Tabulation-based Induction Proofs with Application to Automated Verification

Abhik Roychoudhury C.R. Ramakrishnan I.V. Ramakrishnan S. A. Smolka*

1 Introduction

XSB [14] is a *tabled* logic programming system designed to address shortcomings in Prolog's SLD evaluation mechanism for Horn programs. SLD's poor termination and complexity properties have rendered Prolog unsuitable for deductive database (DDB) and non-monotonic reasoning (NMR) applications. In contrast, XSB's implementation achieves a computationally tight integration of the logic programming (LP), DDB, and NMR paradigms.

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 [2], which computes queries to normal logic programs (containing default negation) according to the well-founded semantics.

Tabled resolution methods introduce a new level of declarativeness over traditional (Prolog-like) logic programming systems. Availability of tabled LP systems makes it feasible to develop a larger class of efficient declarative solutions to complex applications. One such application is *model checking* [3, 11, 4] which is a verification technique aimed at determining whether a system specification possesses a certain property expressed as a temporal logic formula. From a computational viewpoint, algorithmic model checking can be formulated in terms of fixed-point computations. By encoding the semantics of process languages and temporal logics as logic programs we can cast this computation at a high level into computing the minimal model of the logic programs. By using metaprogramming facilities of logic programming, one can implement deductive techniques and thereby integrate them tightly with algorithmic model checking. Our XMC system [12] aims to achieve such an integration.

XMC is an XSB-based model checker written in less than 200 lines of tabled Prolog code. In its current state, XMC can verify finite-state systems specified using value-passing CCS [9] and formulas expressed in modal mu-calculus [8]. XMC's space and time performance is competitive with hand-coded (in C/C++) model checkers such as the Concurrency Factory [5] and SPIN [6] from Bell Labs.

In this abstract we illustrate how tabled resolution, as realized in XSB, can be exploited to construct induction proofs. Our motivation for using tabulation-based induction stems from our desire to perform model checking on *infinite-state systems*. A common example of infinite-state systems are *parameterized* systems, such as an n-bit shift register or a sliding-window protocol having window size w and buffer capacity b. With finite-state model checking, one is limited to verifying only particular instances of such

*The authors are at : Dept. of Computer Science, SUNY at Stony Brook, Stony Brook, NY 11794-4400. E-mail : {abhik, cram, ram, sas}@cs.sunysb.edu

systems (such as a sliding-window protocol with window size 2 and buffer capacity 4). Verifying the entire (infinite) family of instances of a parameterized system requires model checking using induction.

In this abstract we focus only on the tabled logic programming aspects of doing induction. We briefly outline the application of these techniques to verification problems in Section 3.

2 Overview of Tabulation-based Induction

Induction can be "programmed" on top of a tabled resolution system such as XSB. The XSB system has a mechanism to compute *conditional answers* which mark computations whose truth or falsity have not yet been (or cannot be) established. This mechanism is used for handling programs with non-stratified negation under well-founded semantics. For instance, for the program fragment `p :- q. q :- not r. r :- not 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 marked as unconditionally true.

We can implement a scheme to uncover the inductive structure of a verification proof based on the above mechanism for marking and simplifying conditional answers. Let $p(n)$ be the predicate for verifying a property of the n-th instance of a family of systems. Assume that while attempting to evaluate $p(n)$ we encounter the subgoal $p(n-1)$. Note that the truth value of $p(n-1)$ cannot be independently established, since any attempt to do so is bound not to terminate. We can hence "skip over" $p(n-1)$ (*i.e.*, without establishing its truth), and mark $p(n)$ as conditional on the truth of $p(n-1)$. Using this mechanism to skip over the infinite parts of the computation, we will be 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. Preliminary exploration using this approach indicates that for many problems, the structure of the residual program is simple enough that we can complete the proof 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. A detailed description of our technique appears in [13].

2.1 An Example

We now illustrate our approach by proving the associativity of `append/3` predicate using tabulation-based induction.

Consider the predicate `prop/3` that expresses the associativity property of `append/3` (which concatenates two lists to yield a third list) defined by the following logic program:

```
append([ ], Y, Y).
append([X|Xs],Y,[X|Zs]) :- app(Xs, Y, Zs).

prop(X,Y,Z) :- append(X,Y,L1), append(L1,Z,L2), append(Y,Z,L3), append(X,L3,L2).
```

When we compute answers for the query `prop(X,Y,Z)` using (tabled) resolution, we attempt to *enumerate* all substitutions to variables `X`, `Y` and `Z` such that `prop(X,Y,Z)` is true. It is easy to see that evaluation of `prop(X,Y,Z)` terminates for particular values of `X`, `Y` and `Z`, whereas the query `prop(X,Y,Z)` with `X`, `Y` and `Z` not ground does not terminate. The core problem stems from the fact that `prop(X,Y,Z)` as well as some of the subgoals generated while resolving `prop(X,Y,Z)` have infinite number of answers, and any attempt to compute this set by enumeration is bound to fail.

