Justification based on Program Transformation
(Extended Abstract)

Hai-Feng Guo[1], C.R. Ramakrishnan[2], and I.V. Ramakrishnan[2]

[1] Computer Science Department
University of Nebraska at Omaha
Omaha, NE 68182-0116
Email: haifengguo@mail.unomaha.edu
[2] State University of New York at Stony Brook
Stony Brook, NY 11794, USA
Email: {cram,ram}@cs.sunysb.edu

Abstract. *Justification* is to give evidence, in terms of a proof, for the truth value of the result generated by query evaluation of a logic program. In an earlier work we presented algorithms for justifying logic programs. It was based on using tabling for evaluating the program and *post-processing* the memo tables created during evaluation. Justification built in this fashion does not compromise the performance of query evaluation in the sense that it is completely decoupled from the query evaluation process and is done only after evaluation is completed. However, this algorithm is based on meta-interpreting the memo tables and clauses of the program. This is a major source of inefficiency.

In this paper we present a new justification scheme based on program transformation. Justification of a true literal is generated during query evaluation of the transformed program. To justify a false literal, we generate the dual definition for each predicate defined in the program. A new program is then obtained based on those dual predicates and the original program. Query evaluation on this new program yields justification for the false literal. We provide experimental evidence to show that justification using program transformation outperforms the previous techniques in both efficiency and scalability.

1 Introduction

Justifying the truth value of a goal resulting from query evaluation of a logic program corresponds to providing evidence, in terms of a proof, for this truth. Justification plays a fundamental role in automatic verification, especially model checking [2]. Model checking is a problem of determining whether a system specification processes a property expressed as a temporal logic formula. Justification provides a proof if the property holds; otherwise it generates a counterexample showing where the violation occurs in the system. Justification also plays a useful role in applications other than verification. For instance it can be used for efficient generation of parse trees, synthesis controllers for embedded systems [8], etc.

In an earlier work, we had given algorithms [9, 3] for justification using a tabled logic programming system. The naturalness of using a tabled logic programming system for justification is that the answer tables created during query evaluation also serve as the witnesses supporting the result. Justifying the truth value of a goal resulting from query evaluation of a logic program corresponds to providing concise evidence, in terms of a proof, for this truth. Towards that end we presented algorithms for justifying such logic programs by post-processing the memo tables created during query evaluation. Justification in this post-processing fashion is "non-intrusive" in the sense that it is completely decoupled from query evaluation process and is done only after the evaluation is completed.

Justification in such a post-evaluation fashion is based on meta-interpreting the memo tables and clauses of the program. This is a major source of inefficiency because: (i) this meta-interpretation on clauses of the program can become significantly slower than the original query evaluation; (ii) cycle checking that is done to avoid self-explanation suffers from a quadratic time complexity. For example, consider the following tabled logic program:

```
:- table p/0.
p :- p.         p :- q.         q.
?- p.
```

In this case the justifier has to determine that the literal `p` is true due to literal `q` being true, instead of itself. To avoid cyclic explanations we have to maintain a history of the literals that have been used on the proof path. Prior to adding another literal to the proof we have to check if it already appears in the history.

In this paper we present a general justification technique based on program transformation [6]. First let us consider justifying true literals. The transformation adds an extra argument to predicate definitions. The idea is to capture the evidence in these arguments as a side-effect of query evaluation on the transformed program. However, this can cause the table space to explode for tabled literals. Consider a tabled logic program with cyclically defined clauses. A single answer can have infinite number of successful paths. To avoid this problem we do not introduce the extra argument in tabled predicates. Instead when an answer for the tabled call is inserted in the table the corresponding evidence for the answer is asserted. Evidences are accumulated in such a way that tabled predicates need no cycle checking.

Justification of false literals is much more difficult than that of true literals in logic programming systems. The main reason is that the evaluation of a false literal only returns its truth value – *false* without extra information. Therefore the idea of using an extra argument will not work in this case. Instead, we propose a new scheme to efficiently generate evidence for false literals. Our method is based on the completed definition [5] of general logic programs using two-valued logic. To justify a false literal, we generate the dual definition for each predicate defined in the program. A new program is obtained based on those dual predicates and the original program. Justifying a false literal in the original program amounts to justifying its dual is true in the transformed program.

