Verification and Probabilistic Logic Programming

C. R. Ramakrishnan, Andrey Gorlin

Stony Brook University

ICLP 2016 Autumn School

Model Checking

$$S \overset{?}{\models} \varphi$$

Model Checking

$$S \overset{?}{\models} \varphi$$

- S: System specification or implementation
  Automaton, transition system, protocol specification, process expression, code, ...
- φ: Property specification
  Temporal logic formula

Verification of other forms (e.g. refinement checking) are not considered in this talk.

Executable Specifications and Logic Programs

Logic Programming is well-recognized for its suitability for

- Writing interpreters for languages starting from high-level declarative specifications
- Constructing state spaces and searching through them
- Performing meaning-preserving abstractions using clever data representations

Executable Specification of Operational Semantics

$$\frac{e_1 \to e_1'}{(e_1\ e_2) \to (e_1'\ e_2)}$$

```
step(app(E1, E2), app(E1P, E2)) :-
    step(E1, E1P).
```

$$\frac{e_2 \to e_2'}{(v_1\ e_2) \to (v_1\ e_2')}$$

```
step(app(V1, E2), app(V1, E2P)) :-
    isValue(V1),
    step(E2, E2P).
```

$$(\lambda x.\ e_1)\ \ v_2 \to [x \mapsto v_2]e_1$$

```
step(app(lambda(X, E1), V2), E2) :-
    isValue(V2),
    subst(X, V2, E1, E2).

isValue(lambda(_, _)).
```

[Call-By-Value Lambda Calculus]

Substitution

$$\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.\ t) & = & \lambda y.\ [x \mapsto s]t & \text{if } x \neq y \text{ and } y \notin fv(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}$$

Substitution

$$\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.\ t) & = & \lambda y.\ [x \mapsto s]t & \text{if } x \neq y \text{ and } y \notin fv(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}$$

- This definition becomes complete only when we consider α-renaming.
- We can program α-renaming explicitly, or better still. . .

Substitution

$$\begin{array}{lcll}
[x \mapsto s]x & = & s & \\
[x \mapsto s]y & = & y & \text{if } y \neq x \\
[x \mapsto s](\lambda y.\ t) & = & \lambda y.\ [x \mapsto s]t & \text{if } x \neq y \text{ and } y \notin fv(s) \\
[x \mapsto s](t_1\ t_2) & = & ([x \mapsto s]t_1)\ ([x \mapsto s]t_2) &
\end{array}$$

- This definition becomes complete only when we consider α-renaming.
- We can program α-renaming explicitly, or better still…
- With suitable restrictions on the way λ-terms are written,
  - represent variables in lambda-terms with logical variables, and
  - use the "standardization" done by resolution to perform the needed α-renaming.
- We used such a strategy to encode model checkers for the *pi*-calculus [Yang et al, VMCAI'03].

Executable Specification of Abstract Semantics

$$\frac{\texttt{p = \&q}}{\texttt{p} \to \texttt{q}}$$

$$\frac{\texttt{p = q} \quad \texttt{q} \to r}{\texttt{p} \to r}$$

$$\frac{\texttt{p = *q} \quad \texttt{q} \to r \quad r \to s}{\texttt{p} \to s}$$

$$\frac{\texttt{*p = q} \quad \texttt{p} \to r \quad \texttt{q} \to s}{r \to s}$$

```
pts(P,Q) :-
    stmt(v(P), addr(Q)).

pts(P,R) :-
    stmt(v(P), v(Q)),
    pts(Q, R).

pts(P,S) :-
    stmt(v(P), star(Q)),
    pts(Q, R), pts(R, S).

pts(R, S) :-
    stmt(star(P), v(Q)),
    pts(P, R),
    pts(Q, S).
```

[Anderson's Context-Insensitive Points-To Analysis]

Demand-Driven Analysis

Compute only the information necessary to determine the *may-point-to* set of x. [Heinze et al., PLDI 2001]

- Tabled query evaluation is naturally demand-driven, but ...

Demand-Driven Analysis

Compute only the information necessary to determine the *may-point-to* set of x. [Heinze et al., PLDI 2001]

- Tabled query evaluation is naturally demand-driven, but ...
- Clauses of the form `pts(R, S) :- stmt(star(P), v(Q)), ...` lead to generate-and-test evaluation.

Demand-Driven Analysis

Compute only the information necessary to determine the *may-point-to* set of x. [Heinze et al., PLDI 2001]

- Tabled query evaluation is naturally demand-driven, but …
- Clauses of the form `pts(R, S) :- stmt(star(P), v(Q)), ...` lead to generate-and-test evaluation.
- **Trick:** replicate *points-to* (`pts`) as *pointed-to-by* (`ptb`).

```
pts(R, S) :-
    stmt(star(P), v(Q)),
    pts(P, R),
    pts(Q, S).
```

⇒

```
pts(R, S) :-
    ptb(R, P),
    stmt(star(P), v(Q)),
    pts(Q, S).
```

[PPDP'05]

Model Checking as Query Evaluation

- Encode the semantic equations of temporal logics as a logic program.
- Query evaluation over the program will perform model checking.

We consider traditional query evaluation methods developed and used in LP literature.

- Rybalchenko et al take a very different (and neat) approach: posing verification problems as constraint solving over Horn Constraints.
- Verification of certain infinite-state systems is enabled by the construction and use of specialized Horn Constraint solvers.

