Model Checking with Probabilistic Tabled Logic Programming

ANDREY GORLIN

C. R. RAMAKRISHNAN

SCOTT A. SMOLKA

Department of Computer Science
Stony Brook University, Stony Brook, NY 11794-4400, U.S.A.
(*e-mail:* {agorlin, cram, sas}@cs.stonybrook.edu)

Abstract

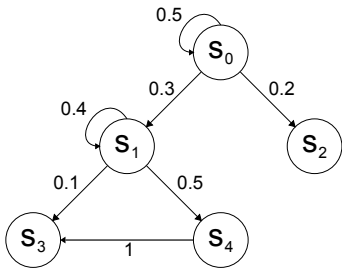
We present a formulation of the problem of probabilistic model checking as one of query evaluation over probabilistic logic programs. To the best of our knowledge, our formulation is the first of its kind, and it covers a rich class of probabilistic models and probabilistic temporal logics. The inference algorithms of existing probabilistic logic-programming systems are well defined only for queries with a finite number of explanations. This restriction prohibits the encoding of probabilistic model checkers, where explanations correspond to executions of the system being model checked. To overcome this restriction, we propose a more general inference algorithm that uses finite generative structures (similar to automata) to represent families of explanations. The inference algorithm computes the probability of a possibly infinite set of explanations directly from the finite generative structure. We have implemented our inference algorithm in XSB Prolog, and use this implementation to encode probabilistic model checkers for a variety of temporal logics, including PCTL and GPL (which subsumes PCTL*). Our experiment results show that, despite the highly declarative nature of their encodings, the model checkers constructed in this manner are competitive with their native implementations.

1 Introduction

Beginning in 1997, we formulated the problem of model checking as one of query evaluation over logic programs (Ramakrishna et al. 1997). The attractiveness of this approach is that the operational semantics of complex process languages (originally CCS (Milner 1989), followed by value-passing calculi (Ramakrishnan 2001), the pi-calculus (Milner et al. 1992), and mobile calculi with local broadcast (Singh et al. 2008)), as well as the semantics of complex temporal logics (e.g., the modal mu-calculus (Kozen 1983)), can be expressed naturally and at a high level as clauses in a logic program. Model checking over these languages and logics then becomes query evaluation over the logic programs that directly encode their semantics.

The past two decades have witnessed a number of important developments in Probabilistic Logic Programming (PLP), combining logical and statistical inference, and leading to a number of increasingly mature PLP implementations. A natural question is whether the advances in PLP enable the development of model checkers for *probabilistic systems*, the same way traditional LP methods such as tabled evaluation and constraint handling enabled us to formulate model checkers for a variety of non-probabilistic systems.

It turns out that existing PLP inference methods are not sufficiently powerful to be used as a basis for probabilistic model checking. One of the earliest PLP



