Justifying Proofs using Memo Tables

Abhik Roychoudhury
Dept. of Computer Science
SUNY Stony Brook
Stony Brook, NY 11794, USA
abhik@cs.sunysb.edu

C.R. Ramakrishnan
Dept. of Computer Science
SUNY Stony Brook
Stony Brook, NY 11794, USA
cram@cs.sunysb.edu

I.V. Ramakrishnan
Dept. of Computer Science
SUNY Stony Brook
Stony Brook, NY 11794, USA
ram@cs.sunysb.edu

ABSTRACT

Tableau-based proof systems can be elegantly specified and directly executed by a tabled Logic Programming (LP) system. Our experience with the XMC model checker shows that such an encoding can be used to search for the existence of a proof very efficiently. However, the users of a tableau system are often interested in getting sufficient evidence (in terms of the tableau proof rules) on why a proof does or does not exist. In this paper, we address the problem of constructing such an evidence without introducing *any* additional computational overhead to the proof search.

A tabled LP system maintains a memo table of "lemmas" that were tried and possibly proved during query evaluation. We propose the concept of *justifier* for extracting sufficient evidence for the truth or falsehood of literals in a logic program, by post-processing the memo tables created during query evaluation. Based on this logic program justifier, we show how to construct evidence for the presence/absence of tableau in a tableau-based proof system. We provide experimental results showing the effectiveness of the justifier in constructing succinct evidence of the evaluation performed by the XMC model checker. Finally we discuss the role of the justifier as a programming abstraction for encoding efficient algorithms as tabled logic programs.

Categories and Subject Descriptors

D.1 [**Programming Techniques**]: Logic Programming; D.2 [**Software**]: Software Engineering; D.2.5 [**Software Engineering**]: Testing and Debugging—*debugging aids*

1. INTRODUCTION

Tableau-based proof systems are used for deductive reasoning in a variety of computing applications, including automated theorem proving [14], and in specification and verification of temporal properties of concurrent systems [4, 29, 33]. Such systems are typically presented as a set of *proof rules*. Given a set of proof rules and a *goal* (which is a proof obligation), a tableau is a proof tree which is constructed by repeated application of the rules to the goal.

A successful tableau is a finite proof tree whose leaves represent empty goals. Thus, goals with a successful tableaux are in the least set closed under the application of the proof rules. Each proof rule is comprised of a (possibly empty) set of premises, side conditions and a conclusion, and can be readily encoded as a logic program. The least fixed point semantics of logic programs ensures that existence of a successful tableau for a goal can be checked using query evaluation (using a suitable resolution strategy) over the encoded program. The XMC model checker [27] shows that such a check can be done very efficiently as well.

Checking for existence of a tableau is only a part of the problem. It is often necessary to *construct* sufficient evidence to show the existence or absence of a tableau. This evidence may be used, for instance, to debug specifications that showed unexpected properties in a verification run. However, explicit construction of a tableau while searching for a proof can significantly slow down the proof system. In this paper, we describe techniques for reconstructing such evidence *after* evaluation of the query, using the results from evaluation itself. Beginning with a fundamental technique for constructing evidence for logic programs, we build a framework for presenting the evidence at the level of the high-level tableau rules themselves. Below, we give a brief introduction to tabled logic programming and its application to tableau construction using a non-trivial but short example drawn from verification of concurrent systems.

1.1 Encoding and Evaluating Tableau-Based Proof Systems: An Example

Figure 1(a) shows the proof rules of a tableau system for the *non-bisimilarity* relation between the states of two automata. The non-bisimilarity relation is the complement of the bisimulation relation in concurrency theory [23]. In a rule, the premises and conclusion appear above and below the horizontal line respectively while the side condition appears on its side.

The automata under consideration are labeled transition systems: transition from a state s to state s' on symbol a is represented by $s \stackrel{a}{\rightarrow} s'$. Given a pair of automata, the first rule says that state p in one automata is non-bisimilar to state q in the other automata (denoted by $p \not\approx q$) whenever there exists a transition $p \stackrel{a}{\rightarrow} p'$ and p' is non-bisimilar to every state q' such that $q \stackrel{a}{\rightarrow} q'$. The second rule says that non-bisimilarity is a symmetric relation.

The logic program encoding of this proof system is shown

PPDP '00, Montréal, Canada.

$$(1)\quad \frac{p' \not\approx q'_1, \ldots, p' \not\approx q'_n}{p \not\approx q} \quad \exists a\ p \xrightarrow{a} p' \wedge \{q'_1, \ldots, q'_n\} = \{q' \mid q \xrightarrow{a} q'\}$$

$$(2)\quad \frac{q \not\approx p}{p \not\approx q}$$

(a)

```
:- table nbisim/2.
nbisim(P, Q) :-
    trans(P, A, P1),
    forall(Q1, trans(Q,A,Q1),
           nbisim(P1,Q1)).
nbisim(P, Q) :- nbisim(Q, P).
```

(b)

Figure 1: Proof rules for *not-bisimilar* relation (a), and its encoding as a tabled logic program (b)

in Figure 1(b). In the program, we use the 3-ary `trans` relation to encode the labeled transition system. The least model of the logic program will contain `nbisim(p,q)` whenever the states `p` and `q` are not bisimilar. However, observe that if `p` and `q` are bisimilar then evaluation of the query using Prolog-style SLD resolution will not terminate since the second clause (encoding the symmetry rule) will produce an infinite calling sequence.

Tabled resolution techniques, *e.g.* OLDT [30] and SLG [6], avoid such infinite calling sequences by augmenting SLD strategy with *memo tables*. At a high level, a tabling system evaluates programs by recording subgoals (referred to as *calls*) and their provable instances (referred to as *answers*) in a table. Clause resolution, which is the basic mechanism for program evaluation, proceeds as follows. If the subgoal is already present in the table, then it is resolved against the answers recorded in the table; otherwise the subgoal is entered in the table and a new proof tree with this subgoal as the root is initiated. Answers to the subgoal are computed by resolving it against program clauses using SLD resolution, and are recorded in the table. Thus tabled evaluation of a logic program results in a forest of proof trees called the SLG forest [6]. (Figure 2(b) is the SLG forest generated by the query `:- nbsim(p,q)` for the automata in Figure 2(a).)

1.2 From Truth To Proof

The logic program encoding of the proof system is very concise. However, while it establishes the truth or falsehood of a goal, the logic programming system provides little or no information on why the conclusion was reached. This problem usually falls under the purview of debugging: using a trace based debugger and its navigation mechanisms (setting breakpoints or spy points, skips, leaps, etc.) to *trace* through the proof search itself. There are several salient problems with this approach.

1. A tracer displays the process of searching for the proof, and hence shows the exploration of unsuccessful as well as successful proof paths. In contrast, the user is often most interested in the final proof itself, rather than the manner in which the search was conducted.

2. The proof search strategy of Prolog, with its forward and backward evaluation, already makes tracing a Prolog execution considerably harder than tracing through procedural programs. The complex scheduling and fixed-point computing strategies of tabled resolution make this hard problem even worse.

3. Tracing repeats, at a slower pace, what the original execution did, and hence considerably degrades the performance of a proof system.

4. Trace-based debuggers provide no support for translating the results of the trace (which is at logic program evaluation level) to the problem space (e.g., tableau rule level).

Visual tools [5, 10, 31] can be used to graphically present the SLG forest and help alleviate the second problem. However, the other problems are fundamental to the approach of "watching the system prove a goal" and hence remain.

These limitations raise the following interesting questions. Can we *reconstruct* a proof/disproof for a goal after the evaluation for the goal is complete without reevaluating the goal? Can the reconstruction be done without impacting the performance of the initial evaluation? Can the reconstructed proofs be mapped to the original problem domain: e.g., to construct the non-bisimilarity tableau for the example in Figure 1(a)? In this paper we present techniques that answer the above questions in the affirmative.

