EECS 144/244: Fundamental Algorithms for System Modeling, Analysis, and Optimization

Discrete Systems

Lecture: State-Space Exploration

Stavros Tripakis

University of California, Berkeley

ANALYSIS OF DISCRETE SYSTEMS: STATE-SPACE EXPLORATION

State-Space Exploration

Goal: explore all reachable states of the system (modeled as a transition system).

Main application: exhaustive verification by *reachability analysis*.

- Check that system is never in an "incorrect" state (*safety*)
  - deadlock state
  - state which does not satisfy a given (state) property
  - e.g., "train is at intersection but gate is not lowered"
  - "autopilot is off but pilot thinks it is on"
  - ...

State-Space Exploration

Goal: explore all reachable states of the system (modeled as a transition system).

Main application: exhaustive verification by *reachability analysis*.

- Check that system is never in an “incorrect” state (*safety*)
  - deadlock state
  - state which does not satisfy a given (state) property
  - e.g., “train is at intersection but gate is not lowered”
  - “autopilot is off but pilot thinks it is on”
  - ...
- Also the basis for checking *liveness* properties: every so often system does something useful.

State-Space Exploration

- For finite-state systems, it can be done fully automatically! (in principle)
- Forms the basis for *model-checking*
  - Turing award 2007: Clarke, Emerson, Sifakis.
  - Established practice in the industry (mainly hardware, but increasingly also software).

Model-Checking and State-Space Exploration

Model-checking: check a model M (transition system) against a specification ϕ:

$$M \models \phi?$$

- Often ϕ is a temporal logic formula:
  - If ϕ is an LTL formula: $M \models \phi$ means $\sigma \models \phi$ for *every* trace of M.
  - If ϕ is a CTL formula: $M \models \phi$ means $s_0 \models \phi$ for *every* initial state s_0 of M.

Model-Checking and State-Space Exploration

Model-checking: check a model M (transition system) against a specification ϕ:

$$M \models \phi?$$

- Often ϕ is a temporal logic formula:
  - If ϕ is an LTL formula: $M \models \phi$ means $\sigma \models \phi$ for *every* trace of M.
  - If ϕ is a CTL formula: $M \models \phi$ means $s_0 \models \phi$ for *every* initial state s_0 of M.

State-space exploration: the basis for model-checking. E.g.,

- Can build monitor M_ϕ and reduce the LTL model-checking question to a state-space exploration question about the composition of M and M_ϕ (we will revisit this when we talk about composition).
- Can extend symbolic reachability analysis (later in this lecture) to CTL model-checking.

State-Space Exploration Algorithms

- Enumerative (also called "explicit state").
- Symbolic
  - Bounded model-checking using SAT/SMT solvers.
  - Symbolic reachability.

An Enumerative Algorithm: Depth-First Search

Assume given: Kripke structure (P, S, S_0, L, R).

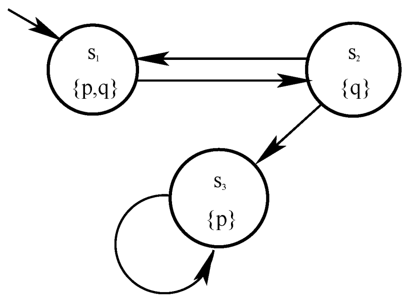
main:

$V := \emptyset$; /* V: set of visited states */
for all $s \in S_0$ **do**
  DFS(s);
end for

DFS(s):

check s; /* is s a deadlock? is given $p \in L(s)$? ... */
$V := V \cup \{s\}$;
for all s' such that $(s, s') \in R$ **do**
  if $s' \notin V$ **then**
    DFS(s'); /* recursive call */
  end if
end for

An Enumerative Algorithm: Depth-First Search



Let's simulate the algorithm on this graph.

An Enumerative Algorithm: Depth-First Search

Quiz:

- Does the algorithm terminate?
- Does it visit all reachable states?
- Does it visit any unreachable states?
- What is the complexity of the algorithm?