To the best of our knowledge justification via program transformation is a new approach. Our justification scheme has been successfully applied for generation of witnesses and counterexamples in our μ-calculus model checker XMC [7]. If a formula with a universal path quantifier is false in a given system model, XMC will find a computation path which demonstrates that the negation of the formula is true. Likewise, when XMC determines that a formula with an existential path quantifier is true, it will find a computation path that demonstrates why the formula is true.

2 Justifying True Literals

2.1 Witnesses as Explicit Predicate Arguments

Logic programming makes it convenient to build a representation of witness by adding an extra argument whose instantiation becomes the witness. Justifying a true literal can be easily implemented as a side effect of query evaluation by recording its local explanation as a predicate argument explicitly. Consider a true literal A that unifies with A' in the clause A':-B such that $A'\theta = A$. Let A' be the predicate p/n. We can extend the definition of p/n with an extra parameter recording $B\theta$ as an evidence. The extension can be formally described with the following transformation rules:

$$p(t_1, ..., t_n). \Rightarrow p(t_1, ..., t_n, true).$$
$$p(t_1, ..., t_n) \text{ :- } L_1, ..., L_m. \Rightarrow p(t_1, ..., t_n, [L_1^{'}, ..., L_m^{'}]) \text{ :-} L_1^{'}, ..., L_m^{'}.$$

where $n \geq 0$, $m \geq 1$, $\forall 1 \leq i \leq m$, L_i is a user defined predicate $q(a_1, ..., a_k)$ $(k \geq 0)$, and $L_i^{'}$ is $q(a_1, ..., a_k, E_i)$ with E_i as the evidence for $q(a_1, ..., a_k)$.

2.2 Witnesses for Tabled Predicates As Implicit Arguments

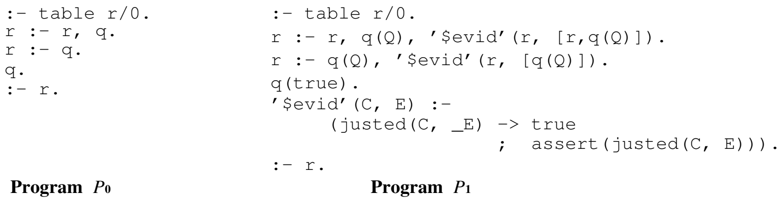
Witnesses as explicit arguments for tabled predicates can increase the table space explosively. In tabled Prolog system, such as XSB [10], calls to tabled predicates are stored in a searchable structure together with their proven instances. This collection of tabled calls paired with their answers is generally referred to as a *table*. To put witnesses as extra tabled predicate arguments results in recording witnesses as part of answers to tabled calls, which might make the global table space increased dramatically because there could be many explanations for a single answer in the original program. Therefore, a concise representation of witnesses for tabled predicates is required.

Witnesses for tabled calls can be recorded implicitly by asserting partial justification [3] results into a dynamic database. This partial justification only records the first witness for each tabled answer, that is, the witness when the answer is generated to be added into the table. Partial justifications are defined similar to complete justifications, except that the leaf nodes of a partial justification are either labelled `fail`, `fact`, or by other tabled literals, and all the interior nodes except the root are labelled by non-tabled literals. Partial justifications can be easily composed together to yield a complete justification for a literal. Informally composition amounts to "stringing" together the partial justifications of tabled literals at the leaf nodes labelled by those literals.

We now present our transformation algorithm for true justification.

```
algorithm Transform($C_1$ : old clause, $C_2$: new clause)
    let $C_1$ be $p(t_1, ..., t_n)$ :- $L_1, L_2, ..., L_m$.
    for $i \rightarrow 1$ to $m$ do
        if ($L_i$ is a user-defined non-tabled call) then
            let $L_i$ be a form of $q(a_1, ..., a_k)$
            set $B_i := q(a_1, ..., a_k, Q_i)$      (* $Q_i$ is a new variable *)
        else      (* $L_i$ is a tabled call or built-in system call *)
            set $B_i := L_i$
    if ($p/n$ is a tabled predicate) then
        if ($m = 0$) then      (* $C_1$ is a fact *)
            set $J$ := '\$evid'($p(t_1, ..., t_n), true$)
        else      (* $C_1$ is a rule *)
            set $J$ := '\$evid'($p(t_1, ..., t_n), [B_1, B_2, ..., B_m]$)
        set $C_2 := \{p(t_1, ..., t_n)$ :- $B_1, B_2, ..., B_m, J.\}$
    else      (* $p/n$ is a non-tabled predicate *)
        if ($m = 0$) then
            set $J := true$
        else
            set $J := [B_1, ..., B_m]$
        set $C_2 := \{p(t_1, ..., t_n, J)$ :- $B_1, B_2, ..., B_m.\}$
```