(a)

```
% 3 "switches" (random processes) for transitions
% from states s0, s1 and s4, respectively.
values(t(s0), [s0, s1, s2]).
values(t(s1), [s1, s3, s4]).
values(t(s4), [s3]).

% Distribution parameters of the random variables.
set_sw(t(s0), [.5, .3, .2]).
set_sw(t(s1), [.4, .1, .5]).
set_sw(t(s4), [1]).

% Transition from S at instance I goes to T,
% as determined by the corresponding random process.
trans(S, I, T) :-  msw(t(S), I, T).

% Starting at state S at instance I, state T is reachable.
reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

(b)

Fig. 1. (a) Example Markov chain; (b) PRISM encoding of transitions in the chain.

inference procedures, used in PRISM (Sato and Kameya 1997), is formulated in terms of the set of *explanations* of answers. PRISM puts in place three restrictions to make its inference work: (a) *independence*: random variables used in any single explanation are all independent; (b) *mutual exclusion*: two distinct explanations of a single answer are mutually exclusive; and (c) *finiteness*: the number of possible explanations of an answer is finite. Subsequent systems, notably ProbLog (De Raedt et al. 2007) and PITA (Riguzzi and Swift 2010a) have eliminated the independence and mutual exclusion restrictions of PRISM. This, however, is still insufficient for model checking, as the following example shows.

Motivating Example: Figure 1 shows a Markov chain and its representation in PRISM. In any execution of the chain, a transition from state, say s, is independent of any previous transitions (including those from the same state). The definition of the `trans` predicate has an explicit instance parameter `I`, which is also used in `msw`. PRISM treats different instances of the same random variable as independent. Thus `trans` correctly encodes the semantics of the Markov chain.

We first consider simple reachability questions of the form: What is the likelihood that on an execution of the chain from a start state s, a final state t will be reached? The reachability question using the `reach` predicate is defined in Figure 1(b). Consider the likelihood of reaching state `s3` from `s0`. This query can be posed as the predicate `prob(reach(s0,0,s3),P)`, where `prob/2` finds the probability of answers (`P`) to a given query `reach(s0,0,s3)`.

The query `prob(reach(s0,0,s3),P)` cannot be evaluated in PRISM. We illustrate this by first describing PRISM's inference at a high level. In PRISM, inference of probabilities proceeds in the same way as logical inference, except when the selected literal is an `msw`. In this case, the inference procedure enumerates the values of the random variable and continues the inference for each value (by backtracking). The probability of a derivation is simply the product of the probabilities of the random variables (`msw` outcomes) used in that derivation (under the independence assumption). The probability of a query answer is the sum of probabilities of the set of all derivations for that answer (using the mutual-exclusion and finiteness as-

sumptions). Note that `reach(s0,0,s3)` has infinitely many derivations, and hence PRISM cannot infer its probability.

Markov chains can be encoded in ProbLog and LPAD (Vennekens et al. 2004) in a similar manner. The sequence of random-variable valuations used in the derivation of an answer is called an *explanation*. In contrast to PRISM, ProbLog (De Raedt et al. 2007) and PITA (Riguzzi and Swift 2010a), which is an implementation of LPAD, materialize the set of explanations of an answer in the form of a BDD. Probabilities are subsequently computed based on the BDD. This approach permits these systems to correctly infer probabilities even when the independence and mutual-exclusion assumptions are violated. However, the *set of explanations* of `reach(s0,0,s3)` is infinite.[1] Since BDDs can only represent finite sets, the probability of `reach(s0,0,s3)` cannot be computed in ProbLog or LPAD.

To correctly infer the probability of `reach(s0,0,s3)`, we need an algorithm that works even when the set of explanations is infinite. Moreover, it is easy to construct queries where the independence and mutual exclusion properties do not hold. For example, consider the problem of inferring the probability of reaching `s3` or `s4` (i.e., the query `reach(s0,0,s3); reach(s0,0,s4)`). Since some paths to `s3` pass through `s4`, explanations for `reach(s0,0,s3)` and `reach(s0,0,s4)` are not mutually exclusive. The example of Fig. 1 illustrates that to build model checkers based on PLP, we need an inference algorithm that works even when the finiteness, mutual-exclusion and independence assumptions are simultaneously violated.

Summary of Contributions: In this paper, we present PIP (for "Probabilistic Inference Plus"), a new algorithm for inferring probabilities of queries in a probabilistic logic program. PIP is applicable even when explanations are not necessarily mutually exclusive or independent and the number of explanations is infinite. We demonstrate the utility of this new inference algorithm by constructing model checkers for a rich class of probabilistic models and temporal logics (see Section 5). Our model checkers are based on high-level, logical encodings of the semantics of the process languages and temporal logics, thus retaining the highly declarative nature of our prior work on model checking non-probabilistic systems.

We have implemented our PIP inference algorithm in XSB Prolog (Swift et al. 2012). Our experimental results show that, despite the highly declarative nature of our encodings of the model checkers, their performance is competitive with their native implementations.

The rest of this paper develops along the following lines. Section 3 provides requisite background on probabilistic logic programming. Section 4 presents our PIP algorithm. Section 5 describes our PLP encodings of probabilistic model checkers, while Section 6 contains our experimental evaluation. Section 7 offers our concluding remarks and directions for future work.

2 Related Work

There is a substantial body of prior work on encoding complex model checkers as logic programs. These approaches range from using constraint handling to represent sets of states such as those that arise in timed systems (Gupta and Pontelli 1997;

[1] This is in contrast to the link-analysis examples used in ProbLog and PITA (Riguzzi and Swift 2010b), where, even though the number of derivations for an answer may be infinite, the number of explanations is finite.

Du et al. 2000; Mukhopadhyay and Podelski 2000; Pemmasani et al. 2002), data-independent systems (Sarna-Starosta and Ramakrishnan 2003) and other infinite-state systems (Delzanno and Podelski 1999; Mukhopadhyay and Podelski 1999); tabling to handle fixed point computation (Ramakrishnan et al 2000; Farwer and Leuschel 2004); procedural aspects of proof search to handle name handling (Yang et al. 2004) and greatest fixed points (Gupta et al. 2007). However, all these works deal only with non-probabilistic systems.

With regard to related work on probabilistic inference, Statistical Relational Learning (SRL) has emerged as a rich area of research into languages and techniques for supporting modeling, inference and learning using a combination of logical and statistical methods (Getoor and Taskar 2007). Some SRL techniques, including Bayesian Logic Programs (BLPs) (Kersting and Raedt 2000), Probabilistic Relational Models (PRMs) (Friedman et al. 1999) and Markov Logic Networks (MLNs) (Richardson and Domingos 2006), use logic to compactly represent statistical models. Others, such as PRISM (Sato and Kameya 1997), Stochastic Logic Programs (SLP) (Muggleton 1996), Independent Choice Logic (ICL) (Poole 2008), CLP(BN) (Santos Costa et al. 2003), ProbLog (De Raedt et al. 2007), LPAD (Vennekens et al. 2004) and CP-Logic (Vennekens et al. 2009), define inference primarily in logical terms, subsequently assigning statistical properties to the proofs. Motivated primarily by knowledge representation problems, these works have been naturally restricted to cases where the models and the inference proofs are finite. Recently, a number of techniques have generalized these frameworks to handle random variables that range over continuous domains (e.g. (Kersting and Raedt 2001; Narman et al. 2010; Wang and Domingos 2008; Gutmann et al. 2010; Gutmann et al. 2011; Islam et al. 2011)), but they still restrict proof structures to be finite.

