Verification of Infinite-State Systems

An Introduction to Research in the LMC Lab

CSE 653: Seminar in Concurrency; Fall 2002

Formal Verification

- **Modeling:**
  - Systems: (hardware, control algorithms, software components, ...)
    Modeled as different kinds of automata (FSA, PDA, EFSA, ...).
  - Properties:
    * Specified as a formula in temporal logic (e.g., every `send` is eventually followed by a `receive`)
    * Specified as another (more abstract) automaton
- **Verification:**
  - Model vs. Temporal formula: ***Model Checking***
  - Model vs. Model: ***Equivalence Checking***

System Models

- Verification techniques typically treat systems as automata (basic underlying representation)
- End users may specify models as programs (expressions) in formal languages (e.g., Process Algebras)
  These formal notations are translated into automata for verification.
  - Formal specification: $p \stackrel{def}{=} \overline{a}.b.p$
  - Automaton: $p \xrightarrow{\overline{a}} p' \xrightarrow{b} p$

Value-Passing Models

- Processes and communications take *parameters*.
- ```
  recv(Current) ::=
      in?Seq.
      (    Seq == Current . out!Seq. recv(1-Current)
        +  Seq != Current . recv(Current) )
  ```
- Specifications look more like a program.
- Control states correspond to “program counter values”
- Parameters are variables that take values over some domain.
  ```

Infinite-State Systems

- Systems with infinite (or unbounded) data domains

```
medium ::= in?X . out!X . medium
```

- Systems with unbounded number of concurrent components

```
chain(1) ::= cell
chain(N) ::= cell | chain(N-1)
```

- Push-down systems (unbounded stack)

```
up ::= in?X . up. out!X  +  out!middle
```

- Systems with infinite name spaces (mobile processes, security protocols)

```
sender(RecvKey, MyID) ::=
    nu(Nonce, out!enc(RecvKey, msg(Nonce, MyID))...)
```


Verification by Query Evaluation

Basic idea behind most of LMC research.

Model Checking: Given a system description with start state s_0 and a property φ

$$s_0 \stackrel{?}{\models} \varphi$$

Query Evaluation: Encode the "$\models$" relation as predicate `models` in a logic program.

$s_0 \stackrel{?}{\models} \varphi$ is determined by solution to the query `models(`s_0, φ`)`

A Simple Example

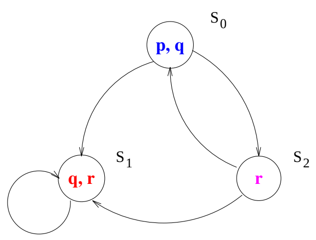
Verifying Reachability Properties

- Encode system model using "EDB facts"
- Encode reachability relation using Horn Clauses
- Issue appropriate query

A Simple Example – II

Encoding FSA using EDB facts

Structure



Encoding

```
edge(s0, s1).
edge(s0, s2).
edge(s1, s1).
edge(s2, s0).
edge(s2, s1).

prop(s0, p).
prop(s0, q).
prop(s1, q).
prop(s1, r).
prop(s2, r).
```

A Simple Example — III

Reachability relation:

```
reach(X,Y) :- edge(X,Y).
reach(X,Y) :- reach(X,Z), edge(Z,Y).
```

Query: e.g., "Is a state where 'r' is true reachable from state s_0?

```
?- reach(s0, S), prop(S, r).
```

Answers: `S=s1`, `S=s2`

Representing EFSA

- `medium ::= in?X . out!X . medium`

```
edge(medium, read(in,X), medium1(X)).
edge(medium1(X), write(out,X), medium(X)).
```

```
recv(Current) ::=
   in?Seq.
   (    Seq == Current . out!Seq. recv(1-Current)
     +  Seq != Current . recv(Current) )
```

```
edge(recv(C), read(in, Seq), recv1(C,Seq)).
edge(recv1(C,Seq), write(out, Seq), recv(NewSeq)) :-
    Seq == C, NewSeq is 1-C.
edge(recv1(C,Seq), Action, Dest) :-
    Seq \= Current,
    edge(recv(C), Action, Dest).
```


Modal Mu-calculus

```
Fexp -->  Fname
       |  tt
       |  Fexp /\ Fexp
       |  Fexp \/ Fexp
       |  diam(A, Fexp)
       |  diamMinus(A, Fexp)
       |  box(A, Fexp)
       |  boxMinus(A, Fexp)

Definition -->
          Fname -= Fexp
       |  Fname += Fexp
```

An Example: *deadlock freedom*

```
df -= boxMinus(nil,  df) /\ diamMinus(nil, tt)
```


Model Checking with Modal Mu-Calculus

```
models(S, tt).

models(S, F1 \/ F2)) :- models(S, F1) ; models(S, F2).

models(S, F1 /\ F2)) :- models(S, F1),  models(S, F2).