Enumerative Methods

Many algorithms: DFS, BFS, A*, ...

Many approaches to combat state-space explosion: partial-order reduction, symmetry reduction, bit-state hashing, ...

In-depth discussion: Computer-Aided Verification course (see also bibliography).

SYMBOLIC METHODS

Symbolic Methods: Why?

The plague of exhaustive verification: *state explosion*.

- A chip with 100 flip-flops: 2^{100} (potentially reachable) states.
- That is $1267650600228229401496703205376$ states.
- Even if each state costs 1 bit to store, this still makes $2^{100-60-8} = 2^{32} = 4,294,967,296$ exabytes ...

Symbolic methods aim to improve this.

A seminal paper was subtitled "*Symbolic model checking:* 10^{20} *states and beyond.*" [Burch et al., 1990].

10^{20} is still less than 2^{67}, but a great leap forward at that time.

Symbolic Representation of State Spaces

Key idea:

Instead of reasoning about individual states, reason about **sets** *of states.*

How do we represent a set of states?

Symbolic representation:

Set $=$ *predicate.*

Set of states $=$ *predicate on state variables.*

Symbolic Representation of Sets of States

Examples:

1. Assume 3 state variables, p, q, r, of type boolean.

$$S_1 : \qquad p \vee q$$

Symbolic Representation of Sets of States

Examples:

1. Assume 3 state variables, p, q, r, of type boolean.

$$S_1 : \qquad p \vee q = \{p\overline{q}r, p\overline{q}\overline{r}, \overline{p}qr, \overline{p}q\overline{r}, pqr, pq\overline{r}\}$$

Symbolic Representation of Sets of States

Examples:

1. Assume 3 state variables, p, q, r, of type boolean.

$$S_1 : \qquad p \vee q = \{p\overline{q}r, p\overline{q}\overline{r}, \overline{p}qr, \overline{p}q\overline{r}, pqr, pq\overline{r}\}$$

2. Assume 3 state variables, x, i, b, of types real, integer, boolean.

$$S_2 : \qquad x > 0 \wedge (b \rightarrow i \geq 0)$$

   How many states are in S_2?

Symbolic Representation of Transition Relations

Key idea:

> *Use a predicate on* **two copies** *of the state variables: unprimed (current state) + primed (next state).*

If $\vec{x}$ is the vector of state variables, then the transition relation R is a predicate on $\vec{x}$ and $\vec{x}'$:

$$R(\vec{x}, \vec{x}')$$

e.g.,

$$R(x, i, b, x', i', b')$$

Symbolic Representation of Transition Relations

Examples:

1. Assume one state variable, p, of type boolean.

$$R_1 : \qquad (p \rightarrow \neg p') \wedge (\neg p \rightarrow p')$$

Which transition relation does this represent? Is it a relation or a function (deterministic)?

Symbolic Representation of Transition Relations

Examples:

1. Assume one state variable, p, of type boolean.

$$R_1 : \qquad (p \rightarrow \neg p') \wedge (\neg p \rightarrow p')$$

   Which transition relation does this represent? Is it a relation or a function (deterministic)?

2. Assume one state variable, n, of type integer.

$$R_2 : \qquad n' = n + 1 \vee n' = n$$

   Which transition relation does this represent? Is it a relation or a function (deterministic)?

Symbolic Representation of Kripke Structures

Kripke structure:

$$(P, S, S_0, L, R)$$

Symbolic representation:

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

where

- $P = \{x_1, x_2, ..., x_n\}$: set of (boolean) state variables, also also taken to be the atomic propositions.[1]
- Predicate $\mathit{Init}(\vec{x})$ on vector $\vec{x} = (x_1, ..., x_n)$ represents the set S_0 of initial states.
- Predicate $\mathit{Trans}(\vec{x}, \vec{x}')$ represents the transition relation R.