Executable Specification of Semantic Equations

$[\![\cdot]\!]$ is the smallest set such that:

% $[\![p]\!]$ = states satisfying prop. p.
$[\![p]\!] = \{s \mid p \in AP(s)\}$

% Conjunction:
$[\![\varphi_1 \wedge \varphi_2]\!] = [\![\varphi_1]\!] \cap [\![\varphi_2]\!]$

% $[\![\mathrm{EF}\ f]\!] =$
% $\quad \{s \mid \exists t.\ \ s \xrightarrow{*} t \text{ and } t \in [\![f]\!]\}$

$$[\![EF\varphi]\!] = [\![\varphi]\!] \cup \{s \mid \exists t.\ s \rightarrow t, t \in [\![EF\varphi]\!]\}$$

$\vdots$

```
models(S,prop(P)) :-
    holds(S, P).

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)) :-
    findall(T, trans(S, T), L),
    all_models(T, af(F)).
...
```

[Computation Tree Logic's Semantics (Fragment)]

Model Checking as Query Evaluation

Mobile Ad-Hoc Networks

Parameterized Systems

Multi-Agent Systems

Model Checkers

`Infinite-State Systems`

π-Calculus

Model Checking as Query Evaluation

Mobile Ad-Hoc Networks

Parameterized Systems

Multi-Agent Systems

Model Checkers

`Infinite-State Systems`

π-Calculus

Probabilistic Systems

System Models: Kripke Structures and LTSs

- State transition systems: *directed graphs* with
  - Vertices representing system states, and
  - Edges representing transitions between states
- Labels on edges representing "actions": *Labeled Transition Systems*.
- Vertices associated with sets of propositions: *Kripke structures*.

Property Specification: Temporal Logics I

Computational Tree Logic (CTL):

φ	$\rightarrow$	$p \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$	Propositions, logical connectives
	$\mid$	$E\phi \mid A\phi$	State formulae
ϕ	$\rightarrow$	$X\varphi \mid \varphi_1 \; U \; \varphi_2 \mid \varphi_1 R \; \varphi_2$	Path formulae

- Semantics of the CTL is usually given in terms of computation trees of Kripke Structures.
- The meaning of path formulae are given in terms of sets of paths (runs, or sequences of states) of the system.
- Informally, $\varphi_1 \; U \; \varphi_2$ holds in a run means φ_1 holds in every state of the run until a state where φ_2 holds.
- *Derived* path formulae $F \; \varphi$ and $G \; \varphi$ are often used for simplicity. $F \; \varphi \equiv tt \; U \; \varphi$

Property Specification: Temporal Logics II

Let K be a Kripke structure and $AP(s)$ denote the set of atomic propositions associated with state s in K.

- $K, s \models p$ if $p \in AP(s)$
- $K, s \models \varphi_1 \wedge \varphi_2$ if $K, s \models \varphi_1$ and $K, s \models \varphi_2$. (sim. for "$\vee$")
- $K, s \models A\phi$ if for *every* path π in K with $init(\pi) = s$, $K, \pi \models \phi$.
- $K, s \models E\phi$ if for *some* path π in K with $init(\pi) = s$, $K, \pi \models \phi$.
- $K, \pi \models X\varphi$ if $K, \pi[1] \models \varphi$
- $K, \pi \models \varphi_1 \; U \; \varphi_2$ if $\pi = s_1, s_2, \ldots$, and $\exists i \geq 1$ such that
  - $K, s_i \models \varphi_2$, and
  - $\forall 1 \leq j < i$, $K, s_j \models \varphi_1$.
- $\ldots$

Property Specification: Temporal Logics III

Modal Mu-Calculus

φ	$\rightarrow$	$tt \mid \varphi \wedge \varphi \mid \varphi \vee \varphi$	logical connectives
	$\mid$	$\langle \alpha \rangle \varphi$	Diamond formulae
	$\mid$	$[\alpha]\varphi$	Box formulae
	$\mid$	X	Formula variable
	$\mid$	$\mu X.\varphi$	Least fixed point formula
	$\mid$	$\nu X.\varphi$	Greatest fixed point formula

Semantics of modal mu-calculus formulae are defined over LTSs, with α ranging over the actions of the LTS.

Property Specification: Temporal Logics IV

Examples in the “assembly language of propositional temporal logics”:

1. “a” action is eventually possible:

$$\mu X.(\langle a\rangle tt) \vee (\langle - \rangle X)$$

2. “b” action is eventually possible *from each state*:

$$\nu Y.[-]Y \wedge (\mu X.(\langle b\rangle tt) \vee (\langle - \rangle X))$$

3. “c” action is enabled infinitely often on all infinite paths:

$$\nu Y.\mu X.[-](((\langle c\rangle tt \wedge Y) \vee X)$$

System Specification: Process Languages

- Core languages such as CCS, pi-calculus etc. that elucidate the meaning of interleaving, synchronization, communication etc.
- Value-passing languages add variables and values to the core to allow succinct specifications.
- Meanings of specifications in such languages are given in terms of LTSs and Kripke Structures ("translating them down").
- Derived LTSs may not be finite state.
- Richer languages permit description of
    - Push-down systems (analogous to recursive programs)
    - Parameterized systems (where aspects of a systems, e.g. number of processes of a specific family, may be only symbolically specified).

A Model Checker for CTL: I

CTL formulae represented as Prolog Terms of the following forms:

- `prop`(P) atomic propositions
- `neg`(F), `and`(F_1, F_2), `or`(F_1, F_2): logical connectives.
- `ex`(F), `eu`(F_1, F_2), `er`(F_1, F_2): formulae with existential path quantifier
- `ax`(F), `au`(F_1, F_2), `ar`(F_1, F_2): formulae with universal path quantifier

Kripke structure represented as a set of Prolog facts:

- `trans`(S,T): transition from state S to state T.
- `holds`(S,P): proposition P holds at state S.

A Model Checker for CTL: II

```
% Propositions and their negations
models(S, prop(P)) :- holds(S, P).
models(S, neg(prop(P))) :- not holds(S, P).

% Conjunction and Disjunction
models(S, and(F1,F2)) :- models(S, F1), models(S, F2).
models(S, or(F1, F2)) :- models(S, F1); models(S, F2).

% EX:
models(S, ex(F)) :- trans(S, T), models(T, F).
```

A Model Checker for CTL: II

- $E\varphi_1\ U\ \varphi_2$ can be “unrolled” as:

$$\varphi_2 \vee (\varphi_1 \wedge (EX\ (E\varphi_1 U \varphi_2)))$$

  where the unrolling is finite (least fixed point).