where the predicate ‘$\$evid$’$/2$, as shown in Figure 1, is a new predicate introduced to only record the partial justification for each tabled answer when the answer is generated the first time. Figure 1 illustrates a transformation example on a tabled logic program for true justification. Running the query `r` on the transformed program P_1 will return a truth result **`true`**, as well as the partial justifications asserted to support the result which is $[q(true)]$.



```
:- table r/0.
r :- r, q.
r :- q.
q.
:- r.
```

Program *P*0

```
:- table r/0.
r :- r, q(Q), ’$evid’(r, [r,q(Q)]).
r :- q(Q), ’$evid’(r, [q(Q)]).
q(true).
’$evid’(C, E) :-
     (justed(C, _E) -> true
                    ;  assert(justed(C, E))).
:- r.
```

Program *P*1

Fig. 1. Program Transformation for True Justification

2.3 Properties

Observe that in the transformed program we do not check for cyclic explanations. This is obvious for non-tabled predicates. For tabled predicates our strategy of only recording the first witness for each tabled answer will ensure that true justification can be constructed without having to check for cyclic explanations. The reason is this: if a witness for a tabled answer is asserted and a cyclic explanation for this tabled answer is therefore formed, then this tabled answer must not be new, which is contradictory to the assumption that only the first witness is recorded for each tabled answer.

Additionally, since witnesses are generated as arguments, whether explicitly for non-tabled answers or implicitly for tabled answers, justification has been integrated in query evaluation. Based on this observation, we have:

Proposition 1. *True justification time based on program transformation algorithm (as described in section 2.2) is linear in query evaluation time of the original program.*

3 Justifying False Literals

Justification of false literals is much more difficult than that of true literals in LP systems. Failed literals only return the truth value *false* without any tracing information. In this section, we present a transformation technique for justifying false literals. It is done in two steps. Suppose we want to justify that L is false. In the first step, a new literal $\overline{L}$, as the dual literal of L, is defined in the program; In the second step, the true literal $\overline{L}$ is justified as a side effect by applying the transformation technique described in Section 2.

3.1 Dual Predicates

Definition 1 (Dual Literal). *Given a literal L, the literal $\overline{L}$ is its dual iff:*

$$\overline{L} \leftrightarrow \sim L.$$

Definition 2 (Dual Predicate). *Let p/n be a predicate, where p is the predicate name and n is the arity of p. We say that the predicate $\overline{p}/n$ is its dual iff for any literal instance $p(t_1, ..., t_n)$, where $t_1, ..., t_n$ are terms*

$$\overline{p}(t_1, ..., t_n) \wedge p(t_1, ..., t_n) = \mathtt{f}alse \quad and \quad \overline{p}(t_1, ..., t_n) \vee p(t_1, ..., t_n) = \mathtt{t}rue.$$

Note that dual predicates can be automatically defined without modifying the declarative semantics of the source program; and dual predicates are defined in such a way that true justification of the dual literals can be easily constructed.

Our strategy to define the dual predicate is based on the *completed definition* [5] of a predicate. For simplicity of exposition we will illustrate the concepts in the transformation on the predicate $p/1$ whose k clauses $C_1, ..., C_k$ are defined as:

$$p(t_i) \text{ :- } L_{i,1}, L_{i,2}. \quad 1 \leq i \leq k$$

where $L_{i,1}$ and $L_{i,2}$ are two literals, and t_i is a term. Suppose that y_i is the only variable in clause C_i. Then, disregarding the order of literals in the body of the clause, each clause can be represented as:

$$\forall x \; p(x) \leftarrow \exists y_i((x = t_i) \wedge L_{i,1} \wedge L_{i,2})$$

where x is a variable not appearing in the clause and $x = t_i$ is an identity relation. We introduce the variable x to unify the definitions. The *completed definition* of $p/1$ is:

$$\forall x \; p(x) \leftrightarrow \bigvee_{i=1}^{k} \exists y_i((x = t_i) \wedge L_{i,1} \wedge L_{i,2})$$

which is consistent with the formula:

$$\forall x \; \overline{p}(x) \leftrightarrow \bigwedge_{i=1}^{k} \forall y_i((x \neq t_i) \vee \overline{L_{i,1}} \vee \overline{L_{i,2}})$$

