STATE-SPACE EXPLORATION: SYMBOLIC TECHNIQUES (continued)

Stavros Tripakis
University of California, Berkeley

SYMBOLIC REACHABILITY ANALYSIS
Recall: Symbolic Representation of Kripke Structures

$$(P, \text{Init}, \text{Trans})$$

where

- $P = \{x_1, x_2, \ldots, x_n\}$: set of boolean state variables, also taken to be the atomic propositions.
- Predicate $\text{Init}(\vec{x})$ on vector $\vec{x} = (x_1, \ldots, x_n)$ represents the set $S_0$ of initial states.
- Predicate $\text{Trans}(\vec{x}, \vec{x}')$ represents the transition relation $R$.

Recall: Symbolic Representation

- Set of states = predicate $\phi(\vec{x})$ on vector of state variables $\vec{x}$. E.g.:
  - $\text{Init}(x, y, z) : x \land \neg y$
  - $\text{Bad}(x_1, x_2) : x_1 = \text{crit} \land x_2 = \text{crit}$

- Transition relation = predicate $\text{Trans}(\vec{x}, \vec{x}')$ on state variables and next-state variables. E.g.:
  - $\text{Trans}(x, y, x', y') : x' = x + 1 \land (y' = 0 \lor y' = 1)$

- How do we perform set-theoretic operations with predicates?
  - Union of two sets represented by $\phi_1$ and $\phi_2$: $\phi_1 \lor \phi_2$.
  - Intersection of two sets represented by $\phi_1$ and $\phi_2$: $\phi_1 \land \phi_2$.
  - Complement of a set represented by $\phi$: $\neg \phi$. 
Symbolic Reachability Analysis

Main idea:

- Start with set of initial states $S_0$.
- Compute $S_1 := S_0 \cup \{\text{all 1-step successors of } S_0\}$.
- Compute $S_2 := S_1 \cup \{\text{all 1-step successors of } S_1\}$.
- ... 
- Until $S_{k+1} = S_k$.
- $S_k$ contains all reachable states.

Computing Successors Symbolically

Given a set of states represented as a predicate $\phi(x)$.

We want to compute a new predicate $\phi'$, representing the set of all 1-step successors of states in $\phi(x)$.
Predicate Transformer

Successors can be computed by a **predicate transformer**:

\[
\text{succ}(\phi(\vec{x})) := (\exists \vec{x} : \phi(\vec{x}) \land \text{Trans}(\vec{x},\vec{x}'))[\vec{x}' \sim \vec{x}]
\]

- \(\exists \vec{x} : \phi(\vec{x}) \land \text{Trans}(\vec{x},\vec{x}')\): successors of states in \(\phi\)
- \([\vec{x}' \sim \vec{x}]\): renames variables so that resulting predicate is over current state variables

Example:

\[
\begin{align*}
\phi &= 0 \leq x \leq 5 \\
\text{Trans} &= x \leq x' \leq x + 1 \\
\text{succ}(\phi) &= (\exists x : 0 \leq x \leq 5 \land x \leq x' \leq x + 1)[x' \sim x] \\
&= (\exists x : 0 \leq x \leq 5 \land 0 \leq x' \leq 5 + 1)[x' \sim x] \\
&= (0 \leq x' \leq 6)[x' \sim x] \\
&= 0 \leq x \leq 6
\end{align*}
\]

How to do quantifier elimination automatically?

In the case of propositional logic, quantifier elimination is simple. Suppose \(x\) is a boolean variable:

\[
\exists x : \phi \equiv \phi[x \sim 0] \lor \phi[x \sim 1]
\]
Predicate Transformer: Another Example

\[
\text{succ}(p \land q) = (\exists p, q : p \land q \land \text{Trans})[p' \sim p, q' \sim q]
\]
\[
= (\exists p, q : p \land q \land p' \land q')[p' \sim p, q' \sim q]
\]
\[
= (p' \land q')[p' \sim p, q' \sim q]
\]
\[
= p' \land q
\]