[1]this is done for simplicity, the two could be separated

Symbolic Representation of Kripke Structures

Kripke structure:

$$(P, S, S_0, L, R)$$

Symbolic representation:

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

where

- $P = \{x_1, x_2, ..., x_n\}$: set of (boolean) state variables, also also taken to be the atomic propositions.[1]
- Predicate $\mathit{Init}(\vec{x})$ on vector $\vec{x} = (x_1, ..., x_n)$ represents the set S_0 of initial states.
- Predicate $\mathit{Trans}(\vec{x}, \vec{x}')$ represents the transition relation R.

Basis of the language of NuSMV.

[1] this is done for simplicity, the two could be separated

Example: NuSMV model

```
MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = FALSE
TRANS
  next(output) = !input | next(output) = output
```

What is the Kripke structure defined by this NuSMV program?

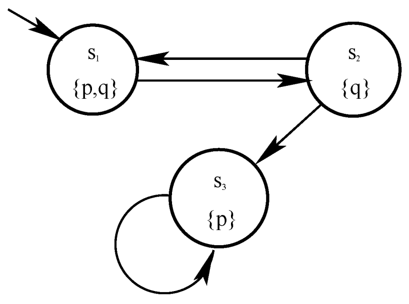
Example: NuSMV model

```
MODULE inverter(input)
VAR
  output : boolean;
INIT
  output = FALSE
TRANS
  next(output) = !input | next(output) = output
```

What is the Kripke structure defined by this NuSMV program?

What about P and L?

Example: Kripke Structure



Represent this symbolically.

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

$$s_0, s_1, ..., s_k$$

where $k \leq n$, $s_0 \in S_0$ and $s_k \in Bad$, for some given set Bad.

Bounded Model-Checking

Question:

Can a "bad" state be reached in up to n steps (transitions)?

i.e., does there exist a path

$$s_0, s_1, ..., s_k$$

where $k \leq n$, $s_0 \in S_0$ and $s_k \in Bad$, for some given set 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.
- SMT (SAT Modulo Theory) solvers also making good progress.
- BMC exploits this.

Bounded Model-Checking (BMC)

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

How to use them for BMC?

- Bad state reachable in 0 steps iff

$$\mathsf{SAT}\Big(\qquad\qquad\qquad\qquad \Big)$$

Bounded Model-Checking (BMC)

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

How to use them for BMC?

- Bad state reachable in 0 steps iff

$$\mathsf{SAT}\big(Init(\vec{x}) \wedge Bad(\vec{x})\big)$$

Bounded Model-Checking (BMC)

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

How to use them for BMC?

- Bad state reachable in 0 steps iff

$$\mathsf{SAT}\big(Init(\vec{x}) \wedge Bad(\vec{x})\big)$$

- Bad state reachable in 1 step iff

$$\mathsf{SAT}\big(\qquad\qquad\qquad\qquad\qquad\qquad \big)$$

Bounded Model-Checking (BMC)

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

How to use them for BMC?

- Bad state reachable in 0 steps iff

$$\mathsf{SAT}\big(Init(\vec{x}) \land Bad(\vec{x})\big)$$

- Bad state reachable in 1 step iff

$$\mathsf{SAT}\big(Init(\vec{x}_0) \land Trans(\vec{x}_0, \vec{x}_1) \land Bad(\vec{x}_1)\big)$$

Bounded Model-Checking (BMC)

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

How to use them for BMC?

- Bad state reachable in 0 steps iff

$$\mathsf{SAT}\big(Init(\vec{x}) \wedge Bad(\vec{x})\big)$$

- Bad state reachable in 1 step iff

$$\mathsf{SAT}\big(Init(\vec{x}_0) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge Bad(\vec{x}_1)\big)$$

- ...
- Bad state reachable in n steps iff

$$\mathsf{SAT}\big(\qquad\qquad\qquad\qquad\qquad\qquad \big)$$

Bounded Model-Checking (BMC)

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

How to use them for BMC?