Modeling and analysis of probabilistic systems, both discrete- and continuous-time, has been an actively researched area. Probabilistic Computation Tree Logic (PCTL) (Hansson and Jonsson 1994) is a widely used temporal logic for specifying properties of discrete-time probabilistic systems. PCTL* (Baier 1998) is a probabilistic extension of LTL and is more expressive than PCTL. Generalized Probabilistic Logic (GPL) (Cleaveland et al. 2005) is a probabilistic variant of the modal mu-calculus. The Prism model checker (Kwiatkowska et al. 2011) is a leading tool for modeling and verifying a wide variety of probabilistic systems: Discrete- and Continuous-Time Markov chains (DTMCs and CTMCs, respectively) and Markov Decision Processes. There is also prior work on techniques for verifying more expressive probabilistic systems, including Recursive Markov chains (RMCs) (Etessami and Yannakakis 2009) and Probabilistic Push-Down systems (Kucera et al. 2006), both of which exhibit context-free behavior. The probability of reachability properties in such systems is computed as the least solution to a corresponding set of monotone polynomial equations. PReMo (Wojtczak and Etessami 2007) is a model checker for RMCs. Reactive Probabilistic Labeled Transition Systems (RPLTS) (Cleaveland et al. 2005) generalize Markov chains by adding external choice (multiple labeled actions). GPL properties of such systems are also computed as the least (or greatest, based on the property) solution to a set of monotone polynomial equations. To the best of our knowledge, this paper presents the first implementation of a GPL model checker.

3 Preliminaries

Notations: The root symbol of a term t is denoted by $\pi(t)$ and its i-th subterm by $\arg_i(t)$. Following traditional LP notation, a term with a predicate symbol as root is called an *atom*. The set of variables in a term t is denoted by $vars(t)$. A term t is *ground* if $vars(t) = \emptyset$. We also use the standard notions from LP such as derivation and substitution (Nilsson and Maluszyński 2000). We denote the language of a grammar G by $\mathcal{L}_G$. For any string s in $\mathcal{L}_G$, the set of symbols in s is denoted by $sym(s)$.

Following PRISM, a *probabilistic logic program* (PLP) is of the form $P = P_F \cup P_R$, where P_R is a definite logic program and P_F is the set of all possible `msw/3` atoms. The set of possible `msw` atoms and the distribution of their subsets is given by `values` and `set_sw` directives, respectively. For example, clauses `trans` and `reach` in Fig. 1(b) are in P_R. The set P_F of that program contains `msw` atoms such as `msw(t(s0),0,s0)`, `msw(t(s0),next(0),s0)`, `msw(t(s0),0,s1)`, ..., `msw(t(s1),next(0),s1)`, ...

In an atom of the form $\texttt{msw}(t_1, t_2, t_3)$, t_1 is a term representing a random process (switch in PRISM terminology), t_2 is an instance and t_3 is the outcome of the process at that instance. According to PRISM semantics, two `msw` atoms with distinct processes or distinct instances are independent. Two `msw` atoms with the same process and instance but different outcomes are mutually exclusive.

4 The Inference Procedure PIP

A key idea behind the PIP inference algorithm is to represent the (possibly infinite) set of explanations in a symbolic form. Observe from the example in Fig. 1 that, even though the set of paths (each with its own distinct probability) from state `s0` to state `s3` is infinite, the regular expression $\texttt{s0}^+\texttt{s1}^+\texttt{s4}^?\texttt{s3}$ captures this set exactly. Following this analogy, we devise a grammar-based notation that can succinctly represent infinite sets of finite sequences.

Definition 1 (*Explanation*)
An *explanation* of an atom A with respect to a PLP $P = P_F \cup P_R$ is a set $\xi \subseteq P_F$ of `msw` atoms such that (i) $\xi, P_R \vdash A$ and (ii) ξ is consistent, i.e., it contains no pair of mutually exclusive `msw` atoms.

The set of all explanations of A w.r.t. P is denoted by $\mathcal{E}_P(A)$. □

Example 1 (*Set of explanations*)
Consider the PLP of Fig. 1(b). The set of explanations for `reach(s0,0,s3)` is:

$$\begin{array}{l}
\texttt{msw}(t(s0), 0, s1),\ \texttt{msw}(t(s1), next(0), s3). \\
\texttt{msw}(t(s0), 0, s0),\ \texttt{msw}(t(s0), next(0), s1),\ \texttt{msw}(t(s1), next(next(0)), s3). \\
\vdots \\
\texttt{msw}(t(s0), 0, s1),\ \texttt{msw}(t(s1), next(0), s1),\ \texttt{msw}(t(s1), next(next(0)), s3). \\
\vdots
\end{array}$$

4.1 Representing Explanations

As Example 1 illustrates, a representation in which instance identifiers are explicitly captured will not be nearly as compact as the corresponding regular expression

(shown earlier). On the other hand, a representation (like the regular expression) that completely ignores instance identifiers will not be able to identify identical instances of a random process nor properly distinguish distinct ones.

We solve this problem by observing that in PRISM's semantics, different instances of the same random process are *independent and identically distributed* (i.i.d.). Consequently, the probability of `reach(s0,0,s3)` (reaching `s3` from `s0` starting at instance `0`) is the same as that of `reach(s0,H,s3)` for any instance `H`. Hence, it is sufficient to infer probabilities for a single parameterized instance. Below, we formalize the set of PLP programs for which such an abstraction is possible.

Definition 2 (*Temporal PLP*)
A *temporal* probabilistic logic program is a probabilistic logic program P with declarations of the form `temporal(`$p/n-i$`)`, where p/n is an n-ary predicate, and i is an argument position (between 1 and n) called the *instance argument* of p/n . Predicates p/n in such declarations are called *temporal predicates*. □

The set of temporal predicates in a temporal PLP P is denoted by $\texttt{temporal}(P)$; the set of all predicates in P is denoted by $\texttt{preds}(P)$. By convention, every temporal PLP contains an implicit declaration `temporal(msw/3-2)`, indicating that `msw/3` is a temporal predicate, and its second argument is its instance argument. The instance argument of a predicate p/n is denoted by $\chi(p/n)$.

Let α be an atom in a temporal PLP such that its root symbol is a temporal predicate, i.e., $\pi(\alpha) \in \texttt{temporal}(P)$. Then the *instance of* α, denoted by $\chi(\alpha)$ by overloading the symbol χ, is $\arg_{\chi(\pi(\alpha))}(\alpha)$. We also denote, by $\overline{\chi}(\alpha)$, a term constructed by *omitting* the instance of α; i.e., if $\alpha = f(t_1, \ldots, t_{i-1}, t_i, t_{i+1}, \ldots t_n)$ and $\chi(\alpha) = t_i$, then $\overline{\chi}(\alpha) = f(t_1, \ldots, t_{i-1}, t_{i+1}, \ldots t_n)$.

Explanations of a temporal PLP can be represented by a notation similar to Definite Clause Grammars (DCGs).

Example 2 (*Set of explanations using DCG notation*)
Considering again the program of Fig. 1(b), the set of explanations for `reach(s0,H,s3)` can be succinctly represented by the following DCG:

$$\texttt{expl}(\texttt{reach}(s0, s3), H) \longrightarrow [\texttt{msw}(t(s0), H, s0)], \texttt{expl}(\texttt{reach}(s0, s3), next(H)).$$
$$\texttt{expl}(\texttt{reach}(s0, s3), H) \longrightarrow [\texttt{msw}(t(s0), H, s1)], \texttt{expl}(\texttt{reach}(s1, s3), next(H)).$$
$$\texttt{expl}(\texttt{reach}(s1, s3), H) \longrightarrow [\texttt{msw}(t(s1), H, s1)], \texttt{expl}(\texttt{reach}(s1, s3), next(H)).$$
$$\texttt{expl}(\texttt{reach}(s1, s3), H) \longrightarrow [\texttt{msw}(t(s1), H, s3)], \texttt{expl}(\texttt{reach}(s3, s3), next(H)).$$
$$\texttt{expl}(\texttt{reach}(s1, s3), H) \longrightarrow [\texttt{msw}(t(s1), H, s4)], \texttt{expl}(\texttt{reach}(s4, s3), next(H)).$$
$$\texttt{expl}(\texttt{reach}(s3, s3), H) \longrightarrow [].$$
$$\texttt{expl}(\texttt{reach}(s4, s3), H) \longrightarrow [\texttt{msw}(t(s4), H, s3)], \texttt{expl}(\texttt{reach}(s3, s3), next(H)).$$

Note that each `expl` generates a sequence of `msw`s. For this example, it is also the case that in a string generated from $\texttt{expl}(\texttt{reach}(s0, s3), H)$, the `msw`s all have instances equal to or later than `H`. It is then immediate that `msw(t(s0),H,s0)` is independent of *any* `msw` generated from $\texttt{expl}(\texttt{reach}(s0, s3), next(H))$. This property holds for an important subclass called *temporally well-formed programs*, defined as follows.

Definition 3 (*Temporally Well-Formed PLP*)
A temporal PLP P is *temporally well formed* if for each clause $(\alpha :- \beta_1, \ldots \beta_n) \in P$:

1. If $\pi(\alpha) \in \texttt{temporal}(P)$, then $\forall\ i, 1 \leq i \leq n$, s.t. $\pi(\beta_i) \in \texttt{temporal}(P)$, $\chi(\alpha) = \chi(\beta_i)$, or $\chi(\alpha)$ is a subterm of $\chi(\beta_i)$.
2. If $\pi(\alpha) \notin \texttt{temporal}(P)$, then there is at most one i, $1 \leq i \leq n$, s.t. $\pi(\beta_i) \in \texttt{temporal}(P)$.
3. Instance arguments $\chi(\alpha)$ or $\chi(\beta_i)$ or their subterms are unified only with other instance arguments, their subterms, or ground terms. □

The first condition ensures that instances of predicates on the rhs of a clause are no earlier than those of the lhs. The second condition ensures that a common temporal instance is created for related temporal predicates. The final condition ensures that the effects of first two are not undone by tainting the temporal arguments.

We represent the set of explanations of a temporal atom α by a special term of the form $\texttt{expl}(\overline{\chi}(\alpha), \chi(\alpha))$. The sets of explanations for an atom can be represented succinctly by DCGs. Such DCGs are called explanation generators.