- Hence:

```
% EU
models(S, eu(F1, F2)) :-
    models(S, or(F2, and(F1, ex(eu(F1, F2))))).
```

- Similarly:

```
% AU
models(S, au(F1, F2)) :-
    models(S, or(F2, and(F1, ax(au(F1, F2))))).
```

A Model Checker for CTL: III

- $A\phi \equiv \neg E \neg \phi$ and $\neg X \varphi \equiv X \neg \varphi$
- But encoding `ax` in terms of negation and `ex` means our unrolling of `au` will result in a non-stratified program.
- Hence:

```
% AX
models(S, ax(F)) :-
    findall(T, trans(S, T), L),
    all_models(L, F).
```

where

```
all_models([], F).
all_models([S|L], F) :- models(S, F), all_models(L, F).
```

A Model Checker for CTL: IV

- $\neg(\varphi_1 R\ \varphi_2) \equiv (\neg\varphi_1) U\ (\neg\varphi_2)$
- Hence:

```
% ER
models(S, er(F1, F2)) :-
    negate(F1, NF1),
    negate(F2, NF2),
    tnot models(S, au(NF1, NF2)).
```

where `negate`(F, NF) binds NF to the negation normal form of `neg`(F).

Tabled Evaluation for Model Checking

- Tabled resolution is needed for termination
  - Note unrolling of `eu` and `au`
- `models/2` is not statically stratified
  - Note use of negative dependency in treatment of `er`.
- But query evaluation will be dynamically stratified
  - Expansion of `er` using `au` is not unrolling, and does not lead to cycles.

Complexity of CTL Model Checking

- `models/2`: Given a Kripke structure with $|S|$ states, and a formula of size $|\varphi|$, there are at most $O(|S|.|\varphi|)$ distinct calls to `models/2`.
- Each call is *ground*, so as at most one answer.
- If $|T|$ is the size of the Kripke structure (max. of number of states and transitions), then query evaluation takes $O(|T|.|\varphi|)$ steps.
- Access into the call tables may take an additional $O(|\varphi|)$ time per access.
- But with hash-consing (or any other suitable representation of a formula term), table access time will be $O(1)$ with perfect indexing.
- Hence model checking can be done in $O(|T|.|\varphi|)$ time and $(|S|.|\varphi|)$ space.

Model Checker for the Modal Mu-Calculus: I

Equational form of the modal mu-calculus.

- $\mu X.(\langle a\rangle tt) \vee (\langle - \rangle X)$
  written as $x \overset{\mu}{=} (\langle a\rangle tt) \vee (\langle - \rangle x)$
- $\nu Y.[-]Y \wedge (\mu X.(\langle b\rangle tt) \vee (\langle - \rangle X))$
  written as a set of two equations:

$$\begin{aligned} y &\overset{\nu}{=} [-]y \wedge x \\ x &\overset{\mu}{=} (\langle b\rangle tt) \vee (\langle - \rangle x) \end{aligned}$$

- $\nu Y.\mu X.[-]((\langle c\rangle tt \wedge Y) \vee X)$
  written as a set of two parameterized equations:

$$\begin{aligned} y &\overset{\nu}{=} x(y) \\ x(Z) &\overset{\mu}{=} [-]((\langle c\rangle tt \wedge Z) \vee x(Z)) \end{aligned}$$

Model Checker for the Modal Mu-Calculus: II

- Assume equational form of mu-calculus formulas are represented by a set of facts. The RHS of equations are represented a Prolog terms.

```
% Propositions and their negations
models(S, tt).
models(S, and(F1,F2)) :- models(S, F1), models(S, F2).
models(S, or(F1, F2)) :- models(S, F1); models(S, F2).

% Diamond:
models(S,diam(A,F)) :- trans(S, A, T), models(T, F).

% Box :
models(S, box(A,F)) :-
    findall(T, trans(S, A, T), L), all_models(L, F).
```

- Note the action label “A” in the transition relation.

Model Checker for the Modal Mu-Calculus: trial

- Assume equational form of mu-calculus formulas are represented by a set of facts of the form
  - `lfp`(x, φ) for μ equations, and
  - `gfp`(x, φ) for ν equations.

```
% LFP formula
models(S, form(X)) :- lfp(X, F), models(S, F).

% GFP formula
models(S, form(X)) :-
    gfp(X, F),
    negate(F, NF),
    tnot models(S, NF).
```

Alternation Freedom and Stratification

- As in the case of CTL model checker, the `models/2` predicate defining the mu-calculus model checker is not statically stratified.
- But for formulae with a single fixed point, or alternation-free fixed points, query evaluation is dynamically stratified.
- For formulae with alternation, query evaluation may not even be dynamically stratified.
    - One strategy is to generate a “residual” program that retains the cycles through negation, and generate a preferred stable model of the residue.

Complexity of Modal Mu-Calculus Model Checking

Time and space complexity of query evaluation over `models/2` can be analyzed along the same lines as used for CTL.

- For an alternation-free formula φ, model checking can be done in
  - $O(|S|.|\varphi|)$ space, where $|S|$ is the number of states in the LTS.
  - $O(|T|.|\varphi|)$ time, where $|T|$ is the size of the LTS.

Beyond Finite-State Model Checking: I

- Use constraints to represent sets of equivalent states.
    - Finite number of equivalence classes implies termination (e.g. classical timed automata).
- Allow data variables in *Property Specification* that may unify with data fields in system specification.
    - Encoding is agnostic to which side has variables.
    - Enables verification of a class of *data independent* systems.

Beyond Finite-State Model Checking: II

- Use Forall-Exists quantified Horn Clauses as a constraint language with powerful (albeit incomplete) solvers.
- A number of model checking problems, including CTL model checking of infinite state systems, can be cast as a satisfaction problem over the above constraint language.