- Bad state reachable in 0 steps iff

$$\mathsf{SAT}\big(\mathit{Init}(\vec{x}) \wedge \mathit{Bad}(\vec{x})\big)$$

- Bad state reachable in 1 step iff

$$\mathsf{SAT}\big(\mathit{Init}(\vec{x}_0) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge \mathit{Bad}(\vec{x}_1)\big)$$

- ...
- Bad state reachable in n steps iff

$$\mathsf{SAT}\big(\mathit{Init}(\vec{x}_0) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge \mathit{Trans}(\vec{x}_{n-1}, \vec{x}_n) \wedge \mathit{Bad}(\vec{x}_n)\big)$$

for all $k = 0, 1, ..., n$ **do**
  $\phi := \mathit{Init}(\vec{x}_0) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge \mathit{Trans}(\vec{x}_{k-1}, \vec{x}_k) \wedge \mathit{Bad}(\vec{x}_k)$;
  if SAT(ϕ) **then**
    print "Bad state reachable in k steps";
    output solution as counter-example;
  end if
end for
print "Bad state unreachable up to n steps";

SAT Solving

SAT: given propositional logic formula, check whether it is satisfiable, and return a solution if it is.

Usually formula given in CNF: *Conjunctive Normal Form*.

CNF and DNF

Literal: a variable x or its negation $\overline{x}$.

Clause: a disjunction of literals. E.g.:

$$\begin{aligned} \text{clause 1} &: \quad x + y \\ \text{clause 2} &: \quad \overline{x} + z + w \end{aligned}$$

CNF: conjunction of clauses, i.e., conjunction of disjunctions of literals (also called POS - "product of sums"). E.g.:

$$(x + y) \cdot (\overline{x} + z + w) \cdots$$

DNF: disjunction of conjunctions of literals (also called SOP - "sum of products"). E.g.:

$$(xy) + (\overline{x}zw) + \cdots$$

Translations to/from CNF and DNF

Every formula can be trivially transformed into DNF. How?

Are there more efficient ways to transform into DNF? (Hint: how easy is it to check whether a DNF formula is SAT? how hard is SAT?)

Translations to/from CNF and DNF

Every formula can be trivially transformed into DNF. How?

Are there more efficient ways to transform into DNF? (Hint: how easy is it to check whether a DNF formula is SAT? how hard is SAT?)

NNF (Negation Normal Form): all negations are "pushed" into literals. E.g.:

$(x \wedge y) \rightarrow (z \wedge w) \quad \rightsquigarrow$

Translations to/from CNF and DNF

Every formula can be trivially transformed into DNF. How?

Are there more efficient ways to transform into DNF? (Hint: how easy is it to check whether a DNF formula is SAT? how hard is SAT?)

NNF (Negation Normal Form): all negations are "pushed" into literals. E.g.:

$$(x \wedge y) \rightarrow (z \wedge w) \quad \rightsquigarrow \quad (\neg(x \wedge y)) \vee (z \wedge w) \quad \rightsquigarrow$$

Translations to/from CNF and DNF

Every formula can be trivially transformed into DNF. How?

Are there more efficient ways to transform into DNF? (Hint: how easy is it to check whether a DNF formula is SAT? how hard is SAT?)

NNF (Negation Normal Form): all negations are "pushed" into literals. E.g.:

$$(x \wedge y) \rightarrow (z \wedge w) \quad \rightsquigarrow \quad (\neg(x \wedge y)) \vee (z \wedge w) \quad \rightsquigarrow \quad (\neg x \vee \neg y) \vee (z \wedge w)$$

Homework: write a procedure to convert any boolean expression to NNF.

Given a formula in NNF, how to transform it into CNF?