1.2.0.1 Proofs by Justification:.

We propose the concept of *Justifier* for extracting proofs from the "footprints" of query evaluation left behind by the tabled logic programming engine. After query evaluation using a *tabled* logic programming system, the call (and answer) tables contain the lemmas that were tried (and/or proved). By inspecting the program text with these tables in hand, we can effectively reconstruct a proof (or sufficient evidence to show the lack of a proof) for a goal. Since we use precomputed results, we avoid searching for proofs through paths that were unsuccessful in the initial run. Furthermore, we collect the necessary evidence for presence or absence of a proof *independent of the proof search strategy.* Moreover, the information used for the reconstruction is already computed by the tabled evaluation engine and is available "for free"—i.e., without penalizing the original evaluation. Finally, the reconstruction is done by a logic program, and hence can be easily configured to map proof structures from the logic programming level to the level of the encoded problem.

1.3 Related Work

Pfenning investigated the idea of constructing proof objects in a proof system by evaluating an encoding of the proof system in a meta-language Elf [25]. Specifically, Elf is a Prolog like language whose search automatically constructs proofs *during* query evaluation.

For logic programs, a number of approaches to explain the results of query evaluation have been proposed in the literature. Algorithmic debugging techniques [28] explain the evaluation of a query by tracing the proof search performed by SLD resolution. Declarative debugging techniques [20, 24] assume a user-provided intended model of the given program and then attempt to explain the unexpected success/failure of a query by finding a program clause which is false in the intended model. Assertion based debugging techniques [18] perform program validation and debugging by static and dynamic checking of user-provided assertions

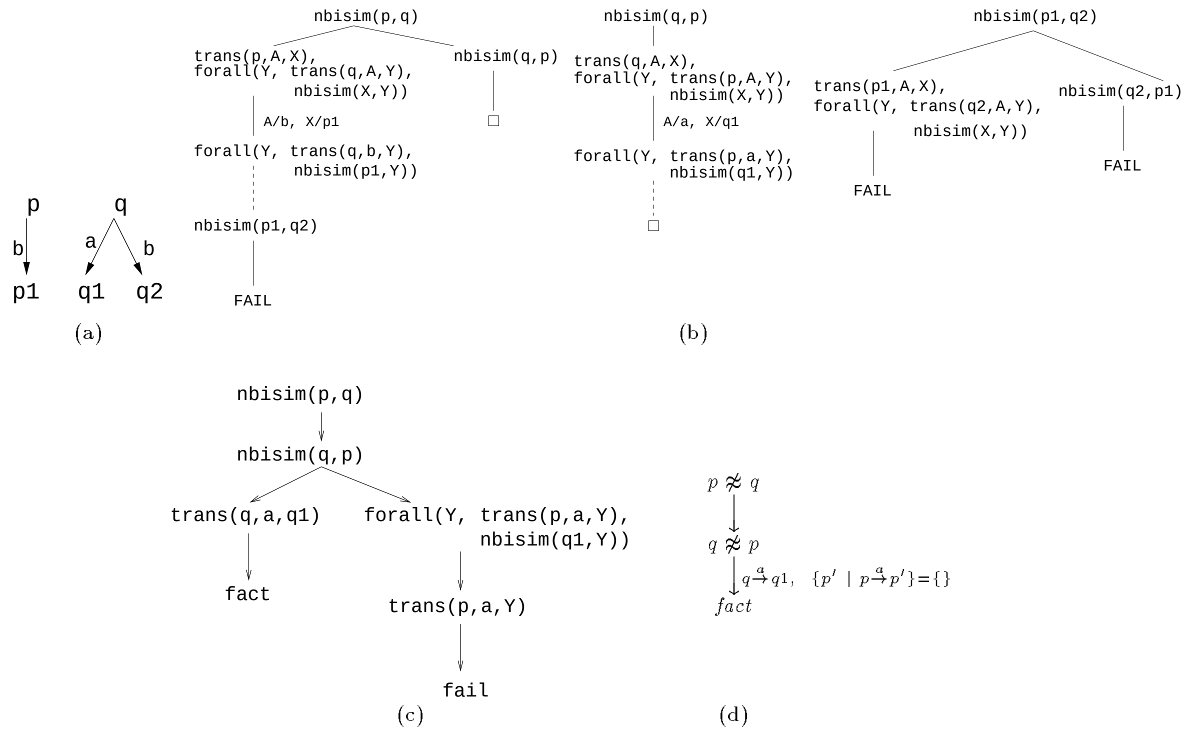

Figure 2: Justifying non-bisimilarity relation : (a) Two non-bisimilar automata (b) Fragment of SLG forest (c) Justification (d) Tableau extracted from justification

(which are essentially partial specification of the intended model of the program).

Although justification is similar in spirit to the above approaches in terms of their objectives it differs considerably from all them. First, it is done as a post-processing step after query evaluation, and not along with the query evaluation (as in algorithmic and assertion-based debugging) or before query evaluation (as in declarative and assertion-based debugging). Therefore rather than showing the entire proof search (as in algorithmic debugging), justification shows only those parts of the computation which led to the success/failure of the query. Moreover justification does not demand any creative input from the user regarding the intended model of the program. This is particularly useful when we have encoded a proof system as a logic program and we are constructing nontrivial proofs in the proof system via query evaluation. The intended model of the program might then be too hard, even impossible to be guessed by the user. For example in the context of model checking, the intended model of the program will contain information about which states of the concurrent system satisfy certain given temporal properties. Hence it is unclear how such techniques can be scaled from explaining logic programs to explaining proof systems encoded as logic programs.

In the context of deductive database programs, [2] explores the construction of explanations. These explanations consist of proof trees based on the underlying proof strategy. Recently, [21] presented the idea of providing explanations at several levels of abstraction. The explanations are constructed using the execution trace of the program. Note that justification also extracts proofs at different levels of abstraction. However, the information required for justification "comes for free" since they are available in the already constructed tables.

1.4 Summary of Results

1. Intuitively justification constructs sufficient evidence for the success or failure of a query to a tabled logic program. We formalize this intuitive concept and describe an efficient algorithm for extracting such a justification from tables created during query evaluation (Section 2).

2. We show how to derive an evidence for the existence or absence of a tableau in terms of tableau proof rules, based on the justification of the logic program that encodes the tableau system (Section 3).

3. We describe the construction of a evidence generator for a real-life model checking system (XMC) based on the justifier described in Section 2. We provide experimental results (in terms of sizes of proof structures) to demonstrate the practical utility of justifiers in model checking (Section 4).

4. The concept of justification forms a basis for a powerful programming abstraction. We discuss this issue in greater length in Section 5.

2. JUSTIFICATION OF LOGIC PROGRAM DERIVATIONS

In this section, we describe the fundamental aspects of constructing a structure, called a *justification* that explains the truth value of an answer computed by tabled resolution. For simplicity of exposition, we begin by defining justification of queries over definite logic programs. We discuss how the definition can be extended to evaluation of normal logic programs under well-founded semantics.

2.0.0.2 *Notational Conventions:.*

We use P to denote logic programs; $HB(P)$, $M(P)$, $\hat{M}(P)$ to denote the Herbrand Base and least Herbrand model and perfect model of P [9] respectively; A and B to denote atoms or literals; α to denote a set of atoms or literals; β to denote a conjunction of atoms (a goal is a conjunction of atoms) or literals; θ to denote substitutions; '$\succeq$' to denote atom subsumption ($A \succeq B$ for A subsumes B); and C to denote a clause in a program. For a binary relation R, we denote its (reflexive) transitive closure by R^*. □

Before describing justification, we need to introduce some preliminary notation for capturing the truth assignments computed by tabled resolution. The tables at the end of the resolution are denoted by $T = T_C \cup T_A$, where T_C are the set of atoms stored in call (or subgoal) tables and T_A are the set of atoms stored in return (or answer) tables.

Definition 1 (Truth Assignment). *The truth assignment of atom A wr.t. tables T, denoted $\tau(A,T)$, is:*

$$\tau(A,T) = \begin{cases} \mathit{true} & A \in T_A \\ \mathit{false} & A \notin T_A \wedge \exists A' \in T_C \; A' \succeq A \\ \mathit{uncomputed} & \mathit{otherwise} \end{cases}$$

We drop the parameter T and write the truth assignment as $\tau(A)$ whenever the tables are obvious from the context. By soundness of tabled resolution, note that when tables T result from resolving a query over a program P, $\tau(A,T) = \mathit{true} \implies \forall\theta A\theta \in M(P)$ and $\tau(A,T) = \mathit{false} \implies \forall\theta A\theta \notin M(P)$ for all atoms A.