Logic Programs

Program Rules

$+$ $\models$ **Query Answers**

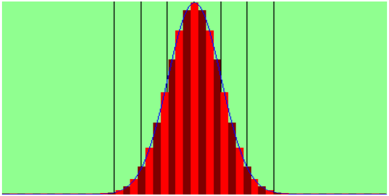
Facts

Probabilistic Logic Programs

Program Rules

$+$ $\models$ **Query Answers**

`Probabilistic Facts`



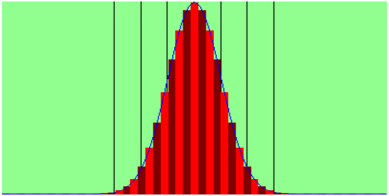
ICL, PRISM, ProbLog, . . .

Probabilistic Logic Programs

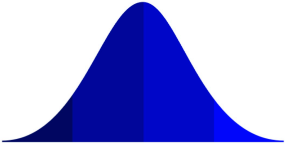
Program Rules

$+$

Probabilistic Facts



$\models$ **Query Answers**



ICL, PRISM, ProbLog, . . .

Independent Choice Logic (ICL)

Poole's ICL forms a precursor to modern PLP languages.
An ICL theory has:

- An *acyclic* logic program
- A collection of atom sets, e.g. $\{\{a_0, a_1\}, \{b_0, b_1, b_2\}\}$, called the choice space.
    - Each set in the collection can be viewed as a random variable; each atom in the set is a possible outcome of the variable.
- Distributions over the choice space.

Semantics of ICL given in terms of the distributions over the choice space.

Stochastic Logic Programs (SLP)

- SLP [Muggleton et al] defines a probability distribution over program clauses.
- Probability of a query answer is computed based on the probabilities of clauses used during resolution.
- SLP is expressive enough to represent a large class of non-recursive stochastic systems (e.g. non-recursive Stochastic Context Free Grammars).

PRISM

A language for probabilistic logic programming with system for inference and parameter learning (Sato et al, since '99).

- Logic programs with a set of **probabilistic facts**: `msw(X, I, V)`, where
    - `X` is a discrete-valued random process
    - `V` is a value generated by the random process
    - `I` is the *instance number*, distinguishing different trials.
- Random variables generated by the same random process are i.i.d.
- Random variables generated by distinct random processes are independent.
- Has a well-defined model-theoretic (*distribution*) semantics, and an operational semantics based on tabled resolution.

Distribution semantics

```
% "a" is a boolean random process
p(X) :-  msw(a, 0, X),
         msw(a, 1, Y),
         X=Y.
values(a, [t,f]).
set_sw(a, [0.3,0.7])
```

Distribution semantics

```
% “a” is a boolean random process
p(X) :-  msw(a, 0, X),
         msw(a, 1, Y),
         X=Y.
values(a, [t,f]).
set_sw(a, [0.3,0.7])
```

Worlds:

msw(a,0,t) msw(a,1,t)	msw(a,0,t) msw(a,1,f)
msw(a,0,f) msw(a,1,t)	msw(a,0,f) msw(a,1,f)

- Outcomes of random processes define worlds.

Distribution semantics

```
% “a” is a boolean random process
p(X) :- msw(a, 0, X),
        msw(a, 1, Y),
        X=Y.
values(a, [t,f]).
set_sw(a, [0.3,0.7])
```

Worlds:

msw(a,0,t) msw(a,1,t) 0.09	msw(a,0,t) msw(a,1,f) 0.21
msw(a,0,f) msw(a,1,t) 0.21	msw(a,0,f) msw(a,1,f) 0.49

- Outcomes of random processes define worlds.
- The probability of a world is assigned based on the probabilities of the outcomes in the world.

Distribution semantics

```
% “a” is a boolean random process
p(X) :- msw(a, 0, X),
        msw(a, 1, Y),
        X=Y.
values(a, [t,f]).
set_sw(a, [0.3,0.7])
```

Worlds:

msw(a,0,t) msw(a,1,t) 0.09	msw(a,0,t) msw(a,1,f) 0.21
msw(a,0,f) msw(a,1,t) 0.21	msw(a,0,f) msw(a,1,f) 0.49

- Outcomes of random processes define worlds.
- The probability of a world is assigned based on the probabilities of the outcomes in the world.
- In each world, `msw`s form a set of logical (non-probabilistic) facts.

Distribution semantics

```
% “a” is a boolean random process
p(X) :- msw(a, 0, X),
        msw(a, 1, Y),
        X=Y.
values(a, [t,f]).
set_sw(a, [0.3,0.7])
```

Models:

msw(a,0,t) msw(a,1,t) 0.09 p(t)	msw(a,0,t) msw(a,1,f) 0.21
msw(a,0,f) msw(a,1,t) 0.21	msw(a,0,f) msw(a,1,f) 0.49 p(f)

- Outcomes of random processes define worlds.
- The probability of a world is assigned based on the probabilities of the outcomes in the world.
- In each world, `msw`s form a set of logical (non-probabilistic) facts.
- Distribution over least models: the least model in each world is assigned the probability of that world.

ProbLog

- At its simplest, a ProbLog program resembles a Prolog program where each clauses is annotated with a discrete probability value.
- These annotations define a distribution of (non-probabilistic) programs, resulting in a distribution semantics.
- ProbLog and PRISM program, when restricted to discrete distributions, can be translated to one another, with the same distribution semantics.
- ProbLog query evaluation materializes explanations in non-trivial structures, and is not subject to PRISM-style independence and mutual exclusion restrictions.