Symbolic Reachability Analysis Algorithm

1: \texttt{Reachable} := \texttt{Init};
2: \texttt{terminate} := \texttt{false};
3: \texttt{repeat}
4: \hspace{1em} \texttt{tmp} := \texttt{Reachable} \lor \texttt{succ(Reachable)};
5: \hspace{1em} \texttt{if} \texttt{tmp} \equiv \texttt{Reachable} \texttt{then}
6: \hspace{2em} \texttt{terminate} := \texttt{true};
7: \hspace{1em} \texttt{else}
8: \hspace{2em} \texttt{Reachable} := \texttt{tmp};
9: \hspace{1em} \texttt{end if}
10: \texttt{until} \texttt{terminate}
11: \texttt{return} \texttt{Reachable};

Does the algorithm terminate? Why?

Quiz: modify the algorithm to make it check reachability of a set of bad states characterized by predicate \textit{Bad}.
Symbolic Reachability Algorithm: checking for \textit{Bad} states

1: \textit{Reachable} := \textit{Init};  
2: \text{terminate} := \text{false};  
3: \text{error} := \text{false};  
4: \textbf{repeat}  
5: \text{tmp} := \textbf{reachable} \lor \textbf{succ} (\textit{reachable});  
6: \textbf{if} \text{ tmp} \equiv \textit{reachable} \textbf{then}  
7: \text{terminate} := \text{true};  
8: \textbf{else}  
9: \textit{reachable} := \text{tmp};  
10: \textbf{end if}  
11: \textbf{if} \text{ SAT} (\textit{reachable} \land \textit{Bad}) \textbf{then}  
12: \text{error} := \text{true};  
13: \textbf{end if}  
14: \textbf{until} \text{ terminate or error}  
15: \textbf{return} (\textit{reachable}, \text{error});

Symbolic Model-Checking: Example

Let’s model-check this symbolically!  
We want to check that all reachable states satisfy $p \lor q$.  
In temporal logic parlance:

\begin{align*}
\text{CTL: } & AG(p \lor q) \\
\text{LTL: } & G(p \lor q)
\end{align*}
Symbolic Model-Checking: Implementation

- For finite-state systems, boolean variables can be used to encode state.
- All predicates then become boolean expressions.
- Efficient data structures for boolean expressions:
  - **BDDs (Binary Decision Diagrams):** see [Bryant, 1992], available from course web site
- Efficient algorithms for implementing logical operations (conjunction, disjunction, satisfiability check) on BDDs.
- In-depth discussion: Computer-Aided Verification course.

Example: BDD

![BDD Diagram]

Can you guess which boolean expression this BDD represents?

\[ x_4 \left( \overline{x}_3 (\overline{x}_2 + x_2 \overline{x}_1) + x_3 (\overline{x}_2 \overline{x}_1 + x_2) \right) + \overline{x}_4 x_2 x_1 \]
Facts about BDDs

- Variable ordering can greatly influence BDD size:
  - For the same boolean function, different variable orderings may result in very different in size BDDs.
  - For example, consider the function
    \[(x_1 \land y_1) \lor (x_2 \land y_2) \lor (x_3 \land y_3)\]
    and the two orderings:
    \[x_1, y_1, x_2, y_2, x_3, y_3\]
    and
    \[x_1, x_2, x_3, y_1, y_2, y_3\]
- Some BDDs have exponential size no matter which ordering we pick.
- For an extensive account of BDDs, see Bryant's paper, also Section 6.7 of [Baier and Katoen, 2008].

FINITE-HORIZON REACHABILITY
(a.k.a. BOUNDED MODEL-CHECKING)
Bounded Model-Checking

Question:

*Can a “bad” state be reached in up to \( n \) steps (transitions)?*

i.e., does there exist a path

\[ s_0, s_1, \ldots, s_k \]

where \( k \leq n \), \( s_0 \in S_0 \) and \( s_k \in \text{Bad} \), for some given set \( \text{Bad} \).

Key idea:

*Reduce the above question to a SAT (satisfiability) problem.*

- SAT problem NP-complete for propositional logic, generally undecidable for predicate logic.