Translations to/from CNF and DNF

CNF(ϕ):
if ϕ is a literal **then**
    return ϕ;
else if ϕ is $\phi_1 \wedge \phi_2$ **then**
    return CNF(ϕ_1) $\wedge$ CNF(ϕ_2);
else if ϕ is $\phi_1 \vee \phi_2$ **then**
    return DistributeOr$\big($CNF(ϕ_1), CNF(ϕ_2)$\big)$;
else
    error: ϕ not in NNF;
end if

DistributeOr(ϕ_1, ϕ_2):
if ϕ_1 is $\phi_{11} \wedge \phi_{12}$ **then**
    return DistributeOr(ϕ_{11}, ϕ_2) $\wedge$ DistributeOr(ϕ_{12}, ϕ_2);
else if ϕ_2 is $\phi_{21} \wedge \phi_{22}$ **then**
    return DistributeOr(ϕ_1, ϕ_{21}) $\wedge$ DistributeOr(ϕ_1, ϕ_{22});
else
    return $\phi_1 \vee \phi_2$;   /* both must be literals at this point */
end if

Translations to/from CNF and DNF

CNF(ϕ):
if ϕ is a literal **then**
    return ϕ;
else if ϕ is $\phi_1 \wedge \phi_2$ **then**
    return CNF(ϕ_1) $\wedge$ CNF(ϕ_2);
else if ϕ is $\phi_1 \vee \phi_2$ **then**
    return DistributeOr(CNF(ϕ_1), CNF(ϕ_2));
else
    error: ϕ not in NNF;
end if

DistributeOr(ϕ_1, ϕ_2):
if ϕ_1 is $\phi_{11} \wedge \phi_{12}$ **then**
    return DistributeOr(ϕ_{11}, ϕ_2) $\wedge$ DistributeOr(ϕ_{12}, ϕ_2);
else if ϕ_2 is $\phi_{21} \wedge \phi_{22}$ **then**
    return DistributeOr(ϕ_1, ϕ_{21}) $\wedge$ DistributeOr(ϕ_1, ϕ_{22});
else
    return $\phi_1 \vee \phi_2$;   /* both must be literals at this point */
end if

How large can CNF(ϕ) be in the worst-case?

Translations to/from CNF and DNF

More efficient translations to CNF exist: they preserve satisfiability of the original formula, by adding extra variables.

Example:

$$ab + cd \quad \rightsquigarrow \quad \underbrace{(a+c)(a+d)(b+c)(b+d)}_{\text{normal translation}}$$

$$ab + cd \quad \rightsquigarrow \quad \underbrace{(z_1+z_2)(\overline{z_1}+a)(\overline{z_1}+b)(\overline{z_2}+c)(\overline{z_2}+d)}_{\text{translation adding variables } z_1, z_2}$$

Idea: z_1 represents ab and z_2 represents cd.

SAT Solving

SAT: given propositional logic formula, check whether it is satisfiable, and return a solution if it is.

Usually formula given in CNF: *Conjunctive Normal Form*.

Solvers exploit this form, e.g.:

- Easy to detect *conflicts* (unsatisfiability).

SAT Solving

Let ϕ be a formula in CNF.
Let $x_1, x_2, ..., x_n$ be the variables appearing in ϕ.

Brute-force algorithm:

- Explore the tree of all assignments to $x_1, x_2, ..., x_n$.
- As soon as a partial assignment results in a conflict, prune the entire subtree and backtrack.
- Either a complete assignment that works is found, or formula is UNSAT.

Worst-case performance: $O(2^n)$.

SAT Solving: Resolution

From two clauses $x + A$ and $\overline{x} + B$, derive new clause $A + B$.

Example:

$$(x + y)(\overline{x} + z) \equiv (x + y)(\overline{x} + z)(y + z)$$

Why?

SAT Solving: Resolution

From two clauses $x + A$ and $\overline{x} + B$, derive new clause $A + B$.

Example:

$$(x + y)(\overline{x} + z) \equiv (x + y)(\overline{x} + z)(y + z)$$

Why?

A form of *learning* (new clauses).

SAT Solving: Resolution

We can assume that every variable x appears both in "positive" form x and in "negative" form $\overline{x}$ in ϕ, that is, ϕ contains at least two clauses $(x + \cdots)$ and $(\overline{x} + \cdots)$.

What if it's not the case?

SAT Solving: Resolution

We can assume that every variable x appears both in "positive" form x and in "negative" form $\overline{x}$ in ϕ, that is, ϕ contains at least two clauses $(x + \cdots)$ and $(\overline{x} + \cdots)$.