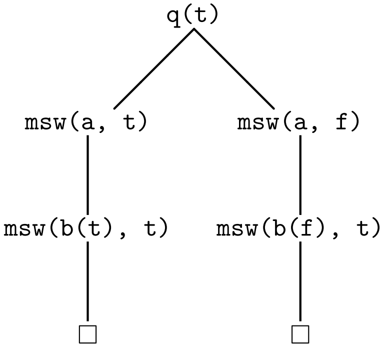
Probabilistic Logic Programs: A quick tour

- Logic-based representation of statistical models
    - Examples include BLPs (Kersting and De Raedt, '00), PRMs (Friedman et al, '99), MLNs (Richarson and Domingos, '06).
    - The underlying statistical network, derived from logical/statistical specifications, is finite.
- Statistical inference over proof structures
    - Conservative extension to traditional logic programs, with explicit or implicit use of random variables and processes.
    - Examples include PRISM (Sato and Kameya, '99), ICL (Poole, '93), CLP(BN) (Santos Costa et al, '03), ProbLog (De Raedt et al, '07), LPAD (Vennekens et al, '09).
    - In terms of expressive power, PRISM, ProbLog and LPAD coincide; however, they use different inference procedures.

Evaluation in PRISM — I

```
% Finite Mixture Model
q(Y) :-  msw(a, 0, X),
          msw(b(X), 0, Y).

values(a, [t,f]).
values(b(_), [t,f]).
set_sw(a, [0.3,0.7])
set_sw(b(t), [0.6,0.4])
set_sw(b(f), [0.5,0.5])
```

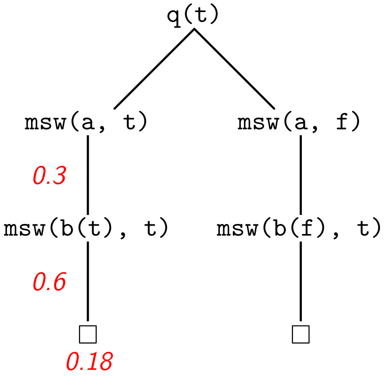
Explanations



Evaluation in PRISM — I

```
% Finite Mixture Model
q(Y) :-  msw(a, 0, X),
          msw(b(X), 0, Y).

values(a, [t,f]).
values(b(_), [t,f]).
set_sw(a, [0.3,0.7])
set_sw(b(t), [0.6,0.4])
set_sw(b(f), [0.5,0.5])
```

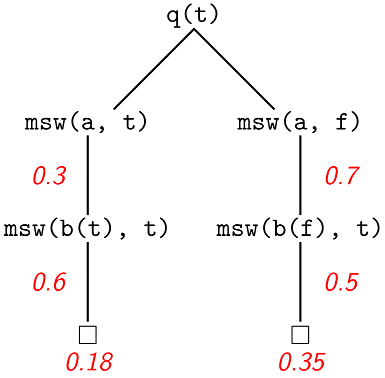
Explanations *and Probabilities*



Evaluation in PRISM — I

```
% Finite Mixture Model
q(Y) :-  msw(a, 0, X),
          msw(b(X), 0, Y).

values(a, [t,f]).
values(b(_), [t,f]).
set_sw(a, [0.3,0.7])
set_sw(b(t), [0.6,0.4])
set_sw(b(f), [0.5,0.5])
```

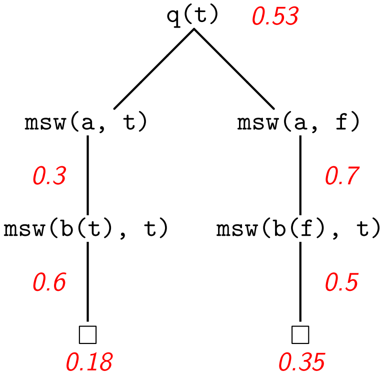
Explanations *and Probabilities*



Evaluation in PRISM — I

```
% Finite Mixture Model
q(Y) :-  msw(a, 0, X),
          msw(b(X), 0, Y).

values(a, [t,f]).
values(b(_), [t,f]).
set_sw(a, [0.3,0.7])
set_sw(b(t), [0.6,0.4])
set_sw(b(f), [0.5,0.5])
```

Explanations *and Probabilities*



Evaluation in PRISM — II

- *Explanation* of an answer: At a high level, the set of `msw`'s used in a derivation of the answer.

Evaluation in PRISM — II

- *Explanation* of an answer: At a high level, the set of `msw`'s used in a derivation of the answer.
- The probability of an explanation is the product of the probabilities of random variables in the explanation.

Evaluation in PRISM — II

- *Explanation* of an answer: At a high level, the set of `msw`'s used in a derivation of the answer.
- The probability of an explanation is the product of the probabilities of random variables in the explanation.
  - If the `msw`'s in a derivation are all independent, then the probability of the explanation can be computed without materializing it.

    [Independence assumption]

Evaluation in PRISM — II

- *Explanation* of an answer: At a high level, the set of `msw`'s used in a derivation of the answer.
- The probability of an explanation is the product of the probabilities of random variables in the explanation.
  - If the `msw`'s in a derivation are all independent, then the probability of the explanation can be computed without materializing it. [Independence assumption]
- The probability of an answer is the probability of the set of explanations of the answer.

Evaluation in PRISM — II

- *Explanation* of an answer: At a high level, the set of `msw`'s used in a derivation of the answer.
- The probability of an explanation is the product of the probabilities of random variables in the explanation.
    - If the `msw`'s in a derivation are all independent, then the probability of the explanation can be computed without materializing it. [Independence assumption]
- The probability of an answer is the probability of the set of explanations of the answer.
    - If explanations are pairwise mutually exclusive, then the probability of the set of explanations is the sum of probabilities of each explanation. [Mutual Exclusion assumption]

Evaluation in PRISM — II

- *Explanation* of an answer: At a high level, the set of `msw`'s used in a derivation of the answer.
- The probability of an explanation is the product of the probabilities of random variables in the explanation.
  - If the `msw`'s in a derivation are all independent, then the probability of the explanation can be computed without materializing it. [Independence assumption]
- The probability of an answer is the probability of the set of explanations of the answer.
  - If explanations are pairwise mutually exclusive, then the probability of the set of explanations is the sum of probabilities of each explanation. [Mutual Exclusion assumption]
  - If the set of explanations is finite, then this sum can be effectively computed. [Finiteness assumption]

Generalizations

- PRISM's inference procedure uses the Independence, Mutual Exclusion and Finiteness assumptions to compute probabilities of answers without materializing the explanations.