Thus, the Horn clause definitions of $p/1$ can therefore be transformed into those of its dual predicate $\overline{p}/1$. Let $\overline{p_i}(x)$ be $\forall y_i((x \neq t_i) \vee \overline{L_{i,1}} \vee \overline{L_{i,2}})$, where $1 \leq i \leq k$. The dual predicate $\overline{p}/1$ is defined as:

$$\overline{p}(x) \text{ :- } \overline{p_1}(x), ..., \overline{p_k}(x).$$

And each $\overline{p_i}(x)$ is defined as:

$$\begin{aligned} &\overline{p_i}(x) \text{ :- } x \neq t_i. \\ &\overline{p_i}(x) \text{ :- } x = t_i, \overline{L_{i,1}}. \\ &\overline{p_i}(x) \text{ :- } x = t_i, forall(L_{i,1}, \overline{L_{i,2}}). \end{aligned}$$

where the literals $\overline{L_{i,1}}$ and $\overline{L_{i,2}}$ are dual predicates of $L_{i,1}$ and $L_{i,2}$ respectively, recursively defined using the same transformation scheme; and predicate $forall(L_{i,1}, \overline{L_{i,2}})$ means that for all the instances of $L_{i,1}$, $\overline{L_{i,2}}$ is always true. Note that for an undefined predicate, its dual is defined as a fact.

```
p :- q, s.
p :- r.
q.
:- p.
```

(a) Program *P*

```
np :- np1, np2.
np1 :- nq.
np1 :- q, ns.
np2 :- nr.
nr.
ns.
```

(b) Dual Definitions

```
np([np1(P1),np2(P2)]) :-
            np1(P1), np2(P2).
np1([nq(Q)]) :- nq(Q).
np1([q(Q), ns(S)]) :- q(Q), ns(S).
np2([nr(R)]) :- nr(R).
nr(true).      ns(true).
q(true).
:- np(X).
```

(c) Witnesses as arguments

Fig. 2. Two Step Transformation to Justify A False Literal

Figure 2 is an illustration of two step transformation to justify a false literal, where the new predicates `np, nq, ns, and nr` are the dual predicates of `p, q, s, and r` respectively. Given the query `:- np(X)`, we can support why p is false with the returned evidence: $[np1([q(true), ns(true)]), np2([nr(true)])]$.

3.2 Dual Definitions Adapted to Tabled Predicates

However, for a tabled logic program involving recursive definitions, the dual predicates defined in the above scheme may not work properly. A tabled call in logic programming system is supposed to be computed in a least fixpoint engine, however, its dual requires a greatest fixpoint computation. For example, consider a tabled logic program P shown in figure 3(a). The dual definitions generated from clause transformation are shown in figure 3(b), where the new predicates `np, nq and nr` are the dual predicates of `p, q and r` respectively. Observe that since there is a self-loop in the definition of `np` in the transformed program it has to be tabled. If we load this transformed program in tabled LP systems, such as XSB, the query `:- np` will return `false` because they only compute the least fixpoints. However, with greatest fixpoint computation, `np` is true.

```
:- table p/0.
p :- q, p.
p :- r.
q.
:- p.
```

(a) Program *P*

```
:- table np/0.
np :- np1, np2.
np1 :- nq.
np1 :- q, np.
np2 :- nr.
nr.
```

(b) Dual Definitions

```
np :- np1, np2.
np1 :- nq.
np1 :- q, \tnot p.
np2 :- nr.
nr.
```

(c) Table–based Dual Definitions

Fig. 3. Dual Definitions

We propose a solution to this problem as follows. We break the self-loop definition of `np` by using `\tnot p` instead in the body of its clauses (see Figure 3(c)). We add these definitions to the original program. Now we can use a tabled LP system to evaluate `:- np`. It will return the correct value true. Formally this modification to $\overline{p_i}(x)$ is:

$$\begin{array}{l} \overline{p_i}(x) \text{ :- } x \neq t_i. \\ \overline{p_i}(x) \text{ :- } x = t_i, L'_{i,1}. \\ \overline{p_i}(x) \text{ :- } x = t_i, forall(L_{i,1}, L'_{i,2}). \end{array} \quad \text{where } L'_{i,j} = \begin{cases} \backslash\texttt{+}\ L_{i,j} & \text{if } L_{i,j} \text{ is a built-in system call;} \\ \backslash tnot\ L_{i,j} & \text{if } L_{i,j} \text{ is a tabled literal;} \\ \overline{L_{i,j}} & \text{otherwise.} \end{cases}$$

$1 \leq j \leq 2$; `\tnot` and `\+` are two operations for tabled negation and Prolog negation respectively.