- In practice, today’s SAT solvers can handle formulas with thousands of variables: see [Malik and Zhang, 2009], available from course web site.

- BMC exploits this.

Bounded Model-Checking (BMC)

Suppose I have predicates \( \text{Init}(x) \), \( \text{Trans}(x, x') \), and \( \text{Bad}(x) \).

How to use them for BMC?

- Bad state reachable in 0 steps iff

\[
\text{SAT} \left( \text{Init}(x) \land \text{Bad}(x) \right)
\]

- Bad state reachable in 1 step iff

\[
\text{SAT} \left( \text{Init}(x_0) \land \text{Trans}(x_0, x_1) \land \text{Bad}(x_1) \right)
\]

- ...

- Bad state reachable in \( n \) steps iff

\[
\text{SAT} \left( \text{Init}(x_0) \land \text{Trans}(x_0, x_1) \land \cdots \land \text{Trans}(x_{n-1}, x_n) \land \text{Bad}(x_n) \right)
\]
BMC Algorithm – Outer Loop

1: \textbf{for all} \ k = 0, 1, \ldots, n \ \textbf{do}
2: \ \phi := \text{Init}(\vec{x}_0) \land \text{Trans}(\vec{x}_0, \vec{x}_1) \land \cdots \land \text{Trans}(\vec{x}_{k-1}, \vec{x}_k) \land \text{Bad}(\vec{x}_k);
3: \ \textbf{if} \ \text{SAT}(\phi) \ \textbf{then}
4: \ \quad \text{print “Bad state reachable in } k \text{ steps”;}
5: \ \quad \text{output solution as counter-example;}
6: \ \quad \textbf{end if}
7: \ \textbf{end for}
8: \ \text{print “Bad state unreachable up to } n \text{ steps”;}

BMC: Soundness and Completeness

1: \textbf{for all} \ k = 0, 1, \ldots, n \ \textbf{do}
2: \ \phi := \text{Init}(\vec{x}_0) \land \text{Trans}(\vec{x}_0, \vec{x}_1) \land \cdots \land \text{Trans}(\vec{x}_{k-1}, \vec{x}_k) \land \text{Bad}(\vec{x}_k);
3: \ \textbf{if} \ \text{SAT}(\phi) \ \textbf{then}
4: \ \quad \text{print “Bad state reachable in } k \text{ steps”;}
5: \ \quad \text{output solution as counter-example;}
6: \ \quad \textbf{end if}
7: \ \textbf{end for}
8: \ \text{print “Bad state unreachable up to } n \text{ steps”;}

BMC algorithm is \textbf{sound} in the following sense:

- if algorithm reports “reachable” then indeed a bad state is reachable
- if algorithm reports “unreachable up to } n \text{ steps” then there is no path of length } \leq n \text{ that reaches a bad state.}

Can we make BMC \textbf{complete}? 

- It should report \textbf{unreachable} iff there are no reachable bad states (w.r.t. any bound).
- Is this even possible in general? For finite-state systems?
Complete BMC: “brute-force” threshold

1: for all \( k = 0, 1, \ldots, n \) do
2: \( \phi := Init(\vec{x}_0) \land Trans(\vec{x}_0, \vec{x}_1) \land \cdots \land Trans(\vec{x}_{k-1}, \vec{x}_k) \land Bad(\vec{x}_k); \)
3: if SAT(\( \phi \)) then
4: print “Bad state reachable in \( k \) steps”;
5: output solution as counter-example;
6: end if
7: end for
8: print “Bad state unreachable up to \( n \) steps”;

A finite-state Kripke structure is essentially a finite graph.

How can we turn BMC into a complete method for finite-state structures?

If we know \(|S|\) (the number of all possible states) then we can set \( n := |S| \).

With 100 boolean variables, \(|S| = 2^{100}\), so this doesn’t work.