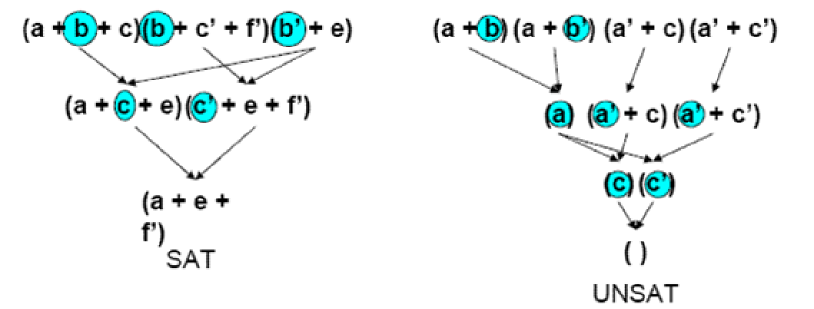
What if it's not the case?

Davis-Putnam algorithm:

for every variable x in ϕ **do**
    for every pair of clauses $A : (x + \cdots)$ and $B : (\overline{x} + \cdots)$ in ϕ **do**
        resolve A and B and add resolvent to ϕ;
        check UNSAT (empty clause);
    end for
    remove from ϕ original clauses containing x or $\overline{x}$;
    remove variables that don't appear both in positive and negative form;
    check SAT;
end for

SAT Solving: Davis-Putnam Algorithm

Examples (a' means negation, i.e., $\overline{a}$):



From Sanjit Seshia.

SAT Solving

Modern SAT solvers use much more involved (and efficient) techniques.

In-depth discussion: Computer-Aided Verification course.

Back to BMC: Soundness vs. Completeness

for all $k = 0, 1, ..., n$ **do**
  $\phi := \mathit{Init}(\vec{x}_0) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge \mathit{Trans}(\vec{x}_{k-1}, \vec{x}_k) \wedge \mathit{Bad}(\vec{x}_k)$;
  if SAT(ϕ) **then**
    print "Bad state reachable in k steps";
    output solution as counter-example;
  end if
end for
print "Bad state unreachable up to n steps";

BMC algorithm is *sound* in the following sense:

- if algorithm reports "reachable" then indeed a bad state is reachable
- if algorithm reports "unreachable" then a bad state is unreachable **but only up to n steps**.

Back to BMC: Soundness vs. Completeness

for all $k = 0, 1, ..., n$ **do**
    $\phi := Init(\vec{x}_0) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge Trans(\vec{x}_{k-1}, \vec{x}_k) \wedge Bad(\vec{x}_k)$;
    if SAT(ϕ) **then**
        print "Bad state reachable in k steps";
        output solution as counter-example;
    end if
end for
print "Bad state unreachable up to n steps";

BMC algorithm is *sound* in the following sense:

- if algorithm reports "reachable" then indeed a bad state is reachable
- if algorithm reports "unreachable" then a bad state is unreachable **but only up to n steps**.

Can we make BMC *complete*?

- Reports unreachable iff bad states are unreachable (w.r.t. any bound).

Back to BMC: Soundness vs. Completeness

```
for all k = 0, 1, ..., n do
    φ := Init(x⃗_0) ∧ Trans(x⃗_0, x⃗_1) ∧ ··· ∧ Trans(x⃗_{k-1}, x⃗_k) ∧ Bad(x⃗_k);
    if SAT(φ) then
        print "Bad state reachable in k steps";
        output solution as counter-example;
    end if
end for
print "Bad state unreachable up to n steps";
```

BMC algorithm is *sound* in the following sense:

- if algorithm reports "reachable" then indeed a bad state is reachable
- if algorithm reports "unreachable" then a bad state is unreachable **but only up to n steps**.

Can we make BMC *complete*?

- Reports unreachable iff bad states are unreachable (w.r.t. any bound).
- Is this even possible in general?