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

models(S, diamMinus(A, F)) :-
    trans(S, B, T),  A \= B, models(T, F).

models(S, box(A, F)) :-
    forall(T, trans(S, A, T), models(T, F)).

models(S, boxMinus(A, F)) :-
    forall((B,T), (trans(S, B, T), A \= B), models(T, F)).
```

Implementing `forall`

Actual encoding of `box` formulas makes free variables explicit:

```
models(S, box(A, F)) :-
   forall(T,
      (S,A,F)^trans(S, A, T),
      models(T, F)).

forall(Bv, Fv^Ant, Cons) :-
   findall((Fv,Cons), Ant, L),
   all_true(Fv, L).
all_true(_, []).
all_true(Fv, [(Fv, Cons)|Rest]) :-
   Cons,
   all_true(Fv, Rest).
```


Fixed Points

Minimal model of the logic program ≡ least fixed point.

```
models(S, Fname) :-
    Fname += Fexp,
    models(S, Fexp).
```

Greatest fixed points can be computed using the identity

$$\nu X.f(X) \equiv \neg\mu X.\neg f(\neg X)$$

```
models(S, Fname) :-
    Fname -= Fexp,
    negate(Fexp, NFexp),
    not models(S, NFexp).
```

where `negate(F, NF)` is such that `NF` ≡ ¬`F` and `NF` itself doesn't contain '¬'.

Nested Fixed Points

- The well-founded model coincides with (the) 2-valued stable model for (dynamically) stratified programs

  ⟹ implementation is complete for alternation-free fragment of modal mu-calculus
- Alternation in formula leads to non-stratified programs.
  - Results in *signed* programs with stable models. The structure of alternation dictates a preference order among the stable models.
  - Stable models can be computed from the residual program.

Value-Passing Modal Mu-calculus

```
order -= [a(X)] follow(X) /\ [-] order.
follow(X) += <b(X)>tt \/ [-b(X)]follow(X).
```

- Variables are quantified by modalities

  variables in `<>` are existential; variables in `[]` are universal.
- For finite-state systems, the encoding of the model checker (as it stands) verifies value-passing formulae.
- For handling infinite-state systems we need constraint processing...

Model Checking Infinite-State Systems

A (trivial?) example:

- System Model:

```
edge(medium, read(in,X), medium1(X)).
edge(medium1(X), write(out,X), medium(X)).
```

- Formula:

```
df -= boxMinus(nil,  df) /\ diamMinus(nil, tt)
```

- The system contains infinite-domain variables, but verification does not involve values over such domains
- When verification does need values over such domains, can we restrict attention to certain small number of elements?
- *Come to Beata's talk to hear more about this.*

Model Checking the Pi Calculus

- Pi calculus is a process algebra where data and channels (ports) come from the same set of *names*.
- Process may create names with explicit scopes.

```
sender(RecvKey, MyID, Out) ::=
    nu(Nonce, Out!enc(RecvKey, msg(Nonce, MyID))...)
```

- . . . but may send private names to other processes too [scope extrusion]
- Names range over an infinite domain, and are dynamically created.
- *Come to Ping's talk to hear how we can verify Pi-calculus specifications*

Model Checking Push-Down Systems

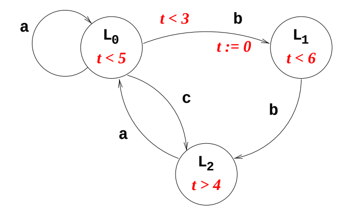
- Due to unbounded (call) stack, push down systems have infinite state spaces.
- For verifying LTL properties of PDSs, we can

  Construct an abstract FSA from a given PDS such that

  the FSA has the property *iff* the PDS has the property.
- Mu-calculus model checking of PDSs is also decidable.
- *Come to Samik's talk to know more about this.*

Real-time Systems

Timed Automaton



Encoding