Generalizations

- PRISM's inference procedure uses the Independence, Mutual Exclusion and Finiteness assumptions to compute probabilities of answers without materializing the explanations.
    - Inference mimics the best known algorithms for certain statistical models (e.g. Viterbi alg. for HMMs).

Generalizations

- PRISM's inference procedure uses the Independence, Mutual Exclusion and Finiteness assumptions to compute probabilities of answers without materializing the explanations.
  - Inference mimics the best known algorithms for certain statistical models (e.g. Viterbi alg. for HMMs).
- ProbLog and PITA (an implementation of LPAD) use BDDs to represent the set of explanations, and consequently remove Independence and Mutual Exclusion assumptions.

Generalizations

- PRISM's inference procedure uses the Independence, Mutual Exclusion and Finiteness assumptions to compute probabilities of answers without materializing the explanations.
    - Inference mimics the best known algorithms for certain statistical models (e.g. Viterbi alg. for HMMs).
- ProbLog and PITA (an implementation of LPAD) use BDDs to represent the set of explanations, and consequently remove Independence and Mutual Exclusion assumptions.
    - Finiteness assumption is still needed since the BDDs need to be effectively constructed.

Evaluation via Knowledge Compilation: I

- Explanations can be viewed as residues of partially evaluating all but the probabilistic facts.
- Each set of explanations can be mapped to a Boolean propositional formula.
  - Each explanation is a set (conjunction) of random variable valuations.
  - Each explanation in an explanation set (i.e. disjunction) supports the derived answer.
- Explanations can be materialized more succinctly using other Boolean formula representations, such as Deterministic Disjunctive Negation Normal Forms (dDNNFs), etc.

Evaluation via Knowledge Compilation: II

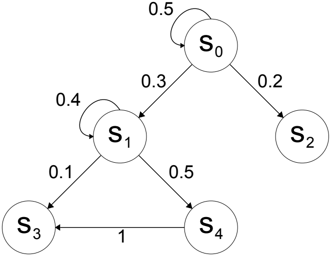
- ProbLog2 uses the more succinct representation of explanations.
- *Weighted model count* is a measure to each Boolean formula, defined a weighted sum of (the number of) satisfiable assignments.
- Weighted model counting can be done in time polynomial in the size of a Boolean formula's dDNNF representation.

Finiteness Assumption and PRISM

- In ICLP'12, Sato and Meyer present a method to generate equations from probabilistic programs with loops.
- This mechanism essentially hides the "instance" variable in `msws`.
- It proceeds under the assumption that different occurrences along a single explanation are independent.
    - This assumption holds for individual runs of a Markov Chain or prefix probability computations in Probabilistic CFGs, but does not satisfied in general.

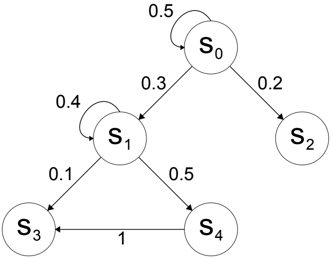
Probabilistic Transition Systems in PRISM

Example Markov Chain



Probabilistic Transition Systems in PRISM

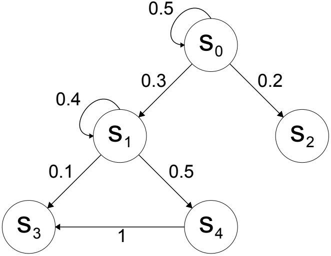
Example Markov Chain



```
% Encoding as a Probabilistic LP
trans(S, I, T) :- msw(t(S), I, T).

% Ranges
:- values(t(s0), [s0, s1, s2]).
:- values(t(s1), [s1, s3, s4]).
:- values(t(s4), [s3]).

% Distributions
set_sw(t(s0), [0.5, 0.3, 0.2]).
set_sw(t(s1), [0.4, 0.1, 0.5]).
set_sw(t(s4), [1]).
```

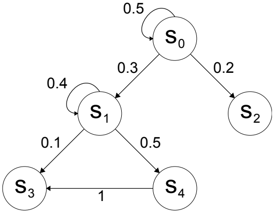
Probabilistic Transition Systems in PRISM