However, instead of attempting to enumerate all answers to `prop(X,Y,Z)`, we seek simply to capture the *structure* of the set of answers— *i.e.*, dependencies between the different answers— as follows.

When the initial call to `prop(X,Y,Z)` is made, we do not know the structure of the answer space, and hence we perform program clause resolution. This results in the call `append(X,Y,L1)`. Resolution of this subgoal with the first clause defining `append/3` yields an answer `X = [ ], Y = L1`. After adding this answer to the answer table for `append(X,Y,L1)`, we explore the other alternative of resolving `append(X,Y,L1)` with the second clause of `append/3`. This leads to the call `append(Xs,Y,Zs)` such that `X=[X1|Xs], L1=[X|Zs]`.

Under normal tabled resolution, since a variant of `append(Xs,Y,Zs)` has been called before, we will resolve this subgoal using answer clause resolution, and generate the answer `Xs = [ ], Y=Zs` for `append(Xs,Y,Zs)`. Propagating this answer further, we will get the answer `X = [X1], L1=[X1|Y]` for the original call `append(X,Y,L1)`. This new answer can be be used to resolve `append(Xs,Y,Zs)` generating a third answer, and so on, resulting in an infinite computation. The infiniteness comes from the fact that we resolve `append(Xs,Y,Zs)` with answers from the table of `append(X,Y,L1)`, which in turn generates more answers to `append(X,Y,L1)`.

Instead of generating answers for `append(X,Y,L1)` by performing answer clause resolution of `append(Xs,Y,Zs)`, we simply "remember" the dependency between answers of `append(Xs,Y,Zs)` and `append(X,Y,L1)`. We generate a *conditional* answer for `append(X,Y,L1)` of the form that `X=[X1|Xs], L1=[X|Zs]` whenever `Xs`, `Y` and `Zs` are answers to `append(Xs,Y,Zs)`. Conditional answers can be computed using the delay and simplification mechanisms of SLG resolution— mechanisms that are used to compute well-founded models. In SLG terminology, then, the structure of the answer space is revealed by the *residual program* (set of conditional answers treated as clauses) when subgoals with potentially infinite number of answers are *delayed*.

Using this model of computation, we obtain the following residual program for the query `prop(X,Y,Z)`:

```
prop(X,Y,Z) :- X = [ ], append(Y,Z,_).
prop(X,Y,Z) :- X = [_|Xs], append(Xs,Y,L1), append(L1,Z,L2),
        append(Y,Z,L3), append(X,L3,L2).

append(X,Y,Z) :- X = [ ], Z = Y.
append(X,Y,Z) :- X = [X1|Xs], Z = [X1|Zs], append(Xs,Y,Zs).
```

Now, folding `append(Xs,Y,L1), append(L1,Z,L2), append(Y,Z,L3), append(X,L3,L2)` into `prop(Xs,Y,Z)` (from the definition of `prop/3` in the original program), and specializing `append(Y,Z,_)` as `app1(Y,Z)`, we obtain the program:

```
prop(X,Y,Z) :- X = [ ], app1(Y,Z).
prop(X,Y,Z) :- X = [_|Xs], prop(Xs,Y,Z).

app1(X,Y) :- X = [ ].
app1(X,Y) :- X = [_|Xs], app1(Xs,Y).
```

We have thus extracted the underlying structure of the answer space for `prop(X,Y,Z)`. Note that *no approximations have been performed* and hence the residual program is *equivalent* to the original definition of `prop/3`. Moreover, it should be noted that the transformations applied to the residual program are simple fold/specialize transformations that can be readily implemented in a tabling system.

The above program is equivalent to the "lemmas" generated by induction proof strategy for Prolog suggested in [7]. While in [7] the lemmas are considered simple enough to be proved by inspection, and hence the final step of the induction proof, we can in fact go one step further.

Note that for every induction proof, there is a specific domain over which the induction variables range. We can define the domain of induction variables also as a logic program. For instance, the variables `X`, `Y` and `Z` in `prop(X,Y,Z)` range over the domain of lists, which can be defined as:

```
list([ ]).
list([_|Xs]) :- list(X).
```

Now, we can reduce the induction problem to the problem of ascertaining whether the counterexample program (see below) has an empty model.

```
counter_example(X,Y,Z) :- list(X), list(Y), list(Z), not(prop(X,Y,Z)).
```

By applying fold/unfold transformations (under tabling, to avoid infinite application of these rules), we can, in many cases, reduce the counterexample program to one that finitely fails under tabled evaluation. Our induction proof succeeds when the counterexample predicate finitely fails. However, note that since the counterexample predicate *exactly* captures the set of counterexamples, tabled evaluation of the predicate may not terminate. In such situations, it is desirable to evaluate the predicate over an abstract domain in order to ensure termination.