Back to BMC: Soundness vs. Completeness

```
for all k = 0, 1, ..., n do
    φ := Init(x⃗_0) ∧ Trans(x⃗_0, x⃗_1) ∧ ··· ∧ Trans(x⃗_{k−1}, x⃗_k) ∧ Bad(x⃗_k);
    if SAT(φ) then
        print "Bad state reachable in k steps";
        output solution as counter-example;
    end if
end for
print "Bad state unreachable up to n steps";
```

BMC algorithm is *sound* in the following sense:

- if algorithm reports "reachable" then indeed a bad state is reachable
- if algorithm reports "unreachable" then a bad state is unreachable **but only up to n steps**.

Can we make BMC *complete*?

- Reports unreachable iff bad states are unreachable (w.r.t. any bound).
- Is this even possible in general? For finite-state systems?

Complete BMC: the trivial threshold

for all $k = 0, 1, ..., n$ **do**
  $\phi := \mathit{Init}(\vec{x}_0) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge \mathit{Trans}(\vec{x}_{k-1}, \vec{x}_k) \wedge \mathit{Bad}(\vec{x}_k)$;
  if SAT(ϕ) **then**
    print "Bad state reachable in k steps";
    output solution as counter-example;
  end if
end for
print "Bad state unreachable up to n steps";

A finite-state Kripke structure $K = (P, S, S_0, L, R)$ is essentially a finite graph.

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

Complete BMC: the trivial threshold

```
for all $k = 0, 1, ..., n$ do
    $\phi := Init(\vec{x}_0) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge Trans(\vec{x}_{k-1}, \vec{x}_k) \wedge Bad(\vec{x}_k)$;
    if SAT($\phi$) then
        print "Bad state reachable in $k$ steps";
        output solution as counter-example;
    end if
end for
print "Bad state unreachable up to $n$ steps";
```

A finite-state Kripke structure $K = (P, S, S_0, L, R)$ is essentially a finite graph.

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

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

Complete BMC: the trivial threshold

for all $k = 0, 1, ..., n$ **do**
  $\phi := \mathit{Init}(\vec{x}_0) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge \mathit{Trans}(\vec{x}_{k-1}, \vec{x}_k) \wedge \mathit{Bad}(\vec{x}_k)$;
  if SAT(ϕ) **then**
    print "Bad state reachable in k steps";
    output solution as counter-example;
  end if
end for
print "Bad state unreachable up to n steps";

A finite-state Kripke structure $K = (P, S, S_0, L, R)$ is essentially a finite graph.

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

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: the Completeness Threshold

Reachability diameter : maximum number of steps that it takes to reach all states.

$d := \min\{i \mid \forall s \in \text{Reach} : \exists \text{ path } s_0, s_1, ..., s_j \text{ in } K : j \leq i \wedge s_0 \in S_0 \wedge s_j = s\}$

where Reach is the set of reachable states of K.

Complete BMC: the Completeness Threshold

Reachability diameter : maximum number of steps that it takes to reach all states.

$d := \min\{i \mid \forall s \in \text{Reach} : \exists \text{ path } s_0, s_1, ..., s_j \text{ in } K : j \leq i \wedge s_0 \in S_0 \wedge s_j = s\}$

where Reach is the set of reachable states of K.

d is a much better threshold than $|S|$. Why?

Complete BMC: the Completeness Threshold

Reachability diameter : maximum number of steps that it takes to reach all states.

$d := \min\{i \mid \forall s \in \mathsf{Reach} : \exists \text{ path } s_0, s_1, ..., s_j \text{ in } K : j \leq i \wedge s_0 \in S_0 \wedge s_j = s\}$

where Reach is the set of reachable states of K.

d is a much better threshold than $|S|$. Why?
$d \leq |\mathsf{Reach}| \leq |S|$. It could in practice be much smaller.

Complete BMC: the Completeness Threshold

Reachability diameter : maximum number of steps that it takes to reach all states.