An atom β is said to be *probabilistic* if it depends on an `msw` atom; i.e., an `msw` atom is probabilistic, and an atom β is probabilistic if some clause whose head unifies with β has a probabilistic atom on the rhs. For a probabilistic atom β, let $e_\beta = [\beta]$ if $\beta = \texttt{msw}(r, t, v)$, $e_\beta = \texttt{expl}(\overline{\chi}(\beta), \chi(\beta))$ if β is a temporal atom, and $e_\beta = \texttt{expl}(\overline{\chi}(\beta), \bot)$ if β is not temporal. A time-abstracted derivation of a query Q is a derivation constructed by ignoring bindings to temporal arguments.

Definition 4 (*Explanation Generator*)
Let P be a temporally well-formed PLP and let Q be a query. Then the explanation generator for Q w.r.t. P, denoted by Γ, is a DCG with non-terminals of the form $\texttt{expl}(t_1, t_2)$ and terminals of the form $\texttt{msw}(t_1, t_2, t_3)$ where t_1, t_2, t_3 are terms, if Γ is the smallest set such that the following holds:

Let β_0 be the selected literal in some step of a time-abstracted derivation of Q. Let c be an instance of a clause in P with β_0 as the lhs atom and $\beta_1, \ldots, \beta_l$ be the probabilistic atoms on the rhs. Let θ be the computed answer substitution for the rhs of c in a time-abstracted derivation. Then $e_{\beta_0}\theta \rightarrow e_{\beta_1}\theta, \ldots, e_{\beta_l}\theta \quad \in \Gamma$. □

An explanation generator is said to be *ground* if every non-terminal symbol $\texttt{expl}(t_1, t_2)$ is such that t_1 is ground and every terminal symbol $\texttt{msw}(t_1, t_2, t_3)$ is such that t_1 and t_3 are ground.

The DCG in Example 2 is the explanation generator for the query `reach(s0,H,s3)` over the program given in Fig. 1(b).

Proposition 1
Let P be a temporally well-formed PLP, Q be a ground query, and Γ be the explanation generator for Q w.r.t. P such that Γ is ground. Then, the language of Γ corresponds to the set of explanations of Q, i.e., $\mathcal{E}_P(Q) = \{sym(s) \mid s \in \mathcal{L}_\Gamma\}$.

Note that the generator in Example 2 can be treated as a stochastic grammar (with the probability of the `msw`s representing the probability of each production), and hence the probability of the query can be computed directly. However, this does not hold in general. For instance, the query `reach(s0,H,s3); reach(s0,H,s4)` considered in the introduction will result in an explanation generator where the productions are not mutually exclusive. To treat such generators, we define the *factoring* algorithm described below.

4.2 Factored Explanation Diagrams

The structure of a Factored Explanation Diagram (FED) closely follows that of a BDD. Similar to a BDD, a FED is a labeled direct acyclic graph with two distinguished leaf nodes: `tt`, representing *true*, and `ff`, representing *false*. While the internal nodes of a BDD are Boolean variables, a FED contains two kinds of internal nodes: one representing terminal symbols of explanations (`msw`s), and the other representing non-terminal symbols of explanations (`expl`s). Thus each path in a FED can be viewed as a *production* in a context free grammar. We will ensure, by construction, that distinct paths in a FED are mutually exclusive and that the set of `msw`s used within a path are all mutually independent. Hence, we can view a FED as a *stochastic grammar*, where each production's probability is given by the (product of) probabilities of `msw`s in that production. We use a partial order among nodes, denoted by "$<$", to construct a FED. The order is used to ensure the mutual exclusion and independence properties stated above.

Definition 5 (*Factored Explanation Diagram*)
A *factored explanation diagram* (FED) is a labeled directed acyclic graph with:

- Four kinds of nodes: `tt`, `ff`, $\texttt{msw}(r,h)$ and $\texttt{expl}(t,h)$, where r is a ground term representing a random process, t is a ground term, and h is an instance term;
- Nodes `tt` and `ff` are 0-ary, and occur only at leaves of the graph;
- $\texttt{msw}(r,h)$ is an n-ary node when r is a random process with n outcomes, and the edges to the n children are labeled with the possible outcomes of r;
- $\texttt{expl}(t,h)$ is a binary node, and the edges to the children are labeled 0 and 1.
- If there is an edge from node x_1 to x_2, then $x_1 < x_2$. □

Note that the multi-valued decision diagrams used in the implementation of PITA (Riguzzi and Swift 2010b) are a special case of FEDs with only `tt`, `ff` and $\texttt{msw}(r,h)$ nodes, where r and h are ground.

We represent non-trivial FEDs by $x?Alts$, where x is the node and *Alts* is the list of edge-label/child pairs. For example, a FED F whose root is an `msw` node is written as $\texttt{msw}(r,h)?[v_1{:}F_1, v_2{:}F_2, \ldots, v_n{:}F_n]$, where $F_1, F_2, \ldots, F_n$ are children FEDs (not all necessarily distinct) and $v_1, v_2, \ldots, v_n$ are possible outcomes of the random process r such that v_i is the label on the edge from F to F_i. Similarly, a FED F whose root is an `expl` node is written as $\texttt{expl}(t,h)?[0{:}F_0, 1{:}F_1]$, where F_0 and F_1 are the children of F with edge labels 0 and 1, respectively.

We now define the ordering relation "$<$" among nodes. Let "$\prec$" be a partial order among instances such that $h_1 \prec h_2$ if h_1 represents an earlier time instance than h_2. Let $\sqsubset$ be a total order among instances, along the lines of a lexicographic ordering, such that $h_1 \prec h_2 \Rightarrow h_1 \sqsubset h_2$. If $h_1 \not\preceq h_2$ and $h_2 \not\preceq h_1$, then h_1 and h_2 are incomparable, denoted as $h_1 \not\sim h_2$. We assume an arbitrary order $<$ among terms.

Definition 6 (*Node order*)
Let x_1 and x_2 be nodes in a FED. Then $x_1 < x_2$ if it matches one of the following:

- $\texttt{msw}(r_1,h_1) < \texttt{msw}(r_2,h_2)$ if $h_1 \sqsubset h_2$ or ($r_1 < r_2$ and $h_1 = h_2$)
- $\texttt{msw}(r_1,h_1) < \texttt{expl}(t_2,h_2)$ if $h_1 \prec h_2$ or $h_1 \not\sim h_2$
- $\texttt{expl}(t_1,h_1) < \texttt{expl}(t_2,h_2)$ if $t_1 < t_2$ and $h_1 \not\sim h_2$. □

A node of the form $\texttt{msw}(r, h, v)$ denotes a valuation of random process r at instance h. From PRISM semantics, two random variables are independent if they differ in their process or instance. Thus two `msw` nodes related by "$<$" are independent. A node n of the form $\texttt{expl}(t, h)$ may represent a set of `msw`s at instance h or later. It follows from the above definition that if m is an `msw` node and $m < n$, then m is independent of every `msw` represented by n. Finally, we can order two `expl` nodes with instances h_1 and h_2 only if they have incompatible instances, since they represent sets of `msw`s at or later than h_1 and h_2, respectively.

Definition 7 (*Binary Operations on FEDs*)
$F_1 \oplus F_2$, where $\oplus \in \{\wedge, \vee\}$, is a FED F derived as follows:

- F_1 is `tt`, and $\oplus = \vee$, then $F = \texttt{tt}$.
- F_1 is `tt`, and $\oplus = \wedge$, then $F = F_2$.
- F_1 is `ff`, and $\oplus = \vee$, then $F = F_2$.
- F_1 is `ff`, and $\oplus = \wedge$, then $F = \texttt{ff}$.
- $F_1 = x_1?[v_{1,1}{:}F_{1,1}, \ldots, v_{1,n_1}{:}F_{1,n_1}]$, $F_2 = x_2?[v_{2,1}{:}F_{2,1}, \ldots, v_{2,n_2}{:}F_{2,n_2}]$:
  c1. $x_1 < x_2$: $F = x_1?[v_{1,1}{:}(F_{1,1} \oplus F_2), \ldots, v_{1,n_1}{:}(F_{1,n_1} \oplus F_2)]$
  c2. $x_1 = x_2$: $F = x_1?[v_{1,1}{:}(F_{1,1} \oplus F_{2,1}), \ldots, v_{1,n_1}{:}(F_{1,n_1} \oplus F_{2,n_1})]$
  c3. $x_1 > x_2$: $F = x_2?[v_{2,1}{:}(F_1 \oplus F_{2,1}), \ldots, v_{2,n_2}{:}(F_1 \oplus F_{2,n_2})]$
  c4. $x_1 \not< x_2, x_2 \not< x_1$: $F = \texttt{expl}(\texttt{merge}(\oplus, F_1, F_2), h)?[0{:}\texttt{ff}, 1{:}\texttt{tt}]$ where h is the common part of the instances of x_1 and x_2. □

Note that Def. 7 is a generalization of the corresponding operations on BDDs. Also, when x_1 and x_2 are both `msw` nodes, since $<$ defines a total order between them, case **c4** will not apply. When the operand nodes cannot be ordered (case **c4**), we generate a placeholder (a `merge` node) indicating the operation to be performed. Such placeholders will be expanded when FEDs are built from an explanation generator (see Def. 8 below). Note that `merge` nodes may be generated only if one or both arguments is a FED rooted at an `expl` node.

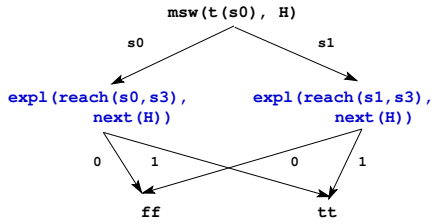
Complements are defined for FEDs exactly as done for BDDs. The complement of F, denoted by $\overline{F}$, is similar to F except that the `tt` and `ff` nodes are swapped.

We now give a procedure for constructing FEDs from an explanation generator for query Q with respect to program P.

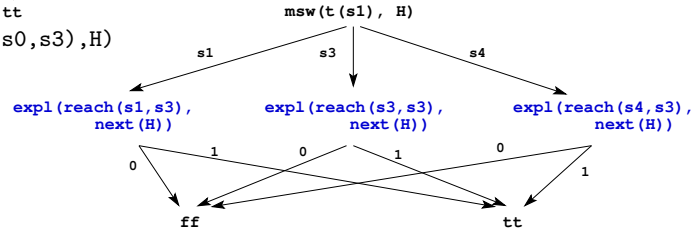
Definition 8 (*Construction of Factored Explanation Diagrams*)
Given an explanation generator Γ, the FED corresponding to goal G, denoted by $\texttt{fed}(G)$, is constructed by mutually recursive functions `fed` and `expand` defined as follows:

- $\texttt{fed}(G)$:
  $= \texttt{msw}(r, h)?[v_1{:}F_1, \ldots, v_n{:}F_n]$ if $G = \texttt{msw}(r, h, v)$, where
