for SPIN was also run on a sparc10 with 128 MB main memory [HP95].

Program	F1		F2	
	Time (sec)	**Space** (MB)	**Time** (sec)	**Space** (MB)
leader2 (unopt)	0.23	0.817	0.22	0.768
(opt)	0.10	0.209	0.11	0.198
leader3 (unopt)	1.21	4.593	1.18	4.342
(opt)	0.46	0.581	0.51	0.596
leader4 (unopt)	8.51	37.366	8.39	35.604
(opt)	2.93	3.079	3.23	3.239
leader5 (unopt)	39.09	170.608	37.51	163.405
(opt)	11.87	11.396	12.87	12.139

Table 1. Illustrating the effect of logic-based optimizations.

Program	**System**	**Time** (sec)	**Space** (MB)
leader5	SPIN	8.1	9.60
	XMC	5.5	0.78
sieve6	SPIN	1.8	2.31
	XMC	10.4	1.23

Table 2. Performance comparison with SPIN on value-passing examples.

path in each equivalence class. The results quoted above for SPIN (and for XMC) were obtained without the use of partial order reduction, and the numbers go down appreciably (especially for sieve) with the use of this technique. The programmability of XSB should, however, allow the implementation of partial order reduction techniques within XMC, a topic currently under investigation.

5 Conclusions and Future Work

We have provided experimental evidence that writing efficient algorithmic model checkers in a tabled logic programming system is a viable idea. Our work on XMC reveals a number of directions for future work. For example, we have not considered *alternating* fixed points [EL86] in this paper. The logic-programming-based approach to model checking, however, suggests a promising technique in which inner fixed points are computed *symbolically*, thereby avoiding their repeated computation.

Traditionally, model checking has been viewed as an *algorithmic* technique, although there is initial activity on combining model checking with *deductive* methods (see our discussion of related work in Section 1). Observe that (optimized) XSB meta-interpreters can be used to execute arbitrary deductive systems. Hence, the XSB-based approach offers a unique opportunity to fully and flexibly integrate algorithmic and deductive model checking. Demonstrating the feasibility of this idea is another direction for future work.

XSB's unification mechanism handles interpreted and uninterpreted variables in a value-passing language as ground and non-ground logical variables, respectively. Moreover, XSB automatically offers *lazy grounding* of variables. Since grounding can increase the search space of a query (every possible valuation must be considered), lazy grounding can result in substantial savings. As future work, we plan to experimentally measure the impact of lazy grounding on performance and investigate how it can be used to effectively realize Wolper's *data independence* scheme [Wol86].

References

[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.

[BBC+96] N. Bjørner, A. Browne, E. Chang, M. Colón, A. Kapur, Z. Manna, H. B. Sipma, and T. E. Uribe. STeP: Deductive-algorithmic verification of reactive and real-time systems. In Alur and Henzinger [AH96], pages 415–418.

[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.

[CS96] R. Cleaveland and S. Sims. The NCSU concurrency workbench. In Alur and Henzinger [AH96], pages 394–397.

[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.

[DRRS95] S. Dawson, C.R. Ramakrishnan, I.V. Ramakrishnan, and T. Swift. Optimizing clause resolution: Beyond unification factoring. In *International Logic Programming Symposium*, pages 194–208. MIT Press, 1995.

[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.

[GKPP97] R. Gerth, R. Kuiper, W. Penczek, and D. Peled. A partial order approach to branching time model checking. *Information and Computation*, 1997.

[HP95] G. J. Holzmann and D. Peled. An improvement in formal verification. In *Seventh Int. Conf. on Formal Description Techniques (FORTE '94)*, pages 177–194. Chapman and Hall, 1995.

[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 Theo-

retical Computer Science, New Brunswick, New Jersey, July 1996. American Mathematical Society.

[JPO95] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunications software. In Wolper [Wol95], pages 127–140.

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

[MP95] Z. Manna and A. Pnueli. *Temporal Verification of Reactive Systems: Safety.* Springer-Verlag, 1995.

[Ost91] J. S. Ostroff. Constraint logic programming for reasoning about discrete event processes. *Journal of Logic Programming*, 11(2/3):243–270, Oct./Nov. 1991.

[PS96] A. Pnueli and E. Shahar. A platform for combining deductive with algorithmic verification. In Alur and Henzinger [AH96], pages 184–195.

[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.

[Rau95] A. Rauzy. Toupie = μ-calculus + constraints. In Wolper [Wol95], pages 114–126.

[RSS95] S. Rajan, N. Shankar, and M. K. Srivas. An integration of model checking with automated proof checking. In Wolper [Wol95], pages 84–97.

[SCK$^+$95] B. Steffen, A. Classen, M. Klein, J. Knoop, and T. Margaria. The fixpoint-analysis machine. In I. Lee and S. A. Smolka, editors, *Proceedings of the Sixth International Conference on Concurrency Theory (CONCUR '95),* Vol. 962 of *Lecture Notes in Computer Science*, pages 72–87. Springer-Verlag, 1995.

[SHIR96] S. K. Shukla, H. B. Hunt III, and D. J. Rosenkrantz. HORNSAT, model checking, verification and games. In Alur and Henzinger [AH96], pages 99–110.

[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.

[SUM96] H. B. Sipma, T. E. Uribe, and Z. Manna. Deductive model checking. In Alur and Henzinger [AH96], pages 208–219.

[TS86] H. Tamaki and T. Sato. OLDT resolution with tabulation. In *Third Int'l Conf. on Logic Programming*, pages 84–98, 1986.

[Ull88] J. D. Ullman. *Principles of Data and Knowledge-base Systems, Volume I.* Computer Science Press, Rockville, MD, 1988.

[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.

[Wol95] P. Wolper, editor. *Computer Aided Verification (CAV '95)*, volume 939 of *Lecture Notes in Computer Science*, Liége, Belgium, July 1995. Springer-Verlag.

[ZSS94] S. Zhang, O. Sokolsky, and S. A. Smolka. On the parallel complexity of model checking in the modal mu-calculus. In *Proceedings of the 9th IEEE Symposium on Logic in Computer Science*, London, England, July 1994.

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