Definition 3 (Completed Form of A Logic Program). *Let P be a logic program. Its completed form, denoted as $comp(P)$, is a transformed program with:*

$$P \cup \{\textit{clauses of } \overline{p}/n \ \mid \ \forall\ p/n \textit{ in } P\}$$

Proposition 2. *Let P and $comp(P)$ be a logic program and its completed form respectively. Then for any false literal L in P, its dual literal $\overline{L}$ is true in $comp(P)$.*

Proposition 3. *Let P be a stratified logic program, its completed form $comp(P)$ is also a stratified logic program.*

3.3 Index Preserving Transformation

In logic programming systems unification is the dominant operation. Indexing on the appropriate argument of a predicate significantly speeds up unification time, since indexing can be used to quickly locate a match set of unifiable clauses based on the non-variable parts of the goal and clauses. We call this appropriate argument of a predicate an *indexed argument.*

Observe that dual predicate definitions introduce an extra unifying variable x in the head of the clauses. So indexing information that may have been present in the original clause is lost. However for well-indexed clauses in the original programs we show how to do the transformation without losing indexing.

Definition 4 (Well-Indexed Clauses). *Let p/n be a predicate. Let $C_1, ..., C_k$ $(k \geq 1)$ be all the clauses defined for p/n. We say $C_1, ..., C_k$ are well-indexed clauses iff for any pair of clause heads, their indexed arguments are not unifiable.*

Let us reconsider the predicate $p/1$ with k well-indexed clauses: $p(t_i)$:- $L_{i,1}, L_{i,2}$. $(1 \leq i \leq k)$. Note that t_m and t_n are not unifiable for $1 \leq m, n \leq k$ and $m \neq n$. Then, its dual predicate $\overline{p}/1$ is defined as:

$$\begin{array}{l} \overline{p}(t_i) \text{ :- } L'_{i,1}. \\ \overline{p}(t_i) \text{ :- } forall(L_{i,1}, L'_{i,2}). \\ \overline{p}(X) \text{ :- } X! = t_1,\ ...,\ X! = t_k. \end{array} \quad \text{where } L'_{i,j} = \begin{cases} \backslash\texttt{+}\ L_{i,j} & \text{if } L_{i,j} \text{ is a built-in system call;} \\ \backslash tnot\ L_{i,j} & \text{if } L_{i,j} \text{ is a tabled literal;} \\ \overline{L_{i,j}} & \text{otherwise.} \end{cases}$$

and $1 \leq j \leq 2$. Those dual definitions will preserve indexing.

4 Preliminary Experimental Results

The new justification scheme based on program transformation has been successfully applied on our XMC [7] model checking system. Model checking in XMC corresponds to evaluating a top-level query that denotes the temporal property of interest. The query succeeds whenever the system being verified satisfies the property. XMC system now returns a query result as well as witnesses or counterexamples to explain the success or failure of the query. The implementation was done using the XSB system.

Benchmark	**Metalock (liveness)**				**Sieve (ae_finish)**				**Leader (ae_leader)**	
Size	**(2,1)**	**(2,2)**	**(3,1)**	**(4,1)**	**7**	**8**	**9**	**10**	**5**	**6**
Evaluation	0.19	4.34	2.93	37.12	2.98	7.38	15.59	30.80	3.05	16.80
Old Justification	0.29	9.39	5.80	381.14	7.20	19.33	49.78	121.45	9.48	104.24
New Justification	0.30	6.24	4.16	89.46	3.18	7.88	16.92	32.48	3.19	17.27

Fig. 4. Time Performance (Seconds)

Figure 4 compares the time performance on model checking between our previous post-evaluation justification and the justification tool based on program transformation. The model checking examples used in these experiments (*metalock*, *sieve*, *leader*) were taken from the XMC collection. *Metalock* refers to a model of the Java Metalocking algorithm [1], *sieve* refers to a model of sieving algorithm to find a prime quickly, and *leader* refers a model of leader election algorithm. Both *sieve* and *leader* are adapted from SPIN's test suite [4].