  for all i, $F_i = \texttt{tt}$ if $v_i = v$ and $F_i = \texttt{ff}$ otherwise;
  $= \texttt{expand}(G)$ if $G = \texttt{expl}(t, h)$, h is either ground or a variable;
  $= G?[0{:}\texttt{ff}, 1{:}\texttt{tt}]$ otherwise.
- $\texttt{expand}(\beta_0) = F$ where $\{(\beta_0 \rightarrow \beta_{1,1}, \ldots, \beta_{1,n_1}), \ldots, (\beta_0 \rightarrow \beta_{k,1}, \ldots, \beta_{k,n_k})\}$ is the set of all clauses in Γ with β_0 on the left hand side, and

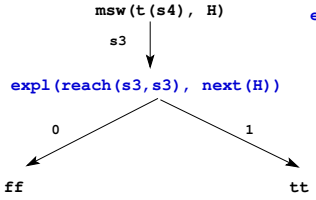
$$F = \bigvee_{i=1}^{k} \bigwedge_{j=1}^{n_k} \texttt{fed}(\beta_{i,j})$$



(a) FED for `expl(reach(s0,s3),H)`



(b) FED for `expl(reach(s1,s3),H)`



(c) FED for `expl(reach(s4,s3),H)`

Fig. 2. FEDs for Example 2

- $\texttt{expand}(\texttt{merge}(\oplus, F_1, F_2)) = F$ where
  - if $F_1 = \texttt{expl}(t_1, h_1)?[0{:}F_{1,0}, 1{:}F_{1,1}]$ and $F_1' = \texttt{expand}(\texttt{expl}(t_1, h_1))$, then $F = \oplus((\overline{F_1'} \wedge F_{1,0}) \vee (F_1' \wedge F_{1,1}), F_2)$.
  - if $F_2 = \texttt{expl}(t_2, h_2)?[0{:}F_{2,0}, 1{:}F_{2,1}]$ and $F_2' = \texttt{expand}(\texttt{expl}(t_2, h_2))$, then $F = \oplus(F_1, (\overline{F_2'} \wedge F_{2,0}) \vee (F_2' \wedge F_{2,1}))$.

Omitting the last (`merge`) case, the definition can be understood as follows. The key function is `expand`, which constructs a FED from a set of clauses in Γ. Function `fed` (i) defines FEDs for `msw` nodes, and (ii) controls the expansion by stopping expansion of `expl` nodes when the instance is partially specified. The `merge` case processes a binary operation over FEDs that was left frozen earlier. In our implementation, the above definition is turned into a tabled logic program. Furthermore, FEDs are maintained using a dictionary to ensure that they have a DAG structure.

Example 3
Three of the four FEDs for the explanation generator in Example 2 are shown in Fig. 2. The FED for `expl(reach(s3,s3),H)`, not shown in the figure, is `tt`.

4.3 Computing Probabilities from FEDs

Recall that a factored explanation diagram can be viewed as a *stochastic grammar*. Following (Etessami and Yannakakis 2009), we can generate a set of simultaneous equations from the stochastic grammar and find the probability of the language from the least solution of the equations. The generation of equations from the factored representation of explanations is formalized below.

Definition 9 (*Temporal Abstraction*)
Given a temporal PLP P, the temporal abstraction of a term t, denoted by $\texttt{abs}(t)$, is $\overline{\chi}(t)$ if $\pi(t) \in \texttt{temporal}(P)$ and $\chi(t)$ is non-ground, and it is t otherwise. That is, for a term t with a temporal predicate as root, $\texttt{abs}(t)$ omits its instance argument if that argument is not ground. □

// x_i: prob($\texttt{expl}(\texttt{reach}(s\mathbf{i}, \_, s3))$); t_{ij}: prob($\texttt{msw}(t(s\mathbf{i}), \_, s\mathbf{j})$

$x_0 = t_{00} * x_0 + t_{01} * x_1$	$t_{00} = .5$	$t_{14} = .5$
$x_1 = t_{11} * x_1 + t_{13} * x_3 + t_{14} * x_4$	$t_{01} = .3$	$t_{43} = 1$
$x_3 = 1$	$t_{11} = .4$	
$x_4 = t_{43} * x_3$	$t_{13} = .1$	

Fig. 3. Set of equations generated from the set of FEDs of Example 3

Definition 10 (*Distribution*)
Let ρ be a random process specified in a PLP P. The set of values produced by ρ is denoted by $\texttt{values}_P(\rho)$. The distribution of ρ, denoted by $\texttt{distr}_P(\rho)$, is a function from the set of all terms over the Herbrand Universe of P to $[0, 1]$ such that $\sum_{v \in \texttt{values}_P(\rho)} \texttt{distr}_P(\rho)(v) = 1$ □

Definition 11 (*System of Equations for PLP*)
Let Γ be an explanation generator, `fed` be the relation defined in Def. 8, V be a countable set of variables, and f be a one-to-one function from terms to V. The system of polynomial equations $E_{(\Gamma,V,f)} = \{(f(\texttt{abs}(G)) = \mathcal{P}(F)) \mid \texttt{fed(G,F)}$ holds$\}$, where $\mathcal{P}$ is a function that maps FEDs to polynomials, is defined as follows:

$$\begin{aligned}
\mathcal{P}(\texttt{ff}) &= 0 \\
\mathcal{P}(\texttt{tt}) &= 1 \\
\mathcal{P}(\texttt{msw}(r,h)?[v_1{:}F_1, \ldots, v_n{:}F_n]) &= \textstyle\sum_{i=1}^{n} \texttt{distr}(r)(v_i) * \mathcal{P}(F_i) \\
\mathcal{P}(\texttt{expl}(t,h)?[0{:}F_0, 1{:}F_1]) &= f(\texttt{abs}(\texttt{expl}(t,h))) * \mathcal{P}(F_1) \\
&\quad + (1 - f(\texttt{abs}(\texttt{expl}(t,h)))) * \mathcal{P}(F_0)
\end{aligned}$$

The set of equations for Example 3 is shown in Fig. 3.

The implementation of the above definition is such that shared FEDs result in shared variables in the equation system. The correspondence between a PLP in factored form and the set of monotone equations permits us to compute the probability of query answers in terms of the least solution to the system of equations.

Theorem 2 (*Factored Forms and Probability*)
Let Γ be an explanation generator for query Q w.r.t. program P. Let V be a set of variables and let f be a one-to-one function from terms to V. Then, X is the probability of a query answer Q evaluated over P, denoted by $\texttt{prob}(Q, X)$, if X is the value of the variable $f(\texttt{expl}(\overline{\chi}(Q), \chi(Q)))$ in the least solution of the corresponding set of equations, $E_{(\Gamma,V,f)}$.

The following properties show that the algorithm for finding probabilities of a query answer is well defined.

Proposition 3 (*Monotonicity*)
If Γ is an explanation generator in factored form, V is a set of variables and f is a one-to-one function as required by Def. 11, then the system of equations $E_{(\Gamma,V,f)}$ is monotone in $[0, 1]$.

Monotone systems have the following important property:

Proposition 4 (*Least Solution (Etessami and Yannakakis 2009)*)
Let E be a set of polynomial equations which is monotone in $[0, 1]$. Then E has a least solution in $[0, 1]$. Furthermore, a least solution can be computed to within an arbitrary approximation bound by an iterative procedure.

Note that FEDs may be non-regular since `expl` nodes may have other `expl` nodes as children, and hence the resulting equations may be non-linear. Proposition 4 establishes that the probability of query answers can be effectively computed even when the set of equations is non-linear.

The probability of the language of explanations in Example 2 (via the equations in Fig. 3) is given by the value of x_0 in the least solution, which is 0.6.

5 Applications

We now present two model checkers that demonstrate the utility of PIP.

PCTL: The syntax of an illustrative fragment of PCTL is given by:

$$\begin{array}{rl} SF ::= & \texttt{prop}(A) \mid \texttt{neg}(SF) \mid \texttt{and}(SF_1, SF_2) \mid \texttt{pr}(PF, \texttt{gt}, B) \mid \texttt{pr}(PF, \texttt{geq}, B) \\ PF ::= & \texttt{until}(SF_1, SF_2) \mid \texttt{next}(SF) \end{array}$$

Here, A is a proposition and B is a real number in $[0, 1]$. The logic partitions formulae into *state* formulae (denoted by SF) and *path* formulae (denoted by PF). State formulae are given a non-probabilistic semantics: a state formula is either true or false at a state. For example, formula `prop(a)` is true at state s if proposition `a` holds at s; a formula $\texttt{and}(SF_1, SF_2)$ holds at s if both SF_1 and SF_2 hold at s. The formula $\texttt{pr}(PF, \texttt{gt}, B)$ holds at a state s if the probability p of the set of all paths on which the path formula PF holds is such that $p > B$ (similarly, $p \geq B$ for `geq`).

The formula $\texttt{until}(SF_1, SF_2)$ holds on a given path $s_0, s_1, s_2, \ldots$ if SF_2 holds on state s_k for some $k \geq 0$ and SF_1 holds for all s_i, $0 \leq i < k$. Full PCTL has a *bounded* `until` operator, which imposes a fixed upper bound on k; we omit its treatment since it has a straightforward non-fixed-point semantics. The *probability* of a path formula PF at a state s is the sum of probabilities of all paths starting at s on which PF holds. This semantics is directly encoded as the probabilistic logic program given in Fig. 4. In this encoding, `trans/3` encodes the transition relation of a Discrete Time Markov Chain (DTMC). Note that the encoding violates PRISM's exclusiveness condition (e.g., consider `pmodels` when $\texttt{models}(S, SF_2)$ holds) as well as the finite support condition (due to cycles in the DTMC). Observe the use of an abstract instance argument "_" in the invocation of `pmodels/3` from `pmodels/2`. This ensures that an explanation generator can be effectively computed for any query to `pmodels/2`.

GPL: GPL is an expressive logic based on the modal mu-calculus for probabilistic systems (Cleaveland et al. 2005). GPL subsumes PCTL and PCTL* in expressiveness. GPL is designed for model checking *reactive probabilistic transition systems* (RPLTS), which are a generalization of DTMCs. In an RPLTS, a state may have zero or more outgoing transitions, each labeled by a distinct action symbol. Each action has a distribution on destination states.

Syntactically, GPL has *state* and *fuzzy* formulae, where the state formulae are similar to those of PCTL. The fuzzy formulae are, however, significantly more