2.1 Structure of Justification

Let A be an answer to some query in program P, i.e., $\tau(A) = \mathit{true}$. We can complete one step in explaining this answer by finding a clause C such that (i) A unifies with the head of C, and (ii) each literal B in the body of C has $\tau(B) = \mathit{true}$. If $\tau(A) = \mathit{false}$, we can explain this failure by showing that for all clauses C whose heads unify with A, there is at least one literal B in C such that $\tau(B) = \mathit{false}$. We call such one-step explanations as a *locally consistent explanations.*

Definition 2. *Locally consistent explanation for an atom A w.r.t. program P and table T, denoted by $\xi_{(P,T)}(A)$ is a set of sets of atoms s.t.*

1. *If $\tau(A) = \mathit{true}$:*

   $\xi_{(P,T)}(A) = \{\alpha_1, \alpha_2, \ldots, \alpha_m\}$, with each α_i being a set of atoms $\{B_1, B_2, \ldots, B_n\}$ such that:

   (a) $\forall\ 1 \le j \le n\ \tau(B_j) = \mathit{true}$, and

   (b) $\exists\ C \equiv A' {:-}\ \beta$ and a substitution θ such that $A'\theta = A$ and $\beta\theta \equiv (B_1, B_2, \ldots, B_n)\theta$.

2. *If $\tau(A) = \mathit{false}$:*

   $\xi_{(P,T)}(A) = \{L\}$, a singleton collection where $L = \{B_1, B_2, \ldots, B_n\}$ is the smallest set such that

   (a) $\forall 1 \le j \le n\ \tau(B_j) = \mathit{false}$, and

   (b) $\forall$ substitutions θ and $C \equiv A' {:-}\ (B'_1, B'_2, \ldots, B'_l)$, $A'\theta = A\theta \implies \exists 1 \le k \le l$ such that $B'_k\theta \in L$ and $\forall\ 1 \le i < k\ \tau(B'_i\theta) = \mathit{true}$.

(a)	(b)

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

Figure 3: Example programs

(a)	(b)
$\xi(\mathrm{p}) = \{\ \{\mathrm{p}\}, \{\mathrm{q}\}\ \}$ $\xi(\mathrm{q}) = \{\ \{\}\ \}$	$\xi(\mathrm{p}) = \{\ \{\mathrm{q}\}\ \}$ $\xi(\mathrm{q}) = \{\ \{\mathrm{p},\mathrm{r}\}\ \}$ $\xi(\mathrm{r}) = \{\ \{\}\ \}$

Figure 4: Locally consistent explanations [(a) and (b)] for example programs in Figure 3(a) and (b)

We write $\xi_{(P,T)}(A)$ as $\xi(A)$ whenever the program P and table T are clear from the context. In the above definition for $\xi(A)$ such that $\tau(A) = \mathit{true}$ (case 1), the second condition 1(b) states that an explanation in the collection forms an instance of an r.h.s. of a clause C whose head unifies with A. The first condition ensures that all atoms in an explanation have a truth assignment of *true*. When $\tau(A) = \mathit{false}$ (case 2), the two conditions 2(a) and 2(b) ensure that for every clause C whose head unifies with A (under substitution θ), there is a literal B_k on the r.h.s. of C such that $B_k\theta$ has truth assignment *false*, and every earlier literal in C has truth assignment *true*. The restriction of L to be the smallest such set ensures that L contains only those $B_k\theta$ that are specified by condition 2(b).

From Definition 2, and the soundness of tabled resolution, it follows that if A is used in resolution then $\tau(A)$ coincides with the truth values of all atoms in the sets in $\xi(A)$.

Theorem 3 (Soundness of ξ). *Let P be a program and T the tables after resolution of some query to P. Then $\forall A\ \tau(A) \in \{\mathit{true}, \mathit{false}\} \implies \forall\ L \in \xi(A)$, $B \in L$ and ground substitutions θ, $B\theta \in M(P) \iff A\theta \in M(P)$.*

Observe that, for an atom A, the different sets in the collection $\xi(A)$ represent different consistent explanations for the truth or falsehood of A. For instance, consider the programs in Figure 3(a) and the corresponding ξ's in Figure 4(a). That $\xi(\mathrm{p})$ contains $\{\mathrm{q}\}$ means that the truth of q alone is sufficient to (locally) explain the truth of p. In contrast, for the program in Figure 3(b), $\xi(\mathrm{q})$ contains $\{\mathrm{p},\mathrm{r}\}$ which indicates that to explain the falsehood of q, one needs to explain the falsehoods of both $\{\mathrm{p}\}$ and $\{\mathrm{r}\}$. In this sense, one can view the "set of sets" representation of locally consistent explanations as an encoding of the dependencies in disjunctive normal form.

An answer A is explained by answers $\{B_1, B_2, \ldots, B_k\}$ in $\xi(A)$ and then (recursively) explaining each B_i. The explanation can be captured by a graph, whose edges are determined by locally consistent explanations. When tabled resolution finds that an answer A has $\tau(A) = \mathit{true}$, then

$$\begin{array}{c} \texttt{p} \\ \downarrow \\ \texttt{q} \\ \downarrow \\ \texttt{fact} \end{array} \qquad \begin{array}{c} \texttt{p} \\ \downarrow \\ \texttt{q} \\ \swarrow \quad \searrow \\ \texttt{ancestor} \quad \texttt{r} \\ \qquad\qquad \downarrow \\ \qquad\qquad \texttt{fail} \end{array}$$

(a) (b)

Figure 5: Justification of p evaluated w.r.t. programs in Figures 3(a) and (b), respectively

clearly there is a finite sequence of locally consistent explanations that lead to fact (i.e. an atom B such that $\{\} \in \xi(B)$ and $\tau(B) = \mathit{true}$). We mark such a conclusion by using a special node labeled '`fact`'.

Note that not all sequences of locally consistent explanations may be finite, even for A such that $\tau(A) = \mathit{true}$. For instance, consider the explanation sequences for query p over program in Figure 3(a). There is an infinite sequence since $\{\mathtt{p}\}$ is in $\xi(\mathtt{p})$. However, such cycles represent "unfounded" proof paths and hence do not explain why $\tau(p) = \mathit{true}$. Hence, we develop a stronger characterization of what constitutes a *justification*. Before formally defining this notion, we develop a similar intuition for justification of false literals. For a goal A with $\tau(A) = \mathit{false}$, there are two distinct ways in which tabled resolution reaches this conclusion:
1. there are no clause heads that can unify with the given goal A: i.e., $\{\} \in \xi(A)$.
2. the goal A depends on itself, without a base case.
We distinguish between these two scenarios by marking the first node as '`fail`' and the second as '`ancestor`'.

In summary, we do not use cyclic explanations to justify a true literal. In contrast, cyclic explanations describe infinite proof paths and can be used to justify a false literal. Instead of explicitly representing these cycles, however, we choose to keep the justification as an acyclic graph. Formally:

Definition 4 (Justification). *A justification for an atom A with respect to program P and table T, denoted by $\mathcal{J}(A, P, T)$ is a directed acyclic graph $G = (V, E)$ with vertex labels chosen from $T \cup \{\mathtt{fact}, \mathtt{fail}, \mathtt{ancestor}\}$ such that:*

1. *G is rooted at A, and is connected*
2. $(B_1, \mathtt{fact}) \in E \iff \{\} \in \xi(B_1) \wedge \tau(B_1) = \mathit{true}$
3. $(B_1, \mathtt{fail}) \in E \iff \{\} \in \xi(B_1) \wedge \tau(B_1) = \mathit{false}$
4. $(B_1, \mathtt{ancestor}) \in E \iff \tau(B_1) = \mathit{false} \wedge \xi(B_1) = \{L\}$ $\wedge\ \exists\ B_2 \in L$ *s.t.* $(B_2, B_1) \in E^* \ \vee\ B_2 = B_1$
5. $(B_1, B_2) \in E \ \wedge\ B_2 \in T \ \wedge\ \tau(B_1) = \mathit{false} \iff$ $\xi(B_1) = \{L\} \ \wedge\ B_2 \in L \ \wedge\ (B_2, B_1) \notin E^*$
6. $(B_1, B_2) \in E \ \wedge\ B_2 \in T \ \wedge\ \tau(B_1) = \mathit{true} \implies$ $\exists L \in \xi(B_1)$ *s.t.* $B_2 \in L \ \wedge\ \forall B' \in L\ (B', B_1) \notin E^*$
7. $B_1 \in V \wedge \tau(B_1) = \mathit{true} \implies \exists$ *unique* $L \in \xi(B_1)$ *s.t.* $\forall B_2 \in L\ (B_1, B_2) \in E \wedge B_2 \in T \wedge (B_2, B_1) \notin E^*$

The above definition uses two sets of conditions for adding edges from a vertex in the justification graph. The first set is based on ξ's, while the second set specifies the global constraint that an edge can be added only when no cycles are created. In the above, rule 1 ensures that only information relevant to A, the answer being justified, is present in the graph. Rules 2 & 3 mark the end (leaf) states of derivations. Rules 4 & 5 ensures that the graph stays acyclic, while still containing information about cyclic dependencies between failed answers. Rules 6 & 7 select, among the different sets in $\xi(B_1)$, one that does not contain an ancestor to B_1.