The experimental evidence, shown in Figure 4, indicates that the new justification using program transformation outperforms the old one in both efficiency and scalability. The evaluation timings show the query evaluation performance on the original programs. The measurement results for old justification do not include the query evaluation time due to the fact that it is completely decoupled from query evaluation process, whereas those for the new scheme combine both evaluation and justification. Observe that the new justification has improved the efficiency significantly, especially as the model checking problems get big. For the example of `metalock(4,1)` (with 4 threads and 1 objects), the old justification is more than an order of magnitude slower than the time required for query evaluation (about ten times), while the new justification is only about twice as slow.

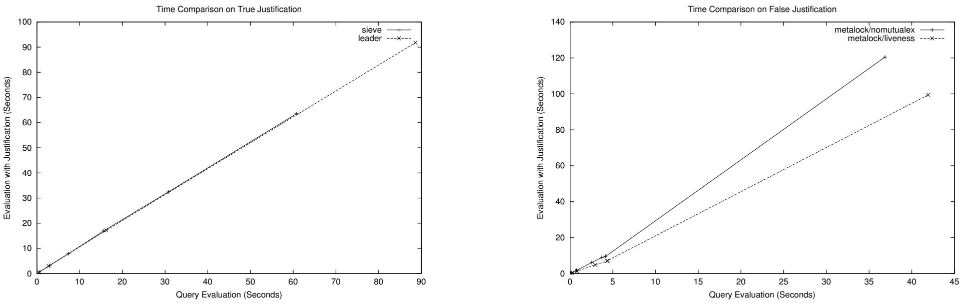


Fig. 5. Time Comparison between Evaluation and Transformed Evaluation

Based on program transformation, true justification as side effects incurs a small temporal overhead compared with the original query evaluation. Figure 5(a) shows a strict linear relation between the query evaluation timing and the new evaluation timing including justification. False justification based on duality is invoked when a query returns failure. False justification includes two steps: the original query evaluation and the corresponding justification of its negated literal. So false justification will require more time than the query evaluation time, as shown in Figure 5(b). But in practice this is not significant (slightly more than twice). Note that locating the false literal in each clause is very time consuming. Figure 5(b) shows the non-linearity between false justification time and original evaluation time.

5 Conclusion

In this paper, we presented a new justification scheme using program transformation. To justify a true literal, we extend the predicate definitions by including evidence as arguments, so that justification of a true literal can be obtained as a side effect of query evaluation. To justify a false literal, we generate the dual definition for each predicate defined in the program. A new program is then obtained based on those dual predicates and the original program. Justification showing why a literal is false is then reduced to a new program why its negated one is true. Our justification scheme provides a formal method to construct evidence, both witnesses and counterexamples, for verification systems.

References

1. S. Basu, S. A. Smolka, and O. R. Ward. Model checking the java meta-locking algorithm. In *Proceedings of 7th IEEE International Conference and Workshop on the Engineering of Computer Based Systems*, Edinburgh, Scotland, 2001.
2. E.M. Clarke, O. Grumberg, K.L. McMillan, and X. Zhao. Efficient generation of counterexamples and witnesses in symbolic model checking. In *Design Automation Conference*, pages 424–435, 1995.
3. Hai-Feng Guo, C.R. Ramakrishnan, and I.V. Ramakrishnan. Speculative beats conservative justification. In *International Conference on Logic Programming*, pages 150–165, 2001.
4. G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
5. J.W. Lloyd. *Foundations of Logic Programming, Second Edition.* Springer-Verlag, 1987.
6. A. Pettorossi and M. Proietti. Automatic derivation of logic programs by transformation. In *Lecture Notes for the 2000 European Summer School on Logic, Language, and Information*, 2000.
7. C.R Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, Y. Dong, X. Du, A. Roychoudhury, and V.N. Venkatakrishnan. Xmc: A logic programming based verification toolset. In *Computer Aided Verification*, 2000.
8. Parthasarathi Roop. Forced simulation: a formal approach to component based development of embedded systems, 2000. PhD thesis.
9. Abhik Roychoudhury, C. R. Ramakrishnan, and I. V. Ramakrishnan. Justifying proofs using memo tables. In *Principles and Practice of Declarative Programming*, pages 178–189, 2000.
10. XSB. The XSB logic programming system v2.3, 2001. Available by anonymous ftp from `www.cs.sunysb.edu/∼sbprolog`.