```
% State Formulae
models(S, prop(A)) :-
    holds(S, A).
models(S, neg(A)) :-
    not models(S, A).
models(S, and(SF1, SF2)) :-
    models(S, SF1),
    models(S, SF2).
models(S, pr(PF, gt, B)) :-
    prob(pmodels(S, PF), P),
    P > B.
models(S, pr(PF, geq, B)) :-
    prob(pmodels(S, PF), P),
    P >= B.
```

```
% Path Formulae
pmodels(S, PF) :-
    pmodels(S, PF, _).

:- table pmodels/3.
pmodels(S, until(SF1, SF2), H) :-
    models(S, SF2).
pmodels(S, until(SF1, SF2), H) :-
    models(S, SF1),
    trans(S, H, T),
    pmodels(T, until(SF1, SF2), next(H)).
pmodels(S, next(SF), H) :-
    trans(S, H, T),
    models(T, SF).

temporal(pmodels/3-3).
```

Fig. 4. Model checker for a fragment of PCTL

```
%% pmodels(S, PF, H): S is in the model of fuzzy formula PF at or after instant H
%% smodels(S, SF): S is in the model of state formula SF
```

```
pmodels(S, sf(SF), H) :-
    smodels(S, SF).
pmodels(S, and(F1,F2), H) :-
    pmodels(S, F1, H),
    pmodels(S, F2, H).
pmodels(S, or(F1,F2), H) :-
    pmodels(S, F1, H);
    pmodels(S, F2, H).
pmodels(S, diam(A, F), H) :-
    trans(S, A, SW),
    msw(SW, H, T),
    pmodels(T, F, [T,SW|H]).
pmodels(S, box(A, F), H) :-
    findall(SW, trans(S,A,SW), L),
    all_pmodels(L, S, F, H).
```

```
pmodels(S, form(X), H) :-
    tabled_pmodels(S, X, H1), H=H1.

all_pmodels([], _, _, _H).
all_pmodels([SW|Rest], S, F, H) :-
    msw(SW, H, T),
    pmodels(T,F,[T,SW|H]),
    all_pmodels(Rest, S, F, H).