The justification of the truth values of p w.r.t. programs in Figures 3(a) and (b) are given in Figures 5(a) and (b). Note that the selection of a single set out of $\xi(B_1)$ for construction means that the justification is an and-graph. Hence a justification provides one evidence for the truth or falsehood of a literal, even though the tabled evaluation may have explored/provided many more evidences. In the following, we show the "sufficiency" of a justification: that it contains enough information to reconstruct a SLD derivation.

2.2 Justification and SLD resolution

We now investigate the relationship between justification of an atom A in program P and the SLD tree(s) of A in P. For simplicity of exposition, we consider only propositional programs. Extension of our results to non-propositional programs is straightforward. Let P be a program, $A \in HB(P)$ and T the table created by evaluating A in P. Our aim is to show that the justification $\mathcal{J}(A, P, T)$ contains sufficient evidence for showing truth/falsehood of A in P.

Suppose $A \in M(P)$. Recall that $A \in M(P)$ iff there exists a successful SLD derivation of A in program P. Then the justification $\mathcal{J}(A, P, T)$ is a directed acyclic graph whose: *(i)* nodes are labeled with ground atoms, *(ii)* root is labeled with A, *(iii)* leaves are labeled with `fact` and *(iv)* the outgoing arcs of a node denote the application of a clause in P. Thus, a SLD derivation of A in P can be obtained by *linearizing* the justification graph into a sequence of ground goals (a goal is a conjunction of atoms). Formally:

Lemma 5. *Let P be a program, $A \in M(P)$ and J_A a justification of A in P. Then there exists a successful SLD derivation of A in P which can be constructed from J_A.*

Proof Sketch: We construct the SLD derivation $l(J_A)$ where l, the linearization operator is defined as follows. Let G be a directed acyclic graph and let the root of G be A. Let the children of A in G be $B_1, \ldots, B_n$ and the graphs rooted at $B_1, \ldots, B_n$ be $G_1, \ldots, G_n$. Then

$$l(G) = A \to (l(G_1) \wedge (B_2 \wedge \ldots \wedge B_n)) \to \ldots \to l(G_n)$$

where for any sequence of goals $\beta_1, \ldots, \beta_k$ and goal β we have $(\beta_1, \ldots, \beta_k) \wedge \beta = \beta_1 \wedge \beta, \ldots, \beta_k \wedge \beta$ □

Now suppose $A \notin M(P)$. Recall that $A \notin M(P)$ iff there exists a failed SLD tree of A in P, *i.e.*, a SLD tree with only finitely failed or infinite branches. Recall that in a justification, a false atom is explained by one false body atom in each of its clause instances. On the other hand, in a SLD tree a false atom is explained with the clause bodies of the applicable clauses. Given a justification J_A of atom A in program P, we show the existence a failed SLD tree T_A of A which uses the same evidence as J_A. Let the children of A in justification J_A be $B_1, \ldots, B_n$. Then for all $1 \leq i \leq n$ atom B_i appears in goal β_i where β_i is the body of one of the clauses of A. In the SLD tree T_A the children of A are $\beta_1, \ldots, \beta_n$ and we select the atom B_i in goal β_i for resolution ($1 \leq i \leq n$). Continuing in this way, we can construct a SLD tree T_A such that for every finitely failed branch of T_A the sequence of selected atoms is a root-to-leaf path in J_A and for every infinite path of T_A, the longest non-repeating prefix of the sequence of selected atoms is a root-to-leaf path in J_A. Formally:

LEMMA 6. *Let P be a program, $A \notin M(P)$ and J_A a justification of A in P. Then there exists a failed SLD tree T_A of A in P s.t.*
(i) *for every finitely failed branch in T_A: $(A, \beta_1 \wedge A_1 \wedge \beta_1', \ldots, \beta_n \wedge A_n \wedge \beta_n', \texttt{fail})$ s.t. the sequence of selected atoms is $A, A_1, \ldots, A_n$, there exists a root-to-leaf path in J_A: $(A, A_1, \ldots, A_n, \texttt{fail})$.*
(ii) *for every infinite branch in T_A: $(A, \beta_1 \wedge A_1 \wedge \beta_1', \ldots, \beta_n \wedge A_n \wedge \beta_n', \beta_{n+1} \wedge A_i \wedge \beta_{n+1}', \ldots)$ s.t. $1 \leq i \leq n$, the sequence of selected atoms is $A, A_1, \ldots, A_n, A_i, \ldots$ and the atoms $A, A_1, \ldots, A_n$ are distinct, there exists a root-to-leaf path $(A, A_1, \ldots, A_n, \texttt{ancestor})$ in J_A.*

The connection between justification and SLD resolution is formally summarized in the following theorem. This theorem establishes that justification contains sufficient evidence to explain the truth/falsehood of an atom.

THEOREM 7 (SUFFICIENCY OF JUSTIFICATION). *Let P be a program, $A \in HB(P)$ and J_A a justification of A. If $A \in M(P)$ then a successful SLD derivation of A in P can be constructed from J_A. If $A \notin M(P)$ then the selected atom sequence of every path of a failed SLD tree can be constructed from J_A.*

2.3 An Algorithm for Justification

Note that edges to `ancestor` are used to mark cyclic dependencies between failed answers. There is usually a choice of which dependencies to leave as edges in a justification and which to mark as `ancestor`. The optimal solution to this problem is NP-complete, by reduction from Feedback Vertex Set [15]. However, optimality itself is relatively unimportant in this context: the proof search done by the underlying evaluation engine has a far greater impact on the size of the justification graph. So we present a linear-time algorithm to construct a justification, based on depth-first-search that heuristically eliminates "back edges" which contribute to cycles. We describe the algorithm below.

The justification algorithm is given in Figure 6. The algorithm treats the given program P and tables T as global read-only data structures. It monotonically adds entries to two other global structures: the justification graph itself (V: set of vertices, E: set of edges), and a marking (*Done*) on the set of vertices in the graph. The initial call to *Justify* is made with an empty justification graph and *Done* initialized to the empty set.

The recursive algorithm builds the justification graph by traversing it depth-first even as it is constructed. At any point, note that V is the set of "visited" vertices, and *Done* is the set of vertices whose descendents have been completely explored. The algorithm maintains an important invariant that for any call, the parameter A is either unvisited, or is already in *Done*. This invariant implies that (i) the algorithm terminates, and (ii) no cycles are created in the justification graph. This invariant can be easily established by noting that the set $V - Done$ contains exactly those vertices B that are ancestors to the current vertex A. Moreover, if every set in $\xi(A)$ contains a vertex that is an ancestor of A, then A depends recursively on itself without a base case. Hence, $\tau(A) = \mathit{false}$. This property is used by the algorithm for placing edges to `ancestor`.

We define $\|\xi(A)\| = \sum_{L \in \xi(A)} |L|$. Since $\|\xi(A)\|$ is bounded by the size of tables for any A, and there may be at most $\|\xi(A)\|$ outgoing edges from vertex A, the worst-case complexity of the algorithm is $O(|T|^2)$. In fact, any justification algorithm must be $O(|T|^2)$ in the worst case as demonstrated by the example program:

$$P_{worst} = \{\mathrm{p}_i :- \mathrm{p}_j \mid 1 \leq i, j \leq n\}$$

In this program, the justification of any p_i is quadratic in the size of the tables generated by evaluating p_i. It should, however, be noted that $\Sigma_{A \in T} \|\xi(A)\|$, which is a bound on the number of edges in a justification graph, is in turn bounded by the number of resolution steps needed. Hence, justification time is always bounded by time taken for resolution.