```
trans(l0(T), a, l0(T)).
trans(l0(T), b, l1(0)) :- T < 3.
trans(l0(T), c, l2(T)).
trans(l1(T), b, l2(T)).
trans(l2(T), a, l0(T)).

trans(l0(T0), eps(D), l0(T1)) :-
    T1 = T0+D, T1<5.
trans(l1(T0), eps(D), l1(T1)) :-
    T1 = T0+D, T1<6.
trans(l2(T0), eps(D), l2(T1)) :-
    T1 = T0+D, T1>4.

inv(l0(T)) :- T < 5.
inv(l1(T)) :- T < 6.
inv(l2(T)) :- T > 4.
```

Verifying Reachability Properties

```
reach(X,Y) :- trans(X,_,Y), inv(Y).
reach(X,Y) :- reach(X,Z), trans(Z,_,Y), inv(Y).
```

- `X` and `Y` correspond to location/zone pairs.
- Terminates when evaluated using a Constraint LP system with tabling.
  - Needs entailment check when searching through tables.
- Encoding is suited for forward reachability (note the use of location invariants).

  Formulation of backward reachability is similar & straightforward.

Verifying Timed Mu-Calculus Properties

Conservative extension to mu-calculus model checker

1. Universal modality for untimed calculus was encoded operationally, using `forall`.
2. Universal time modality $[\epsilon]$ cannot be encoded in the same way due to quantification over an infinite domain.
3. Elimination of universally quantified interval varible (`D` in above formulation) can be programmed as a basic operation.
4. Needs a ternary `models` relation `models(S, F, SubS)`, where `SubS` is a collection location/zone pairs such that
   - $[\![\texttt{SubS}]\!] \subseteq [\![\texttt{S}]\!]$
   - $\forall \pi \in [\![\texttt{SubS}]\!] \quad \pi \models \texttt{F}$
   - `SubS` is the largest such collection

Implementing Real-Time Model Checkers

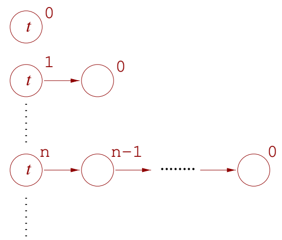
No mature LP system that combines constraint processing with tabulation

- Preliminary implementation of hooks for constraint libraries in XSB. [Cui & Warren '00]
- Tabulation implemeted by meta-programming in SICStus Prolog [Delzanno & Podelski '99; Mukhopadhyay & P. '00]
- Interface for polyhedra packages with XSB [RTSS'00]
- Efficient implementation using Difference Bound Matrices [ICLP'02]

  Come to Giri's talk to know more about this!

Parameterized Systems

Infinite family of finite-state systems.




```
trans(c(s(M)), c(M)).

base(c(0), p).

ef(S, P) :- base(S, P).
ef(S, P) :-
   trans(S, T), ef(T, P).
```

Model Checking Parameterized Systems

Consider:

```
trans(c(s(M)), c(M)).
base(c(0), p).
ef(S, P) :- base(S, P).
ef(S, P) :- trans(S, T), ef(T, P).
```

- Query "`?- ef(c(`k`), p)`" terminates for any finite k.
- Query "`?- ef(c(N), p)`" enumerates all solutions:

```
N = 0;
N = s(0);
N = s(s(0));
```


A Human Proof

Define:

```
nat(0).
nat(s(N)) :- nat(N).
```

Theorem: $\forall$ `N nat(N)` $\Rightarrow$ `ef(c(N), p)`

Proof: By induction on `nat`.

Base case (`N = 0`)**:** `ef(c(0), p)` is true since `base(c(0), p)` is true.

Induction hypothesis: $\forall$ `K` $\leq$ `M nat(K)` $\Rightarrow$ `ef(c(K), p)`

Induction step (`N = s(M)`)**:** `ef(c(s(M)), p)` is true because `ef(c(M), p)` is true (by induction hypothesis) and there exists a transition from state `c(s(M))` to state `c(M)` □

Can we extend query evaluation to automate this proof ?

Proof by Program Transformations

Define in program P_0 `thm(N) :- nat(N), ef(c(N),p).`

Convert the proof obligation to a predicate equivalence `thm` $\stackrel{?}{\equiv}$ `nat` in P_0.

Transform definition of `thm` in P_0 to the following definition in a program P_k:

```
thm(0).
thm(s(N)) :- thm(N).
```

The definition of `thm` in program P_k is syntactically equivalent to that of `nat`; hence `thm` $\equiv$ `nat`

Program Transformation : Unfolding

$$
\begin{array}{l}
\mathtt{q} : -G, \boxed{\mathtt{p}}, G'. \\
\\
\mathtt{p} : -Bd_1. \\
\mathtt{p} : -Bd_2. \\
\vdots \\
\mathtt{p} : -Bd_n.
\end{array}
\quad \xrightarrow{\textit{Unfold}} \quad
\begin{array}{l}
\mathtt{q} : -G, \; Bd_1, \; G'. \\
\mathtt{q} : -G, \; Bd_2, \; G'. \\
\vdots \\
\mathtt{q} : -G, \; Bd_n, \; G'. \\
\mathtt{p} : -Bd_1. \\
\vdots \\
\mathtt{p} : -Bd_n.
\end{array}
$$

- Each step in query evaluation is an application of unfolding.
- Corresponds to the base case as well as finitely evaluated portions of the induction step in an induction proof.

Program Transformation : Folding

$$P_j\ (j \le i): \quad \begin{array}{l} \mathtt{p} :- Bd_1. \\ \vdots \\ \mathtt{p} :- Bd_n. \end{array}$$

$$P_i: \quad \begin{array}{l} \mathtt{q} :- G, \ Bd_1, \ G'. \\ \mathtt{q} :- G, \ Bd_2, \ G'. \\ \vdots \\ \mathtt{q} :- G, \ Bd_n, \ G'. \end{array} \quad \xrightarrow{Fold} \quad \mathtt{q} :- G, \boxed{\mathtt{p}}, G'.$$

- Replaces occurrence of clause body (from a previous program in the transformation sequence) by its head.
- Corresponds to recognition of induction hypothesis.

Verification by Transformations: An Example

P_0 :

```
thm(N) :- nat(N), ef(c(N),p).
nat(0).
nat(s(N)) :- nat(N).
ef(S,P) :- base(S,P).
ef(S,P) :- trans(S,T), ef(T,P).
base(c(0),p).
trans(c(s(N)), c(N)).
```

↓ *Unfold* (Discovering induction schema)

P_1 :

```
thm(0) :- ef(c(0),p).
thm(s(N)) :- nat(N), ef(c(s(N)), p).
```


Transformation Example (contd.)

P_1 :

```
thm(0) :- ef(c(0), p).
thm(s(N)) :- nat(N), ef(c(s(N)), p).
```

↓ *Unfolds* (Base case completed)

P_4 :

```
thm(0).
thm(s(N)) :- nat(N), ef(c(s(N)), p).
```

↓ *Unfolds* (Finite part of induction step)

P_7 :

```
thm(0).
thm(s(N)) :- nat(N), ef(c(N), p).
```

↓ *Fold* (Applying induction hypothesis)

P_8 :

```
thm(0).
thm(s(N)) :- thm(N).
```


Verification by Program Transformation

- Can do nested inductions, using goal replacement transformations.
- Strategies to control the order of transformations:
  - Apply model checking (unfolding) steps as much as possible.
  - Allow interleaving of model checking and deductive (folding, replacement) steps.
- [TACAS'00] Implemented our control strategies to produce proofs for:
  - Liveness in chains (previous example)
  - Mutual exclusion in token rings,
  - Liveness in a family of binary trees, etc.

Bisimulation for basic LTS

Given a labeled transition system $L = (S, T)$, $\mathcal{R}$ is a bisimulation relation if $\mathcal{R}$ is the largest relation such that

$$\begin{array}{lll} \forall s_1, s_2 \in S \quad s_1 \, \mathcal{R} \, s_2 & \Rightarrow & \forall (s_1, a, t_1) \in T \quad \exists (s_2, a, t_2) \in T \quad t_1 \, \mathcal{R} \, t_2 \\ & \wedge & \forall (s_2, a', t_2') \in T \quad \exists (s_1, a', t_1') \in T \quad t_1' \, \mathcal{R} \, t_2' \end{array}$$

Consider $\overline{\mathcal{R}}$, the complement of the bisimulation relation:

$$\begin{array}{lll} \forall s_1, s_2 \in S \quad s_1 \, \overline{\mathcal{R}} \, s_2 & \Leftarrow & \exists (s_1, a, t_1) \in T \quad \forall (s_2, a, t_2) \in T \quad t_1 \, \overline{\mathcal{R}} \, t_2 \\ & \vee & \exists (s_2, a', t_2') \in T \quad \forall (s_1, a', t_1') \in T \quad t_1' \, \overline{\mathcal{R}} \, t_2' \end{array}$$

$\overline{\mathcal{R}}$ is the smallest such relation (i.e., least model for the logical formula above).

Implementing Bisimulation Checking