:- table tabled_pmodels/3.
tabled_pmodels(S,X,H) :-
    fdef(X, lfp(F)),
    pmodels(S, F, H).
```

Fig. 5. Fragment of a model checker for fuzzy formulae in GPL

expressive. The syntax of GPL, in equational form, is given by:

$$
\begin{array}{rcl}
SF & ::= & \texttt{prop}(A) \mid \texttt{neg}(\texttt{prop}(A)) \mid \texttt{and}(SF, SF) \mid \texttt{or}(SF, SF) \\
 & \mid & \texttt{pr}(PF, \texttt{gt}, B) \mid \texttt{pr}(PF, \texttt{lt}, B) \mid \texttt{pr}(PF, \texttt{geq}, B) \mid \texttt{pr}(PF, \texttt{leq}, B) \\
PF & ::= & \texttt{sf}(SF) \mid \texttt{form}(X) \mid \texttt{and}(PF, PF) \mid \texttt{or}(PF, PF) \mid \texttt{diam}(A, PF) \mid \texttt{box}(A, PF) \\
D & ::= & \texttt{def}(X, \texttt{lfp}(PF)) \mid \texttt{def}(X, \texttt{gfp}(PF))
\end{array}
$$

Formula $\texttt{diam}(A, PF)$ holds at a state if there is a transition labeled A after which PF holds; $\texttt{box}(A, PF)$ holds at a state if PF holds after *every transition labeled* A. In the syntax, X denotes a formula variable defined using least- and greatest-fixed-point equations in D using `lfp` and `gfp`, respectively. Formulae are specified as a set of definitions. GPL admits only alternation-free fixed-point formulae.

A part of the model checker for GPL that deals with fuzzy formulae is shown in Fig. 5. Note that fuzzy formulae have probabilistic semantics, and, at the same time, may involve conjunctions or disjunctions of other fuzzy formulae. Thus,

for example, when evaluating $\texttt{models}(s, \texttt{and}(PF_1, PF_2), H)$, the explanations of $\texttt{models}(s, PF_1, H)$ and $\texttt{models}(s, PF_2, H)$ may not be pairwise independent. Thus recursion-free fuzzy formulae cannot be evaluated in PRISM, but can be evaluated using the BDD-based algorithms of ProbLog and PITA. In contrast, *recursive* fuzzy formulae can be evaluated using PIP.

Recursive Markov Chains: Recursive Markov Chains (RMCs) use special *call* and *return* nodes to provide a model of DTMCs with a recursive-call capability (Etessami and Yannakakis 2009). We can check reachability in an RMC by generating a corresponding RPLTS, and model checking a standard GPL formula w.r.t. that RPLTS. See (Gorlin et al. 2012) for the details.

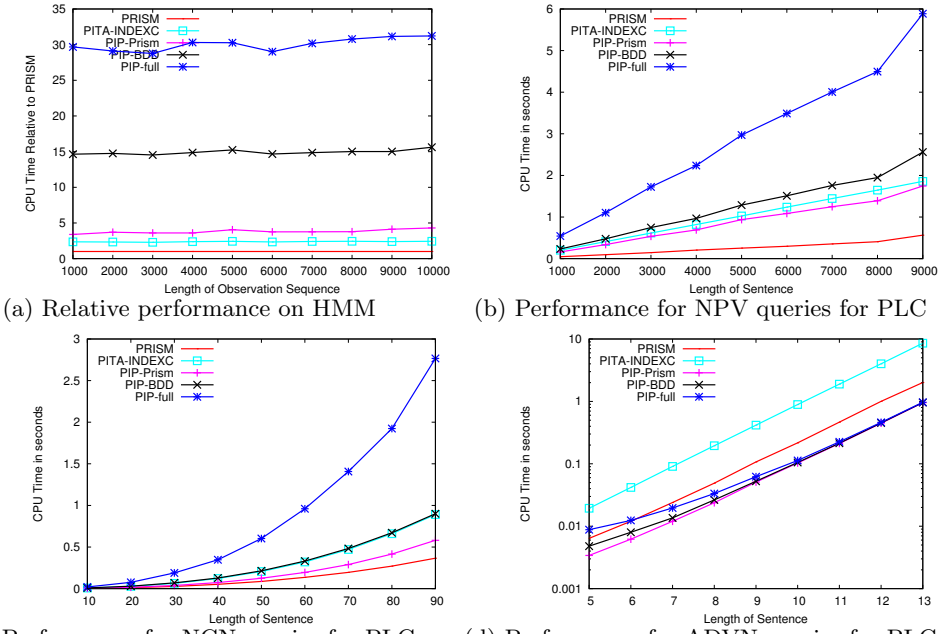
6 Experimental Results

PIP has been implemented using the XSB tabled logic programming system (Swift et al. 2012). An explanation generator is constructed by using query evaluation under the well-founded semantics by redefining `msw`s to backtrack through their potential values and to have the *undefined* truth value. This generates a *residual* program in XSB that captures the dependencies between the original goal and the `msw`s (now treated as undefined values). In the first partial implementation, called **PIP-Prism**, the probabilities are computed directly from the residual program. Note that such a computation will be correct if PRISM's restrictions are satisfied. In general, however, we materialize the explanation generator. The second partial implementation, called **PIP-BDD**, constructs BDDs from the explanation generator and computes probabilities from the BDD. Note that **PIP-BDD** will be correct when the finiteness restriction holds. The full implementation of PIP, called **PIP-full**, is obtained by constructing a set of FEDs from the explanation generator (Def. 8), generating polynomial equations from the set of FEDs (Def. 11), and finally finding the least solution to the set of equations. The final equation solver is implemented in C. All other parts of the three implementations, including the BDD and FED structures, are completely implemented in tabled Prolog.

We present two sets of experimental results, evaluating the performance of PIP on (1) programs satisfying PRISM's restrictions; and (2) a program for model checking PCTL formulae. The results were collected on a machine running Mac OS X 10.6.8, with a 4-core 2.5 GHz Intel Core i5 processor and 4 GB of memory. For the first set of results, we compare PIP with PRISM v2.0.3 and PITA in XSB v3.3.6. The model checking results were compared with that of Prism Model Checker v4.0.3.

Performance on PRISM Programs: Note that all three implementations—PIP-Prism, PIP-BDD and PIP-full may be used to evaluate PRISM programs.

Hidden Markov Model (HMM): We used the simple 2-state gene sequence HMM from (Christiansen and Gallagher 2009) (also used in (Riguzzi and Swift 2011)) for our evaluation. We measured the CPU time taken by the versions of PIP, PRISM and PITA-INDEXC (Riguzzi and Swift 2011) (a version of PITA that does not use BDDs and uses PRISM's assumption) to evaluate the probability of a given observation sequence, for varying sequence lengths. The observation sequence itself was embedded as a set of facts (instead of an argument list). This makes table accesses fast even when shallow indices are used. The number of nodes in the explanation graph (e.g., FED) is linear in the size of the argument list. The performance of the three PIP versions and PITA-INDEXC, relative to PRISM, is shown in Fig. 6(a).



(a) Relative performance on HMM

(b) Performance for NPV queries for PLC

(c) Performance for NCN queries for PLC

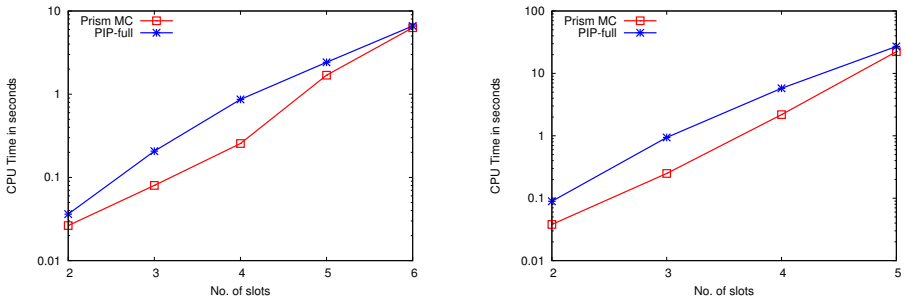
(d) Performance for ADVN queries for PLC

Fig. 6. Performance of PIP on PRISM Programs

CPU times are normalized using PRISM's time as the baseline. Observe that the PIP-Prism and PITA-INDEXC perform similarly: about 3.5 to 4 times slower than PRISM. Construction of BDDs (done in PIP-BDD, but not in PIP-Prism) increases this overhead by a factor of 4. Construction of full-fledged FEDs, generating polynomial equations and solving them (done only in PIP-full) further doubles the overhead. We find that the equation-solving time is generally negligible.

Probabilistic Left Corner Parsing (PLC): This example was adapted from PRISM's example suite, parameterizing the length of the input sequence to be parsed. We measured the CPU time taken by the three versions of PIP, PRISM and PITA. The performance on three queries— *NPV*, *NCN*, and *ADVN*, each encoding a different class of strings— is shown in Fig. 6(b)–(d). As in the HMM example, the sequences are represented as facts instead of lists. The number of nodes in the explanation graph is linear in string length for *NPV* queries; and quadratic for *NCN* and *ADVN* queries. The processing time needed for generating explanations is linear for *NPV* queries; quadratic for *NCN* queries; and exponential in *ADVN* queries. Note that the y-axis is logarithmic in Fig 6(d), and linear in the rest. The performance on *NPV* is similar to that on HMM, with one exception: PIP-Prism marginally outperforms PITA-INDEXC. The differences are more noticeable on *NCN* and become significant on *ADVN*, indicating the relative efficiency of computing explanations in PIP. The performance of PIP relative to PRISM shows smaller overheads on *NPV* and *NCN* (compared to HMM) and exhibits a reversal on *ADVN*. The differences between PIP-Prism, PIP-BDD and PIP-full narrow in *NPV* and become negligible in *ADVN* due to the small size of explanations.

Performance of the PCTL Model Checker: We evaluated the performance of PIP-full for supporting a PCTL model checker (encoded as shown in Fig. 4). We



(a) Number of processes $N = 5$ (b) Number of processes $N = 6$

Fig. 7. Performance of PCTL model checking using PIP and the Prism model checker for Synchronous Leader Election protocol

compared the performance of PIP-based model checker with that of the widely-used Prism model checker (Kwiatkowska et al. 2011). We show the performance of PIP and the Prism model checker on the Synchronous Leader Election Protocol (Itai and Rodeh 1981) for computing the probability that eventually a leader will be elected. Fig. 7 shows the CPU time used to compute the probabilities of this property on systems with different numbers of processes (N) and number of slots used by the protocol (*slots*). Observe that our high-level implementation of a model checker based on PIP performs within a factor of 3 of the Prism model checker (note: the y-axis on these graphs is logarithmic). Moreover, the two model checkers show similar performance trends with increasing problem instances. However, it should be noted that the Prism model checker uses a BDD-based representation of reachable states, which can, in principle, scale better to large state spaces compared to the explicit state representation used in our PIP-based model checker.