The steps described above can be formalized as an algorithm. The details will appear in the full paper.

3 Application to Automated Verification

Algorithmic model checking in the XMC system is implemented in two parts using tabled logic programming. First, the semantics of the process language (value-passing CCS) is specified as a predicate `trans/3` that, given a process definition, evaluates the edge relation in the corresponding Labeled Transition System (LTS). Next, the semantics of the temporal logic (modal mu-calculus) is specified as a predicate `models/2` that, given a formula F in the logic and a state S (a vertex in the LTS), determines whether F holds in S. The predicate `trans/3` is used in the definition of `models/2`. The following is a fragment of the encoding of `trans/3` and `models/2` (see [12] for details):

```
:- table trans/3.  % Evaluate 'trans' relation using tabled resolution.
% Prefix:
trans(Act o P, Act, P).
% Choice:
trans(P + Q, Act, R) :- trans(P, Act, R).
trans(P + Q, Act, R) :- trans(Q, Act, R).
% ..  and so on for other operators

:- table models/2.  % Evaluate 'models' using tabled resolution.
models(S, and(F1, F2)) :- models(S, F1), models(S, F2).
models(S, diam(A,F)) :- trans(S, A, S1), models(S1, F).
```

The processes are specified using value-passing CCS, which augments Milner's CCS to enable communication of values (rather than atomic signals alone) over channels, and computation using these values. Parameterized systems as well as (infinite) families of systems can be encoded using value-passing CCS. For example, a FIFO channel with a fixed buffer size can be specified as:

```
chan(Buf, N, Limit) ::=
     if (N == 0)  then             input_only(Buf, N, Limit)
     else if (N == Limit) then     output_only(Buf, N, Limit)
     else     input_only(Buf, N, Limit) + output_only(Buf, N, Limit).

input_only(Buf, N, Limit) ::= in(input_chan(X)) o chan([X|Buf], N+1, Limit).
output_only(Buf @ [X], N, Limit) ::= out(output_chan(X)) o chan(Buf, N-1, Limit).
```

Now, the problem of verifying some property of the bounded buffer *for an arbitrary, albeit fixed, buffer size* becomes an instance of the model checking problem for infinite-state systems. With the encoding of CCS and modal mu-calculus semantics as a logic program, model checking a parameterized system reduces to evaluating a logic program with infinite answer sets. Applying tabulation-based induction techniques, we first evaluate away the finite parts of the process definition to expose the structure of induction. The induction proof is completed with the counter-example generation phase. If the counter-example program finitely fails, then the formula holds of every system in the infinite family. On the other hand, if the counter-example program finitely succeeds, not only do we have a disproof, but an exact scenario when the formula is false: an invaluable tool for "debugging" safety-critical systems. Moreover, tabulation-based induction is completely automatic, in contrast to systems such as PVS [10] which supply a suite of tools to assist the user derive induction proofs.

We have thus far used our method to verify safety and liveness properties of infinite families of systems using model checking, including the liveness of an n-bit shift register (fed with an infinite supply of input bits) and deadlock freedom of a token ring of n-processes. Since our method is a generic technique for generating induction proofs, it is also applicable to verification problems that do not use model checking. For instance, we have been able to verify the correctness of parameterized hardware circuits such as an n-bit carry-lookahead adder directly using tabulation-based induction. A detailed description of the verification problems as well as our solutions appear in [13].

References

[1] 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.

[2] W. Chen and D. S. Warren. Tabled evaluation with delaying for general logic programs. *JACM*, 43(1), 1996.

[3] 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.

[4] 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.

[5] R. Cleaveland, P. M. Lewis, S. A. Smolka, and O. Sokolsky. The Concurrency Factory: A development environment for concurrent systems. In Alur and Henzinger [1], pages 398–401.

[6] G. J. Holzmann and D. Peled. The state of SPIN. In Alur and Henzinger [1], pages 385–389.

[7] J. Hsiang and M. Srivas. Automatic inductive theorem proving using prolog. *TCS*, 54:3–28, 1987.

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

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

[10] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, and M. Srivas. PVS: Combining Specification, Proof checking and Model checking. In *Proceedings of the Seventh International Conference on Computer Aided Verification (CAV '96),* Vol. 1102 of *Lecture Notes in Computer Science*, pages 411–414. Springer-Verlag, 1996.

[11] 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.

[12] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, Terrance Swift, S.A. Smolka, and D.S. Warren. Effficient model-checking using tabled resolution. *Proceedings of CAV '97*, 1997.

[13] Abhik Roychoudhury, C. R. Ramakrishnan, I. V. Ramakrishnan, and S. A. Smolka. Automated verification of parametrized families using tabled logic programming. Technical report, Dept. of Computer Sc., State university of New York at Stony Brook, December 1997.

[14] XSB. The XSB logic programming system v1.7, 1997. Available by anonymous ftp from `ftp.cs.sunysb.edu`.