Complete BMC: a better threshold

**Reachability diameter**: number of steps that it takes to reach any reachable state.

\[
d := \min \{ i \mid \forall s \in \text{Reach} : \exists \text{ path } s_0, s_1, \ldots, s_j : j \leq i \land s_0 \in S_0 \land s_j = s \}
\]

where Reach is the set of reachable states.

\( d \) is generally a much better threshold than \(|S|\). **Why?**

\( d \leq |\text{Reach}| \leq |S| \).

**Problem**: we don’t know \(|\text{Reach}|\), therefore how to compute \( d \)?
Complete BMC: the Completeness Threshold

**Recurrence diameter**: length of the longest loop-free path.

\[ r := \max \{ i \mid \exists \text{ path } s_0, s_1, \ldots, s_i : s_0 \in S_0 \land \forall 0 \leq j < k \leq i : s_j \neq s_k \} \]

Claim: \( d \leq r \). Why?

Can we compute \( r \)? How?

Use a SAT solver!

\[ r := \max \{ i \mid \text{SAT} \left( \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} \bar{x}_j \neq \bar{x}_k \right) \} \]

**FROM REACHABILITY TO GENERAL SAFETY MODEL-CHECKING**
Model-checking safety properties

Suppose we want to check the safety property

\[ G(p \Rightarrow Xq) \]

where \( p, q \) are atomic propositions.

Can we reduce this model-checking problem to reachability? How?

Yes! Using safety monitors.

Verification using safety monitors

Monitor observes the outputs of Design, and moves to error state as soon as design has violated the safety property.

What would a Monitor for \( G(p \Rightarrow Xq) \) look like?

(Assuming that Monitor synchronizes with Design at every transition of Design. More on this when we talk about composition.)
Verification using liveness monitors

We can use the idea of Monitors also for checking liveness.

- Monitors essentially try to find counter-examples = behaviors that violate the property we are interested in.
- If a counter-example is found, the system violates the property. If not, the system satisfies the property.
- For safety properties, counter-examples are finite behaviors reaching an error state.
- What is a counter-example to a liveness property?
  - A counter-example to a liveness property is an infinite behavior.
  - For finite-state systems, such a behavior corresponds to a lasso, i.e., a finite segment followed by a cycle.
Non-Deterministic Büchi Automata (NBA)

An NBA is described in the same way as an NFA (non-deterministic finite automaton), i.e., as a tuple

$$(\Sigma, S, S_0, \Delta, F)$$

But the interpretation is different:
- An NFA accepts finite words, that is, elements of $\Sigma^*$.  
- An NBA accepts infinite words, that is, elements of $\Sigma^\omega$.  

An element of $\Sigma^\omega$ is an infinite sequence

$$a_1a_2a_3\cdots$$

where for every $i$, $a_i \in \Sigma$.

**NBA acceptance condition**: a word is accepted if there exists a run generating this word such that the run visits an accepting state (i.e., a state in $F$) **infinitely often**.

Büchi automaton: example

What does this Büchi automaton accept?

All infinite words where $a$ appears infinitely often.
Non-Deterministic vs. Deterministic Büchi Automata

What does this Büchi automaton accept?
All infinite words ending with $bbb \ldots$.

This Büchi automaton is non-deterministic. Why?

Does there exist an equivalent (i.e., accepting the same language) deterministic Büchi automaton? No!

Back to verification using liveness monitors

Liveness Monitor = Büchi Automaton.

What would a Liveness Monitor for $G(p \Rightarrow Fq)$ look like?

The monitor accepts all behaviors that violate the property!
Basic algorithms for checking liveness

- We consider only enumerative (explicit-state) algorithms.

- The problem: given a finite graph, where some nodes are marked *accepting*, find whether there is a reachable cycle that visits at least one accepting node.

- The graph is the result of composing the Design with the Büchi Monitor.
  - As with Safety Monitors, the Monitor synchronizes with the Design.

- Can you think of algorithms that can be used to find cycles in a graph?
  - **Strongly-Connected Components** by Tarjan.
    \( O(n + m) \) (\( n = \#\)nodes, \( m = \#\)edges).
  - What about DFS?

---

### Bibliography