It should be noted that GPL is strictly more expressive than PCTL. When a PCTL formula is encoded in GPL, the GPL model checker performs the same set of operations as the PCTL one. While GPL is strictly more expressive, we have not yet encountered a practical property that needs the additional expressive power. Hence we have not separately reported the performance of the GPL model checker.

7 Conclusions

In this paper, we have shown that in order to formulate the problem of probabilistic model checking in probabilistic logic programming, one needs an inference algorithm that functions correctly even when finiteness, mutual-exclusion, and independence assumptions are simultaneously violated. We have presented such an inference algorithm, PIP, implemented it in XSB Prolog, and demonstrated its practical utility by using it as the basis for encoding model checkers for a rich class of probabilistic models and temporal logics.

For future work, we plan to refine and strengthen the implementation of PIP. We also plan to explore more substantial model-checking case studies. It would be interesting to study whether optimizations to exploit data independence and symmetry, which are easily enabled by high-level encodings of model checkers, will be effective for probabilistic systems as well.

Acknowledgments. We thank the reviewers for valuable comments. This research was supported in part by NSF Grants CCF-1018459, CCF-0926190, CCF-0831298, AFOSR Grant FA9550-09-1-0481, and ONR Grant N00014-07-1-0928.

References

Baier, C. 1998. On Algorithmic Verification Methods for Probabilistic Systems. Habilitation thesis, Fakultät für Mathematik & Informatik, Universität Mannheim.

Christiansen, H. and Gallagher, J. P. 2009. Non-discriminating arguments and their uses. In *ICLP*. LNCS, vol. 5649. 55–69.

Cleaveland, R., Iyer, S. P., and Narasimha, M. 2005. Probabilistic temporal logics via the modal mu-calculus. *Theor. Comput. Sci. 342,* 2-3, 316–350.

De Raedt, L., Kimmig, A., and Toivonen, H. 2007. ProbLog: A probabilistic Prolog and its application in link discovery. In *IJCAI*. 2462–2467.

Delzanno, G. and Podelski, A. 1999. Model checking in CLP. In *TACAS*. 223–239.

Du, X., Ramakrishnan, C. R., and Smolka, S. A. 2000. Tabled resolution + constraints: A recipe for model checking real-time systems. In *IEEE RTSS*.

Etessami, K. and Yannakakis, M. 2009. Recursive Markov chains, stochastic grammars, and monotone systems of nonlinear equations. *J. ACM 56,* 1.

Farwer, B. and Leuschel, M. 2004. Model checking object Petri nets in Prolog. In *PPDP*. 20–31.

Friedman, N., Getoor, L., Koller, D., and Pfeffer, A. 1999. Learning probabilistic relational models. In *IJCAI*. 1300–1309.

Getoor, L. and Taskar, B. 2007. *Introduction to Statistical Relational Learning*. The MIT Press.

Gorlin, A., Ramakrishnan, C. R., and Smolka, S. A. 2012. Model checking with probabilistic tabled logic programming. Manuscript, available from `http://www.cs.stonybrook.edu/~cram/probmc`.

Gupta, G., Bansal, A., Min, R., Simon, L., and Mallya, A. 2007. Coinductive logic programming and its applications. In *ICLP*. 27–44.

Gupta, G. and Pontelli, E. 1997. A constraint based approach for specification and verification of real-time systems. In *Proceedings of the Real-Time Systems Symposium*.

Gutmann, B., Jaeger, M., and Raedt, L. D. 2010. Extending ProbLog with continuous distributions. In *Proceedings of ILP*.

Gutmann, B., Thon, I., Kimmig, A., Bruynooghe, M., and Raedt, L. D. 2011. The magic of logical inference in probabilistic programming. *TPLP 11,* 4–5, 663–680.

Hansson, H. and Jonsson, B. 1994. A Logic for Reasoning about Time and Reliability. *Formal Aspects of Computing 6,* 5, 512–535.

Islam, M. A., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2011. Inference in Probabilistic Logic Programs with Continuous Random Variables. *ArXiv e-prints*. `http://arxiv.org/abs/1112.2681`.

Itai, A. and Rodeh, M. 1981. Symmetry breaking in distributive networks. In *FOCS*. 150–158.

Kersting, K. and Raedt, L. D. 2000. Bayesian logic programs. In *ILP Work-in-progress reports*.

Kersting, K. and Raedt, L. D. 2001. Adaptive Bayesian logic programs. In *Proceedings of ILP*.

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

Kucera, A., Esparza, J., and Mayr, R. 2006. Model checking probabilistic pushdown automata. *Logical Methods in Computer Science 2,* 1.

Kwiatkowska, M., Norman, G., and Parker, D. 2011. PRISM 4.0: Verification of probabilistic real-time systems. In *23rd CAV*. LNCS, vol. 6806. 585–591.

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

Milner, R., Parrow, J., and Walker, D. 1992. A calculus of mobile processes, Parts I and II. *Information and Computation 100(1)*, 1–77.

Muggleton, S. 1996. Stochastic logic programs. In *Advances in inductive logic programming*.

Mukhopadhyay, S. and Podelski, A. 1999. Beyond region graphs: Symbolic forward analysis of timed automata. In *FSTTCS*. 232–244.

Mukhopadhyay, S. and Podelski, A. 2000. Model checking for timed logic processes. In *Computational Logic*. 598–612.

Narman, P., Buschle, M., Konig, J., and Johnson, P. 2010. Hybrid probabilistic relational models for system quality analysis. In *Proceedings of EDOC*.

Nilsson, U. and Maluszyński, J. 2000. *Logic, Programming and Prolog*. `http://www.ida.liu.se/~ulfni/lpp`.

Pemmasani, G., Ramakrishnan, C. R., and Ramakrishnan, I. V. 2002. Efficient model checking of real time systems using tabled logic programming and constraints. In *International Conference on Logic Programming (ICLP)*. LNCS. Springer.

Poole, D. 2008. The independent choice logic and beyond. In *Probabilistic ILP*. 222–243.

Ramakrishna, Y. S., Ramakrishnan, C. R., Ramakrishnan, I. V., Smolka, S. A., Swift, T., and Warren, D. S. 1997. Efficient model checking using tabled resolution. In *CAV*. LNCS, vol. 1254. Springer, 143–154.

Ramakrishnan, C. R. 2001. A model checker for value-passing mu-calculus using logic programming. In *PADL*. LNCS, vol. 1990. Springer, 1–13.

Ramakrishnan et al, C. R. 2000. XMC: A logic-programming-based verification toolset. In *CAV*. Number 1855 in LNCS. 576–580.

Richardson, M. and Domingos, P. 2006. Markov logic networks. *Machine Learning*.

Riguzzi, F. and Swift, T. 2010a. An extended semantics for logic programs with annotated disjunctions and its efficient implementation. In *Italian Conference on Computational Logic*. CEUR Workshop Proceedings, vol. 598.

Riguzzi, F. and Swift, T. 2010b. Tabling and answer subsumption for reasoning on logic programs with annotated disjunctions. In *Technical Communications of the International Conference on Logic Programming*. 162–171.

Riguzzi, F. and Swift, T. 2011. The PITA system: Tabling and answer subsumption for reasoning under uncertainty. *TPLP 11,* 4–5, 433–449.

Santos Costa, V., Page, D., Qazi, M., and Cussens, J. 2003. CLP($\mathcal{BN}$): Constraint logic programming for probabilistic knowledge. In *Proceedings of the 19th Conference on Uncertainty in Artificial Intelligence (UAI03)*. Acapulco, Mexico, 517–524.

Sarna-Starosta, B. and Ramakrishnan, C. R. 2003. Constraint-based model checking of data-independent systems. In *International Conference on Formal Engineering Methods (ICFEM)*. Lecture Notes in Computer Science, vol. 2885. Springer, 579–598.

Sato, T. and Kameya, Y. 1997. PRISM: a symbolic-statistical modeling language. In *IJCAI*.

Singh, A., Ramakrishnan, C. R., and Smolka, S. A. 2008. A process calculus for mobile ad hoc networks. In *10th International Conference on Coordination Models and Languages (COORDINATION)*. LNCS, vol. 5052. Springer, 296–314.

Swift, T., Warren, D. S., et al. 2012. The XSB logic programming system, Version 3.3. `http://xsb.sourceforge.net`.

Vennekens, J., Denecker, M., and Bruynooghe, M. 2009. CP-logic: A language of causal probabilistic events and its relation to logic programming. *TPLP*.

Vennekens, J., Verbaeten, S., and Bruynooghe, M. 2004. Logic programs with annotated disjunctions. In *ICLP*. 431–445.

Wang, J. and Domingos, P. 2008. Hybrid Markov logic networks. In *Proceedings of AAAI*.

Wojtczak, D. and Etessami, K. 2007. PReMo: an analyzer for probabilistic recursive models. In *TACAS*.

Yang, P., Ramakrishnan, C. R., and Smolka, S. A. 2004. A logical encoding of the pi-calculus: Model checking mobile processes using tabled resolution. *International Journal on Software Tools for Technology Transfer (STTT) 6,* 1, 38–66.