Example Markov Chain



```
% Encoding as a Probabilistic LP
trans(S, I, T) :- msw(t(S), I, T).

% Encoding of Reachability
reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

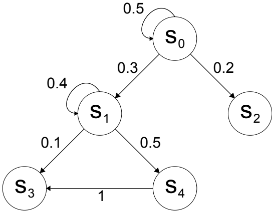
Probabilistic Model Checking as Query Evaluation



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

- What is the probability of reaching s_3 via some path starting at s_0?

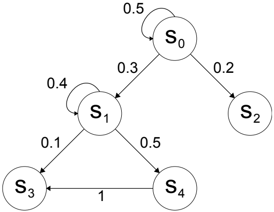
Probabilistic Model Checking as Query Evaluation



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

- What is the probability of reaching s_3 via some path starting at s_0?
- `|?- prob(reach(`s_0`, 0, `s_3`)).`

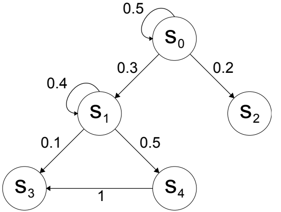
Probabilistic Model Checking as Query Evaluation



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

- What is the probability of reaching s_3 via some path starting at s_0?
- `|?- prob(reach(`s_0`, 0, `s_3`)).`
- Evaluation of the above query will not terminate!

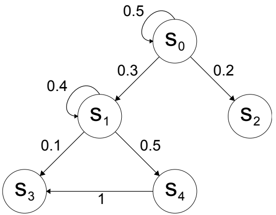
Probabilistic Model Checking as Query Evaluation



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

- What is the probability of reaching s_3 via some path starting at s_0?
- `|?- prob(reach(`s_0`, 0, `s_3`)).`
- Evaluation of the above query will not terminate!
    - There are infinitely many *explanations* for `reach(`s_0`, 0, `s_3`)`

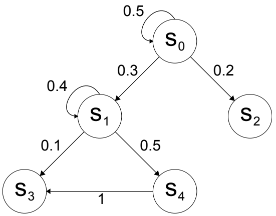
Probabilistic Model Checking as Query Evaluation



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

- What is the probability of reaching s_3 via some path starting at s_0?
- `|?- prob(reach(`s_0`, 0, `s_3`)).`
- Evaluation of the above query will not terminate!
  - There are infinitely many *explanations* for `reach(`s_0`, 0, `s_3`)`
- Distribution semantics is well-defined and gives the correct probability, but standard inference methods cannot evaluate this query.

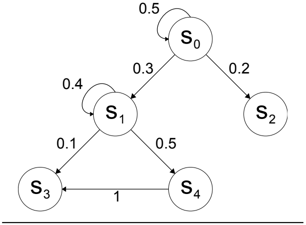
Probabilistic Model Checking as Query Evaluation



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

- What is the probability of reaching s_3 via some path starting at s_0?
- `|?- prob(reach(`s_0`, 0, `s_3`)).`
- Evaluation of the above query will not terminate!
    - There are infinitely many *explanations* for `reach(`s_0`, 0, `s_3`)`
- Distribution semantics is well-defined and gives the correct probability, but standard inference methods cannot evaluate this query.
- More recent extension in PRISM removes finiteness assumption under restricted conditions.

Explanations



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

Explanations for `reach(s0,0,s3)`:

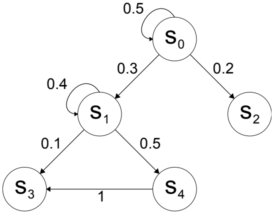
- `msw`(t(s0), 0, s1), `msw`(t(s1), *next*(0), s3).
- `msw`(t(s0), 0, s0), `msw`(t(s0), *next*(0), s1), `msw`(t(s1), *next*(*next*(0)), s3).

⋮

- `msw`(t(s0), 0, s1), `msw`(t(s1), *next*(0), s1), `msw`(t(s1), *next*(*next*(0)), s3).

⋮

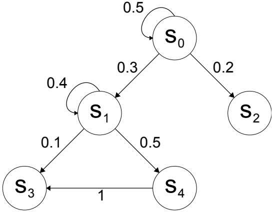
Explanations



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

Note: `prob(reach(s0,0,s3))` is same as `prob(reach(s0,H,s3))` for any `H`.

Explanations

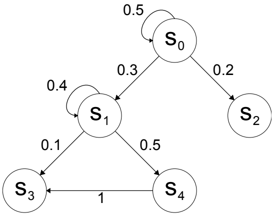


```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

Note: `prob(reach(s0,0,s3))` is same as `prob(reach(s0,H,s3))` for any H.

We can use a *grammar* to represent the set of explanations for the abstracted query.

Explanations



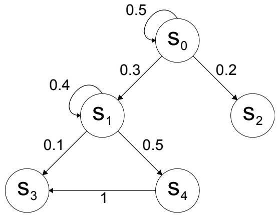
```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

Note: `prob(reach(s0,0,s3))` is same as `prob(reach(s0,H,s3))` for any `H`.

We can use a *grammar* to represent the set of explanations for the abstracted query.

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

Explanations



```
trans(S, I, T) :-
    msw(t(S), I, T).

reach(S, I, T) :-
    trans(S, I, U),
    reach(U, next(I), T).
reach(S, _, S).
```

$$
\begin{aligned}
&\mathtt{expl}(\mathtt{reach}(s0, H, s3)) \longrightarrow \\
&\quad [\mathtt{msw}(t(s0), H, s0)], \\
&\quad \mathtt{expl}(\mathtt{reach}(s0, next(H), s3)). \\
&\mathtt{expl}(\mathtt{reach}(s0, H, s3)) \longrightarrow \\
&\quad [\mathtt{msw}(t(s0), H, s1)], \\
&\quad \mathtt{expl}(\mathtt{reach}(s1, next(H), s3)).
\end{aligned}
$$

is similar to the stochastic grammar:

$$S_0 \xrightarrow{0.5} S_0$$

$$S_0 \xrightarrow{0.3} S_1$$

Answer probability of `reach(s0,H,s3)` is the *language probability* of the above SCFG

and is the least solution to equations of the form:

$x_0 = 0.5x_0 + 0.3x_1$

Abstraction for temporal programs

- *Instance* and *Non-Instance* variables belong to distinct sorts. Variables of one sort cannot be unified with those of the other.
- Only terms containing instance variables can be used as instance arguments of `msw`.
- For any clause of a predicate with an instance argument, the instances on the LHS of the clause must be a subterm of instances on the RHS. This imposes an ordering on time.

Temporally Well-Formed Programs

- A probabilistic logic program with annotations of the form `temporal`$(p/n-i)$.
  - Example: `temporal(reach/3-2)`
  - `reach` is a *temporal* predicate
  - The second argument of an atom with root `reach` is its *instance argument*.
- For a rule defining a temporal predicate, the instance argument of the head must be a subterm of instance arguments of every temporal body predicate.
  - Example:

```
reach(S, I, T) :-
        trans(S, I, U),
        reach(U, next(I), T).
```

- Instance arguments are not bound to non-instance arguments, or vice versa.

Temporally Well-Formed Programs

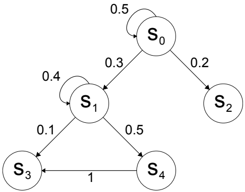
- A probabilistic logic program with annotations of the form `temporal`$(p/n - i)$.
    - Example: `temporal(reach/3-2)`
    - `reach` is a *temporal* predicate
    - The second argument of an atom with root `reach` is its *instance argument*.
- For a rule defining a temporal predicate, the instance argument of the head must be a subterm of instance arguments of every temporal body predicate.
    - Example:

```
reach(S, I, T) :-
        trans(S, I, U),
        reach(U, next(I), T).
```

- Instance arguments are not bound to non-instance arguments, or vice versa.
- In explanation grammars of temporally well-formed programs, `msw`(r, t, x) will always be independent of any `msw` derived from non-terminal `expl(p)`
    - if t is a proper subterm of p's instance argument.

Factored Equation Diagrams

Not all explanation grammars can be translated directly to stochastic grammars.



- Consider the query

  `reach(s0, H, s3); reach(s0, H, s4).`

- The grammar will have productions of the form:

$$\mathtt{expl}(\mathtt{reach}(s0, H, s3); \mathtt{reach}(s0, H, s4)) \longrightarrow \mathtt{expl}(\mathtt{reach}(s0, H, s3)).$$
$$\mathtt{expl}(\mathtt{reach}(s0, H, s3); \mathtt{reach}(s0, H, s4)) \longrightarrow \mathtt{expl}(\mathtt{reach}(s0, H, s4)).$$

- The two productions are not mutually exclusive.

We can *factor* such grammars using Factored Explanation Diagrams (FEDs), which are similar to BDDs.

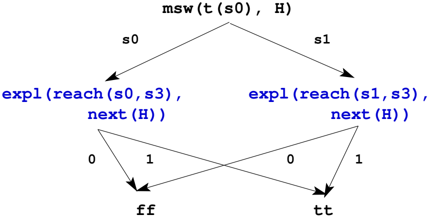
Structure of FEDs

FED is a labeled DAG with

- `tt` and `ff` as leaf nodes
- `msw`(r, h) is an n-ary node if r is a random process with n possible outcomes;

  outgoing edges are labeled with the outcomes.
- `expl`(t, h) is a binary node;

  outgoing edges are labeled 0 and 1.
- If there is an edge from x_1 to x_2, then $x_1 < x_2$ via a specially defined partial order relation.



Operations on FEDs

Boolean operations "$\wedge$" and "$\vee$" can be performed on FEDs along the same line as on BDDs, with one significant change:

- BDD operations are based on a *total* node order.
- We only have a partial node order for FEDs.
- When we recursively push operations down the diagram, we may encounter incomparable nodes.
- We then generate a placeholder `merge` node, and process merges separately.

Operations on FEDs

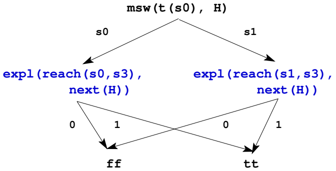
Boolean operations "$\wedge$" and "$\vee$" can be performed on FEDs along the same line as on BDDs, with one significant change:

- BDD operations are based on a *total* node order.
- We only have a partial node order for FEDs.
- When we recursively push operations down the diagram, we may encounter incomparable nodes.
- We then generate a placeholder `merge` node, and process merges separately.
- Note that `msw` nodes are always comparable; so a `merge` will involve at least one `expl` node.
- We expand (one of) the `expl` node(s) with its definition, and perform the postponed operation.

FEDs to Equations

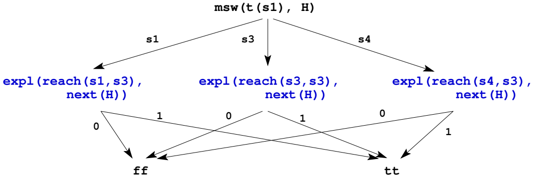
The probability of a set of explanations is computed by generating and solving a set of equations from its FED.

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



$$
\begin{aligned}
x_0 &= t_{00} * x_0 \\
&\quad + t_{01} * x_1 \\
t_{00} &= 0.5 \\
t_{01} &= 0.3
\end{aligned}
$$

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

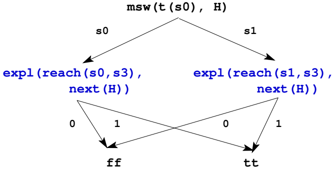


$$
\begin{aligned}
x_1 &= t_{11} * x_1 \\
&\quad + t_{13} * x_3 \\
&\quad + t_{14} * x_4 \\
t_{11} &= 0.4 \\
t_{13} &= 0.1 \\
t_{14} &= 0.5
\end{aligned}
$$

FEDs to Equations

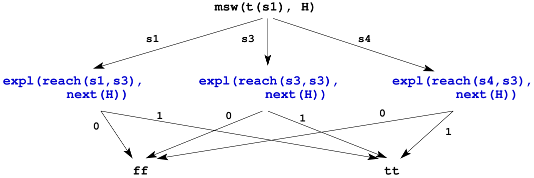
The probability of a set of explanations is computed by generating and solving a set of equations from its FED.

FED for expl(reach(s0,s3), H):



$$
\begin{aligned}
x_0 &= t_{00} * x_0 \\
&\quad + t_{01} * x_1 \\
t_{00} &= 0.5 \\
t_{01} &= 0.3
\end{aligned}
$$

FED for expl(reach(s1,s3), H):



$$
\begin{aligned}
x_1 &= t_{11} * x_1 \\
&\quad + t_{13} * x_3 \\
&\quad + t_{14} * x_4 \\
t_{11} &= 0.4 \\
t_{13} &= 0.1 \\
t_{14} &= 0.5
\end{aligned}
$$

The least solution to these monotone polynomial equations gives the probability of the set of explanations.

Probabilistic Inference for Temporal Queries: Summary

- The set of explanations for a temporal query is conceptually treated as a language defined by a probabilistic grammar.
- This grammar is transformed and materialized as a Factored Explanation Diagram (FED) which ensures that
  - Distinct productions (paths in the diagram) are mutually exclusive.
  - Trials of random variables in a path are independent
- In other words, an FED is a stochastic grammar for he language of explanations.
- Answer probability is computed as the language probability of the grammar: by solving a system of *monotone* polynomial equations.