2.3.0.3 Example:.

For the justification graph shown in Figure 2(c), the root node is `nbisim(p,q)`. $\xi(\texttt{nbisim(p,q)}) = \{\{\texttt{nbisim(q,p)}\}\}$ Since there is no ancestor `nbisim(q,p)` we pick this explanation and now recursively invoke *Justify*(`nbisim(q,p)`). By using the first clause of `nbisim` we have $\xi(\texttt{nbisim(q,p)}) = \{\{\texttt{trans(q,a,q1)}, \texttt{forall(Y,trans(p,a,Y),nbisim(q1,Y))}\}\}$. We recursively invoke *Justify* for both of these two true atoms. These atoms are justified by using predicate `trans`, the transition relation for the automata of Figure 2(a).

2.4 Justification for Normal Logic Programs

The notion of justification, as well as the algorithm we have presented can be easily extended to normal logic programs evaluated under well-founded semantics [16].

First of all, even for stratified programs, we need to consider negative as well as positive literals when defining ξ's and justification. We denote negative literals by `not(`A`)` where A is the corresponding positive literal. Truth values for negative literals is defined as $\tau(\texttt{not}(A)) = \neg\tau(A)$. We define ξ's for positive literals as given in Definition 2; for negative literals, we define $\xi(\texttt{not}(A))$ to be $\{\{A\}\}$: i.e., the truth/falsehood of `not(`A`)` will be explained in terms of the falsehood/truth of A. The current statement of Theorem 3 will remain valid for positive literals provided we consider the perfect model $\hat{M}(P)$ instead of the least Herbrand model $M(P)$; the theorem can be readily extended to accommodate negative literals. The definition of justification can be extended by considering `not(`A`)` as potential label for vertices if $A \in T$.

Recall that well-founded semantics (WFS) is defined over a three-valued model, where each literal is assigned a truth value of *true*, *false* or *undefined*. SLG resolution [6] and its implementations represent literals with *undefined* values as conditional answers and *true* values as unconditional answers. Hence, we can split the answer table T_A into a set of *true* (T_{A_t}) and *undefined* (T_{A_u}) answers. We can now extend Definition 1 by setting $\tau(A) = \mathit{true}$ if $A \in T_{A_t}$ and $\tau(A) = \mathit{undefined}$ if $A \in T_{A_u}$. The definitions of $\tau(A) \in \{\mathit{false}, \mathit{uncomputed}\}$ remain unchanged.

The most substantial extension to accommodate normal programs will be the addition of locally consistent explanations for *undefined* literals. To explain why a literal, say A, is *undefined* under WFS, one needs to show that the for each clause C in the program whose head unifies with A, either (i) the body of C contains a *false* literal, or (ii) all literals in

```
algorithm Justify(A : atom)
        (* Global: P : program, (V,E): Justification, Done ⊆ V *)
    if (A ∉ V) then (* A has not yet been justified *)
        set V := V ∪ {A}
        let α_A ∈ ξ(A) such that (α_A ∩ V) ⊆ Done
        if (α_A exists) then
            if (α_A = {}) then
                if (τ(A) = true) then
                    set E := E ∪ (A, fact)
                else
                    set E := E ∪ (A, fail)
            else (* α_A ≠ {} *)
                for each B ∈ α_A do
                    set E := E ∪ (A, Justify(B))
        else (* No such α_A ⟹ every set in ξ(A) contains an ancestor of A *)
            set E := E ∪ (A, ancestor)
            let {α'_A} = ξ(A) (* note that τ(A) will be false *)
            for each B ∈ (L'_A − (V − Done)) do
                set E := E ∪ (A, Justify(B))
        set Done := Done ∪ {A}
        return A
    else (* A has been justified *)
        return A
```

Figure 6: Justification algorithm

the body of C are *true* or *undefined*. Moreover, there must be at least one clause of type (ii). Hence the evidence for A's undefinedness will be $\xi(A) = \{L\}$ provided $L = \alpha_f \cup \alpha_{tu}$, where α_f is the set of all false literals from clauses of type (i), and α_{tu} is the set of all literals from clauses of type (ii).

Observe that the ξ's of false and undefined literals are singleton sets: there is only one way to justify them. This observation immediately yields the following simple modification to the definition of justification to accommodate normal programs: in rules 4 and 5 of Definition 4 (page) change the test $\tau(B_1) = \mathit{false}$ to $\tau(B_1) \in \{\mathit{false}, \mathit{undefined}\}$. In effect, by separating local concerns of explanation to global concerns of justification, we have been able to accommodate normal programs by suitably extending the (local) notion of ξ alone. Moreover, note that Algorithm *Justify* (Figure 6) does not explicitly test for the truth assignment of a literal for adding edges specified by rules 4–7 of Definition 4 and hence needs no modification for handling normal logic programs.

2.5 Justifying in the Presence of Builtins

We have thus far assumed that all predicate symbols in a program are tabled. Many application programs contain a mixture of tabled and non-tabled predicates. Justification of non-tabled predicates presents two immediate problems. First, justification introduces unacceptable overheads since the only way to determine the truth value of a goal (e.g., to compute ξ) is to reevaluate the goal. Second, non-tabled predicates often involve non-logical constructs for which justification may be difficult or even impossible. We use the following strategy for justifying programs that contain non-tabled predicates. We first ensure that non-tabled predicates are invoked from existing tabled predicates via new wrapper predicates, which are also tabled. We then provide justification rules to reason about the truth value of these wrapper predicates. In the simplest case, the non-tabled predicates do not in turn invoke tabled predicates, and the justification rules will treat the corresponding wrapper predicates as a database of facts. In more complex cases, the justification rules will *hide* the procedural components of the non-tabled predicates and give a logical view of its functionality.

For example, consider the `forall` predicate in the non-bisimulation checking program in Figure 1. This is a user-introduced wrapper predicate which is tabled. Logically `forall(X,G,H)` represents the first order formula $\forall$`X G` $\Rightarrow$ `H`. Note that `forall(X,G,H)` is equivalent to `not(g(X), not(h(X)))`, where `g(X)` and `h(X)` are defined in terms of `G` and `H` respectively. Since the equivalent form can introduce loops through negation, the underlying implementation of `forall` will typically use non-logical Prolog builtins such as `findall`. The use of non-tabled predicates is typically motivated by such pragmatic considerations. However, note that since justification is done after query evaluation, we can consider the logical meaning of these builtins instead of their implementation, at least from the point of view of justification. For instance, we can choose to justify `forall(X,G,H)` in terms of `not(g(X), not(h(X)))`. The justification rules— the mapping of "nonlogical" predicates to their justifiable counterparts— are themselves encoded as a set of Horn clauses.

The wrapper-based scheme for justifying non-tabled predicates works well as long as these predicates do not require any debugging. This assumption clearly holds for builtins. For programs containing user defined tabled and non-tabled predicates, we need to combine justification of tabled predicates with trace-based debugging of non-tabled predicates. Such an integration is a topic of future research.

3. EXTRACTING HIGH-LEVEL PROOFS

So far, we have shown how we can encode a proof system (Figure 1(a)) as a tabled logic program (Figure 1(b)), dispense proof obligations in the proof system by query evaluation of the logic program (Figure 2(b)) and construct the justification graph (Figure 2(c)) from the tables constructed during query evaluation. The extraction of tableau proofs (Figure 2(d)) from justification graph is now shown.