```
:- table nbisim/2.
nbisim(S1, S2) :-
   trans(S1, A, T1),
   no_matching_trans(S2, A, T1).
nbisim(S1, S2) :- nbisim(S2, S1).
no_matching_trans(S2, A, T1) :-
   forall(T2, trans(S2,A,T2), nbisim(T1, T2)).
bisim(S1, S2) :- tnot(nbisim(S1,S2)).
```

Performs *LOCAL* bisimulation checking

Time Complexity: $O(|S| \times |T|)$ assuming unit-time table lookups.

(Tables as binary trees introduces $O(log|S|)$ factor.)

Implementing `forall`

In practice, encoding of `forall` makes free variables explicit.

E.g.,

```
no_matching_trans(S2, A, T1) :-
   forall(T2,
      (S2,A,T1)^trans(S2, A, T2),
      nbisim(T1, T2)).
forall(Bv, Fv^Ant, Cons) :-
   findall((Fv,Cons), Ant, L),
   all_true(Fv, L).
all_true(_, []).
all_true(Fv, [(Fv, Cons)|Rest]) :-
   Cons,
   all_true(Fv, Rest).
```


Bisimulation for Symbolic LTS

Given an extended LTS $L = (S, T)$, $\mathcal{R}_l$ is a ***early*** bisimulation relation if $\mathcal{R}_l$ is the largest relation such that

$$\begin{array}{lll} \forall s_1, s_2 \in S \quad s_1 \, \mathcal{R}_l \, s_2 & \Rightarrow & \forall (s_1, a, t_1) \in T \\ & & \boxed{\forall \sigma} \quad \exists (s_2, a', t_2) \in T \text{ such that} \\ & & a\sigma \succeq a'\sigma \quad \wedge \quad t_1\sigma \, \mathcal{R}_l \, t_2\sigma \\ & & \wedge \textit{... the symmetric case...} \end{array}$$

$\mathcal{R}_e$ is a ***late*** bisimulation relation if $\mathcal{R}_e$ is the largest relation such that

$$\begin{array}{lll} \forall s_1, s_2 \in S \quad s_1 \, \mathcal{R}_e \, s_2 & \Rightarrow & \forall (s_1, a, t_1) \in T \\ & & \exists (s_2, a', t_2) \in T \quad \boxed{\forall \sigma} \text{ such that} \\ & & a\sigma \succeq a'\sigma \quad \wedge \quad t_1\sigma \, \mathcal{R}_e \, t_2\sigma \\ & & \wedge \textit{... the symmetric case...} \end{array}$$

Implementing Symbolic Bisimulation Checking — I

```
:- table nbisim/2.

nbisim(S1, S2) :-
   strans(S1, A1, C1, T1),
   C1,
   no_matching_trans(S2, S1, A1, T1).
nbisim(S1, S2) :- nbisim(S2, S1).

no_matching_trans(S2, S1, A1, T1) :-
   forall((A2,C2,T2),
      (S1,A1,T1,S2)^strans(S2, A2, C2, T2),
      nsimulate(A1,T1, A2,C2,T2) ).

bisim(S1, S2) :- tnot(nbisim(S1,S2)).
```


Implementing Symbolic Bisimulation Checking — II

```
nsimulate(A1,T1, A2,C2,T2) :-
   (C2,
      copy_term((A1,T1,A2,T2), (B1,U1,B2,U2)), % Late
      (A1,T1,A2,T2) = (B1,U1,B2,U2), % Early
      ( B2 =>= B1, nbisim(U1, U2)
      ; B2 />= B1)
   )
   ; not(C2)
   .
```

[TAPD'00]