For our formalization we define a *proof system* to be a set

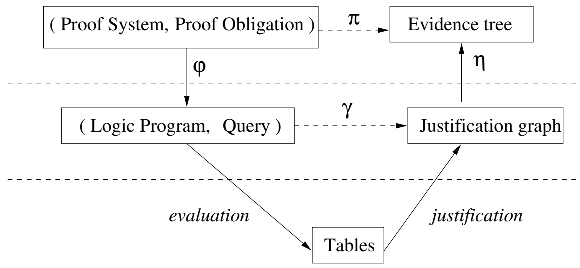

Figure 7: An architecture for justification

of *proof rules*; each consists of a set of premises, a side condition and a conclusion. We denote a rule by $\psi_1, \ldots, \psi_k \xrightarrow{S} \psi$ where $\psi_1, \ldots, \psi_k$ denote the premises, S the side condition and ψ the conclusion of the rule. Axioms in our proof system are denoted as $fact \longrightarrow \psi$. We assume that the side condition of a proof rule is locally testable. We now define the notion of an *evidence tree* for a proof obligation in a proof system as follows:

Definition 8 (Evidence Tree). *Evidence tree for an obligation O in a proof system R, denoted $\pi(O, R)$, is a finite tree T s.t. root of T is O, and for any node ψ in T:*

- *$R \vdash \psi$: Let $\psi_1, \ldots, \psi_k$ be the children of ψ. Then $\forall i,\ 1 \leq i \leq k\ R \vdash \psi_i$, and $\psi_1, \ldots, \psi_k \xrightarrow{S} \psi$ is an instance of a rule in R whose side condition S is true.*
- *$R \not\vdash \psi$: Let $\psi_1, \ldots, \psi_k$ be the children of ψ. Then $\forall i,\ 1 \leq i \leq k\ R \not\vdash \psi_i$, and every instance of a rule in R whose side condition is true and conclusion is ψ will have a premise ψ' s.t. either $\psi' \in \{\psi_1, \ldots, \psi_k\}$ or ψ' appears as an ancestor of ψ in T. If no such $\{\psi_1, \ldots, \psi_k\}$ exists then node ψ has a child "fail".*

The conditions in the above definition of an evidence tree are analogous to those imposed in the definition of locally consistent explanation (Definition 2). Furthermore, as in the case of justification (Definition 4), our notion of evidence also imposes a finiteness restriction. However, note that we do not retain any explicit indication of cycles in the evidence tree whereas in the justification graph cycles are kept track of by maintaining leaves labeled `ancestor`.

An architecture for justification shown in Figure 7, is an overview of the interaction between a proof system and its logic program encoding. We encode the proof rules R and a proof obligation O as a logic program P and a query Q respectively using an encoding function φ. We point out that logic programs encoding tableau systems (such as the ones used in model checking [4, 33]) are stratified in general. Hence, we assume that the encoding function φ maps a proof system to a (dynamically) stratified logic program [26]. The semantics of a stratified logic program P is given by its *perfect model* $\hat{M}(P)$ which is a natural extension of the least Herbrand model semantics for definite programs. Moreover, the perfect model of any stratified program coincides with its unique stable model [17] as well as its 2-valued well-founded model [16].

Definition 9 (Encoding Function). *An encoding function φ is a mapping from (Proof Obligation $\times$ Proof System) to (Query $\times$ Stratified Logic Program) such that for any proof obligation O and proof system R*

$$\varphi(O, R) = (Q, P) \Rightarrow (\ R \vdash O \Leftrightarrow Q \in \hat{M}(P)\ )$$

In our non-bisimilarity example, Figure 1 shows the mapping φ of the proof system to its logic program encoding. The non-bisimilarity relation $\not\approx$ is encoded as the predicate `nbisim`. We use other predicates such as `trans` and `forall` to encode the side conditions of the proof rules.

Once the proof obligation and proof system are encoded as a query and a logic program, we construct the justification graph of the query. This is shown as function γ in Figure 7 and is computed by tabled evaluation followed by justification. Finally, we need to map the justification graph to an evidence tree. This is done via the extraction function η (see Figure 7). Our definition of η is dependent on the encoding function φ since η in some sense undoes the effect of φ. Specifically, the effect of φ is to map one application of a proof rule in an evidence tree to several logic program derivation steps. On the other hand, η maps several steps in the justification graph to a single node in the evidence tree.

We assume $\varphi(O, R) = (\varphi_l(O), \varphi_r(R)) = (Q, P)$. Thus, φ_l is a mapping from proof obligations to ground atoms and φ_r is a mapping from a proof system to a logic program. We further assume that φ_r maps each proof rule to a unique set of program clauses. Then there exists a partial function φ_r^{-1} from program clauses to proof rules s.t. if $\varphi_r(\rho) = \{C_1, \ldots, C_k\}$ for some rule ρ and clauses $\{C_1, \ldots, C_k\}$ then $\varphi_r^{-1}(C_i) = \rho$ for all $1 \leq i \leq k$. Given a justification graph J, in order to construct an evidence tree we need to locate the use of those program clauses for which φ_r^{-1} is defined. We then replace the use of such a program clause C with the use of the corresponding proof rule $\varphi_r^{-1}(C)$. By repeatedly applying φ_r^{-1} to justification graph J starting from the root of J, we obtain $\eta(J)$. Formally, we define η as follows:

Definition 10 (Extraction Function). *Let R be a given proof system and $\varphi_r(R) = P$. For any proof obligation O, let J_O denote the justification of query $\varphi_l(O)$ in program P. Then $\eta(J_O)$ is a tree constructed as follows:*

1. *the root of $\eta(J_O)$ is O.*
2. *(a) If $\varphi_l(O) \in \hat{M}(P)$, let the children of $\varphi_l(O)$ in J_O be obtained by applying program clause $C \in P$. Let $O_1, \ldots, O_k$ be the premises in the corresponding proof rule $\varphi_r^{-1}(C)$. Then $\eta(J_{O_1}), \ldots, \eta(J_{O_k})$ are the subtrees of O in J_O.*
   (b)if $\varphi_l(O) \notin \hat{M}(P)$, let the children of $\varphi_l(O)$ in J_O be body literals from clause instances $\{C_1\theta_1, \ldots, C_k\theta_k\}$. For all $1 \leq i \leq k$, let O_i be a premise of $\varphi_r^{-1}(C_i)\theta_i$ s.t. $\varphi_l(O_i) \notin \hat{M}(P)$. Then the subtrees of O in J_O are $\eta(J_{O_1}), \ldots, \eta(J_{O_k})$.

As shown in Figure 7, we construct the evidence of a proof obligation in a proof system by applying the following steps in sequence: (a) apply the encoding function φ to map the proof obligation and the proof system to a query and a logic program (b) apply function γ to compute the justification of the query in the logic program (c) apply the extraction function η to the justification to obtain the evidence.

Theorem 11. *For any proof system R and proof obligation O, $\eta(\gamma(\varphi(O, R)))$ is an evidence tree.*

The proof is by induction on size of the computed evidence. (Details are skipped.)

Let us revisit the non-bisimilarity example. φ_r^{-1} maps the first clause of `nbisim` to proof rule (1) and the second clause of `nbisim` to proof rule (2) of Figure 1(a). Now, let us consider the justification constructed for the query `nbisim(p, q)` given in Figure 2(c). The first step in the justification of `nbisim(p,q)` corresponds to an application of the second clause for `nbisim`. Using φ_r^{-1}, we map it to an application of proof rule (2). Applying proof rule (2) to the proof obligation $\mathtt{p} \not\approx \mathtt{q}$ we obtain the proof obligation $\mathtt{q} \not\approx \mathtt{p}$. Now, the justification of `nbisim(q,p)` is done by applying the first clause of `nbisim`. Again using φ_r^{-1} we map it to an application of proof rule (1). Applying proof rule (1) on $\mathtt{q} \not\approx \mathtt{p}$ we obtain no new obligations therefore our evidence tree construction is completed. The constructed evidence tree is shown in Figure 2(d).

4. APPLICATION TO MODEL CHECKING

Model checking [8] is an automatic technique for verifying if a finite-state concurrent system specification satisfies a property expressed as a temporal logic formula. In [27] we had demonstrated the feasibility of using logic programming for building flexible and efficient model checkers. In particular, using the XSB tabled logic programming system we developed XMC, a local model checker for a CCS-like value-passing system specification language and the modal mu-calculus temporal logic. XMC is written under 200 lines of tabled Prolog code that directly encodes the semantics of CCS and modal mu-calculus specified as tableau rules. Despite the high-level nature of XMC's implementation, its performance is comparable to that of highly optimized model checkers such as Spin [19] and Murφ [11] on examples selected from the benchmark suite in the standard Spin distribution.

Model checking in XMC corresponds to evaluating a top-level query that denotes the temporal property of interest. The query succeeds whenever the system satisfies the property. To explain the success or failure of the query we use the XMC justifier, developed based on the techniques described in this paper.[1] In this section we sketch the application of justification to logic programming based model checking. We also provide experimental evidence of the effectiveness of justification in the setting of model checking.

For simplicity of exposition we will use CTL, a branching time temporal logic [13] to illustrate the application of justification to model checking. As is usual in CTL model checking we will assume that the system is specified as a Kripke structure [22], which is a 3-tuple (S, R, L) with S denoting the set of states, R the transition relation and L the valuation function that assigns $true/false$ values to the atomic propositions associated with the states. For ease of understanding the material in this section we briefly review the essential aspects of CTL model checking. (See [7] for an excellent introduction to this topic.)

Conceptually CTL formulas can be viewed as describing properties over computation trees that are formed by designating a state in the Kripke structure as the initial state and then unfolding the structure into an infinite tree. CTL formulas are used to specify safety and liveness properties of concurrent systems. They are built from atomic propositions, $\wedge, \vee, \neg$ and the temporal connectives such as **EF**, **AG**, **AF**, etc. The *path quantifiers* **A** and **E** describe the branching structure in the computation tree. Specifically, **A** is used in a particular state to specify that all of the paths starting at that state have some property while **E** specifies that the property holds for some path. The *temporal operators* describe properties of a path through the tree. For example, **F** ("eventually in the future") operator asserts that a property will hold at some state on the path whereas the **G** ("always globally") operator specifies that a property holds at every state on the path.

```
:- table models/2.
models(S, and(F1,F2)) :-  models(S, F1),
                          models(S, F2).
models(S, ef(F)) :- models(S, F).
models(S, ef(F)) :- trans(S, T), models(T, ef(F)).
models(S, af(F)) :- models(S, F).
models(S, af(F)) :- forall(T, trans(S,T),
                             models(T, af(F))).
models(S, ag(F)) :-  negate(F, NF),
                     tnot(models(S, ef(NF))).
...
```

Figure 8: CTL model checker as a logic program

Figure 8 is a fragment of a CTL model checker encoded as a tabled logic program. It is a straightforward encoding of the semantics of CTL. Note that `models(S, F)` is true whenever $\mathtt{S} \models \mathtt{F}$, *i.e.* `S` satisfies the temporal property `F`. The `trans/2` predicate encodes the transition relation of the Kripke structure. The `tnot/1` is the negation operation for tabled predicates. The `forall/3` is a user-defined predicate for evaluating universally quantified first-order formulae. Thus `forall(T, trans(S, T), models(T, af(F)))` denotes the first-order formula $\forall$`T trans(S,T)` $\Rightarrow$ `models(T,af(F))`. To test if a state `s` satisfies a formula `f`, one simply asks the query `:- models(s,f)`. The answer tables created during query evaluation are post-processed by the justifier to explain the yes/no answer to the query.

Figure 9 illustrates the justification of the CTL formula **AG**p that asserts that in every future state along all paths p always holds. This formula is false for the Kripke structure in Figure 9(a) whereas it is true for the one in Figure 9(c). The corresponding justification for both these cases, derived using the techniques in Section 2, is shown in Figure 9(b) and Figure 9(d) respectively. Using the techniques in Section 3 one can transform the above justification done at the level of the logic program into one in terms of the tableau rules for the CTL model checker. Details are omitted.

4.1 Justification in Model Checking

The encoding of the CTL model checker in Figure 8 centers around two predicates: `trans`, encoding the transitional semantics of Kripke structures, and `models`, defining when a state in the Kripke structure models a given CTL formula. By appropriately redefining the `trans` and `models` predicates logic programming based model checkers for other system description languages and other temporal logics can be readily obtained. In fact the encoding of the CTL model checker was obtained in this fashion from our XMC model checker that is designed for value passing CCS and the modal mu-calculus temporal logic. More recently, we retargeted XMC to linear temporal logic (LTL) by redefining `models` in accordance with the proof system given in [3]. The larger

[1]The XMC system with the justifier is freely available from `http://www.cs.sunysb.edu/~lmc`.

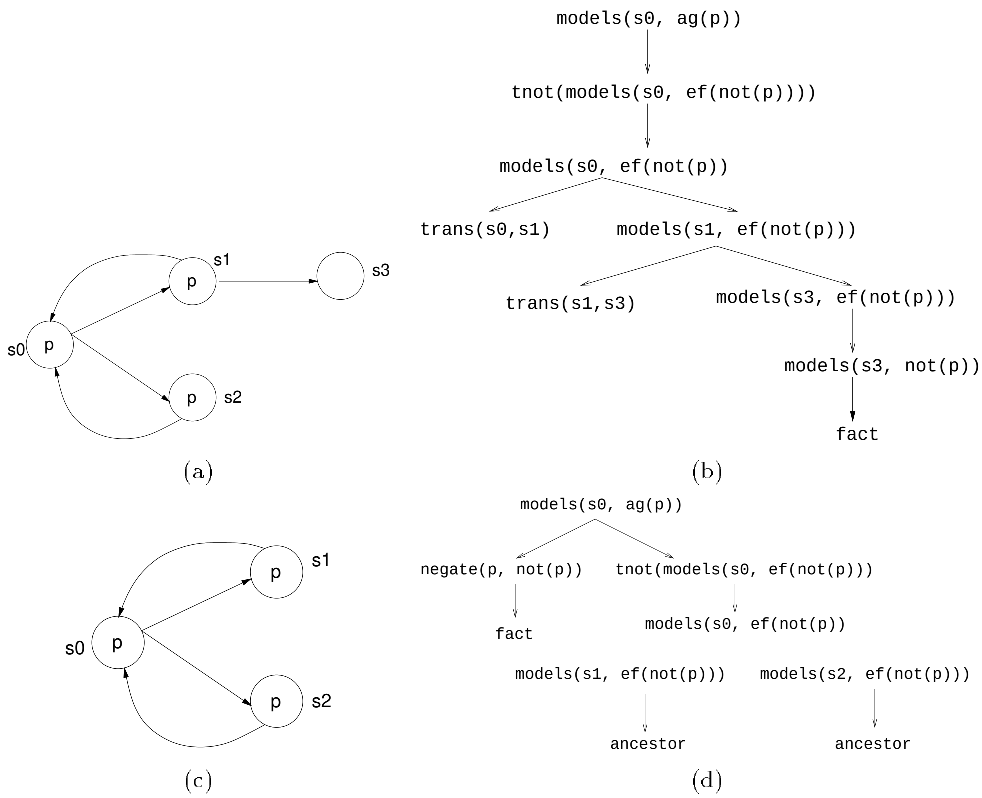

Figure 9: Example to illustrate justification of model checking queries

implication of the retargeting exercise is that since justification is a generic technique for explaining the results of query evaluation one can similarly generate justifiers for all of these retargeted model checkers.

Note that model checkers for both linear time temporal logic (e.g. SPIN [19]) as well as branching time (e.g. SMV [22]) present a counterexample to the user whenever $\mathtt{s} \not\models \mathtt{f}$. But no additional feedback is given to the user for the other case when $\mathtt{s} \models \mathtt{f}$. Justification differs from them in two respects. First, it is not restricted to any single temporal logic. It encompasses both linear and branching time logic. Secondly, it provides evidence not only when a formula is false but also when it is true. Justification goes beyond merely giving a yes/no answer to the model checking question. It generalizes the traditional notion of *counterexample generation* in model checking to *evidence generation*.

4.2 Experimental Results

We have built a justifier for our XMC model checker. It has been in operation for several months now. We have found it to be useful for quickly spotting bugs in specifications.

In Table 1 we present experimental results contrasting the sizes of the run-time trace produced by a trace-based debugger and the evidence produced by the justifier.[2] For the run-time trace we measure the size of the SLG forest created during query evaluation in the XSB tabled logic programming system [34]; for justification we measure the size of the justification DAG. For illustration we use two real-life protocols: the *GNU UUCP i-protocol* [12] and the *Java meta-locking protocol* [1]. The i-protocol is an optimized sliding window protocol for file transfers over serial lines and is part of the GNU UUCP protocol stack. The Java meta-locking protocol is an efficient protocol, developed by Sun Microsystems, for synchronizing access to objects by threads. The k in i-protocol(k) denotes window size k and the i, j pair in Metalock(i,j) denotes i threads and j objects. In i-protocol we show the results of finding a livelock which requires traversing only a fragment of the state space of the concurrent system. In meta-locking protocol, we show the proof size for the mutual exclusion property which requires traversing the entire state space.

Note that whenever the formula is true the justifier constructs the evidence based on just the path that succeeds whereas the SLG forest includes all the failing paths. Thus the difference in sizes between the SLG forest and justification is more dramatic in i-protocol (with livelock present)

System	Formula	SLG Forest Size	Justification Size
meta-lock(2,1)	mutex	6400	890
meta-lock(2,2)	mutex	103K	14K
i-protocol(1)	livelock	83K	2K
i-protocol(2)	livelock	602K	13K

Table 1: Experimental Data

[2] The sizes shown were collected by suppressing the explanation of certain predicates not relevant for explanation of the `models` query from both the justifier as well as the tracer.

where the formula is true than in Metalock where the formula is false. In cases where the formula is false the justifier does not use the true literals for evidence thereby contributing to the difference between the two sizes. It is noteworthy to observe that the difference between the two sizes becomes even more pronounced as the system size increases. This is because as the system size grows so does the number of both failing paths as well as literals that succeed.

Justifier provides succinct evidence that makes it relatively easy to comprehend the results of the model checker. Since it is constructed as a post-processing step the justification DAG is built in its entirety. Hence, by providing mechanisms to navigate the DAG *on-demand* the justifier provides additional opportunities for the user to inspect only interesting parts of the justification DAG. Such mechanisms can include existing techniques from traditional debugging such as setting break points, leap, etc. The utility of these techniques has been amply borne out with our own experience in using the XMC justifier whose implementation has incorporated some of them.

5. DISCUSSIONS

We proposed the concept of justification to explain the success/failure of a query to a logic program. The justifier constructs succinct evidence by post-processing the tables created during query evaluation. We showed how to elevate justification of logic programs to tableau systems and provided evidence of its utility in model checking.

The concept of justification is also useful in other applications. Below we discuss its role in programming. In the evaluation and justification of tableau systems we can clearly discern two distinct albeit separate phases – a query evaluation phase where we *search for the existence* of a successful tableau and a justification phase where we *construct* evidence based on the outcome of the search. Doing an *existence search* followed by a *construction process* offers a simple yet powerful computing paradigm.

As an example, consider the problem of constructing a parse tree from context free grammars (CFG). It well known how to build Definite Clause Grammar (DCG) parsers for recognizing CFGs in cubic time (assuming the grammar is in Chomsky Normal Form). But it is rather difficult to construct a parse tree efficiently without employing complex encoding tricks [32]. We can readily cast the parse tree construction of a given string as one of searching for the existence of a parse tree followed by constructing one if it exists. In the first step, the CFG is encoded as a DCG in a simple and straightforward manner. Testing if a string is parseable then corresponds to query evaluation of this DCG encoding in a tabled logic programming system. If the string is parseable, then in the second step we construct the evidence of the parse by invoking the justifier. This results in the parse tree, which is constructed in linear time. Indeed we have used our justifier tool for efficient construction of parse trees for strings in context free grammars encoded as DCG in the XSB tabled logic programming system [34].

Thus the search and construct paradigm provides an elegant programming abstraction that can bring conceptual simplicity to the formulation and evaluation of certain programming tasks. The justifier makes such a programming abstraction feasible, by providing an efficient implementation scheme for the abstraction.

6. ACKNOWLEDGMENTS:

This work was partially supported by NSF grants CCR-9711386, CCR-9876242 and EIA-9705998. We thank David S. Warren for a preliminary implementation of a justifier for propositional programs, and for valuable discussions during the design of the XMC justifier.

7. REFERENCES

[1] O. Agesen, D. Detlefs, A. Garthwaite, R. Knippel, Y. S. Ramakrishna, and D. White. An efficient meta-lock for implementing ubiquitous synchronization. In *Proceedings of OOPSLA*, 1999.

[2] T. Arora, R. Ramakrishnan, W.G. Roth, P. Seshadri, and D. Srivastava. Explaining program executions in deductive systems. In *Proceedings of DOOD, LNCS 760*, 1993.

[3] G. S. Bhat, R. Cleaveland, and O. Grumberg. Efficient on-the-fly model checking for CTL*. In *LICS'95*, pages 388–397, San Diego, July 1995. IEEE Computer Society Press.

[4] J. C. Bradfield. *Verifying Temporal Properties of Systems.* Birkhauser, 1992.

[5] M. Carro, L. Gomez, and M. Hermenegildo. Some paradigms for visualizing parallel execution of logic programs. In *Intl. Conf. on Logic Programming*, 1993.

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

[7] E. M. Clarke, O. Grumberg, and D. Peled. *Model Checking.* MIT Press, 1999.

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

[9] Subrata K. Das. *Deductive Databases and Logic Programming.* Addison-Wesley, 1992.

[10] T. Diaz and E. Lusk. A graphical tool for observing the behavior of parallel logic programs. In *Symposium on Logic Programming*, 1987.

[11] D. L. Dill. The Murφ verification system. In R. Alur and T. A. Henzinger, editors, *Computer Aided Verification (CAV '96)*, volume 1102 of *Lecture Notes in Computer Science*, pages 390–393, New Brunswick, New Jersey, July 1996. Springer-Verlag.

[12] Y. Dong, X. Du, Y. S. Ramakrishna, C. R. Ramakrishnan, I.V. Ramakrishnan, S. A. Smolka, O. Sokolsky, E. W. Stark, and D. S. Warren. Fighting livelock in the i-Protocol: A comparative study of verification tools. In *TACAS*, volume LNCS 1579, 1999.

[13] E. A. Emerson. *Temporal and Modal Logic - in Handbook of Theoretical Computer Science: Volume B, Formal Models and Semantics.* North-Holland Pub. Co./MIT Press, 1990.

[14] M. Fitting. *Proof methods for modal and intuitionistic logics.* Reidel, 1983.

[15] M. Garey and D.S. Johnson. *Computers and Intractability.* Freeman, 1979.

[16] A. Van Gelder, K.A. Ross, and J.S. Schlipf. Unfounded sets and well-founded semantics for

general logic programs. *Journal of the ACM*, 38(3):620–650, 1991.

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

[18] M. Hermenegildo, G. Puebla, and F. Bueno. *Using Global Analysis, Partial Specifications, and an Extensible Assertion Language for Program Validation and Debugging*, pages 161–192. 1999.

[19] G. J. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[20] J.W. Lloyd. Declarative error diagnosis. *New Generation Computing*, 5(2):133–154, 1987.

[21] S. Mallet and M. Ducasse. Generating deductive database explanations. In *Proceedings of ICLP*, pages 154–168, 1999.

[22] K. L. McMillan. *Symbolic Model Checking*. Kluwer Academic, 1993.

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

[24] L. Naish, P.W. Dart, and J. Zobel. The NU-prolog debugging environment. In *ICLP*, pages 521–536, 1989.

[25] Frank Pfenning. *Logic Programming in the LF logical framework*, pages 149–181. Cambridge University Press, 1991.

[26] T.C. Przymusinski. Every logic program has a natural stratification and an iterated least fixed point model. In *Principles of DataBase Systems*, pages 11–21, 1989.

[27] Y.S. Ramakrishna, C.R. Ramakrishnan, I.V. Ramakrishnan, S.A. Smolka, T. Swift, and D.S. Warren. Efficient model checking using tabled resolution. In *CAV, LNCS 1254*, 1997.

[28] Ehud Y. Shapiro. Algorithmic program diagnosis. In *POPL*, 1982.

[29] C.P. Stirling and D.J. Walker. Local model checking in the modal mu-calculus. In *Proceedings of TAPSOFT, LNCS 351*, pages 369–382, 1989.

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

[31] R. Vaupel, E. Pontelli, and G. Gupta. Visualization of and/or-parallel execution of logic programs. In *Intl. Conf. on Logic Programming*, 1997.

[32] D.S. Warren. *Programming in Tabled Prolog*. Draft Book : Available at `http://www.cs.sunysb.edu/~warren/xsbbook`, 1999.

[33] G. Winskel. Model checking the modal ν calculus. In *Proceedings of ICALP*, 1989.

[34] XSB. The XSB logic programming system v2.2, 2000. Available from `http://xsb.sourceforge.net/`.