$d := \min\{i \mid \forall s \in \text{Reach} : \exists \text{ path } s_0, s_1, ..., s_j \text{ in } K : j \leq i \wedge s_0 \in S_0 \wedge s_j = s\}$

where Reach is the set of reachable states of K.

d is a much better threshold than $|S|$. Why?
$d \leq |\text{Reach}| \leq |S|$. It could in practice be much smaller.

Problem: we don't know Reach, therefore how to compute d?

Complete BMC: the Completeness Threshold

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

$r := \max\{i \mid \exists$ path $s_0, s_1, ..., s_i$ in $K : s_0 \in S_0 \wedge \forall 0 \leq j < k \leq i : s_j \neq s_k\}$

where Reach is the set of reachable states of K.

Complete BMC: the Completeness Threshold

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

$r := \max\{i \mid \exists$ path $s_0, s_1, ..., s_i$ in $K : s_0 \in S_0 \wedge \forall 0 \leq j < k \leq i : s_j \neq s_k\}$

where Reach is the set of reachable states of K.

Then: $d \leq r$. Why?

Complete BMC: the Completeness Threshold

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

$r := \max\{i \mid \exists$ path $s_0, s_1, ..., s_i$ in $K : s_0 \in S_0 \wedge \forall 0 \leq j < k \leq i : s_j \neq s_k\}$

where Reach is the set of reachable states of K.

Then: $d \leq r$. Why?

How to compute r?

Complete BMC: the Completeness Threshold

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

$r := \max\{i \mid \exists$ path $s_0, s_1, ..., s_i$ in $K : s_0 \in S_0 \wedge \forall 0 \leq j < k \leq i : s_j \neq s_k\}$

where Reach is the set of reachable states of K.

Then: $d \leq r$. Why?

How to compute r?

Express it in propositional logic!

$$r := \max\{i \mid \mathsf{SAT}\Big(\mathit{Init}(\vec{x}_0) \wedge \mathit{Trans}(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge \mathit{Trans}(\vec{x}_{i-1}, \vec{x}_i) \wedge \bigwedge_{j=0}^{i-1} \bigwedge_{k=j+1}^{i} \vec{x}_j \neq \vec{x}_k \Big)\}$$

EXHAUSTIVE SYMBOLIC REACHABILITY (a.k.a. SYMBOLIC MODEL-CHECKING)

Symbolic Reachability

Recall:

- Set of states = predicate $\phi(\vec{x})$ on vector of state variables $\vec{x}$. E.g.:
  - $Init(x, y, z) \,:\, x \wedge \neg y$
  - $Bad(x_1, x_2) \,:\, x_1 = crit \wedge x_2 = crit$
- Transition relation = predicate $Trans(\vec{x}, \vec{x}')$ on state variables and next-state variables. E.g.:
  - $Trans(x, y, x', y') : x' = x + 1 \wedge (y' = 0 \vee y' = 1)$

Symbolic Reachability: Predicate Transformer

Then:

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

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \wedge Trans(\vec{x}, \vec{x}')\big)[\vec{x}' \rightsquigarrow \vec{x}]$$

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

Symbolic Reachability: Predicate Transformer

Then:

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

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \rightsquigarrow \vec{x}]$$

  - $\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')$: successors of states in ϕ
  - $[\vec{x}' \rightsquigarrow \vec{x}]$: renames variables so that resulting predicate is over current state variables

Example:

$$\begin{array}{rcl} \phi & = & 0 \leq x \leq 5 \\ \mathit{Trans} & = & x \leq x' \leq x+1 \\ \mathbf{succ}(\phi) & = & (\exists x : 0 \leq x \leq 5 \wedge x \leq x' \leq x+1)[x' \rightsquigarrow x] \end{array}$$

Symbolic Reachability: Predicate Transformer

Then:

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

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \rightsquigarrow \vec{x}]$$

  - $\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')$: successors of states in ϕ
  - $[\vec{x}' \rightsquigarrow \vec{x}]$: renames variables so that resulting predicate is over current state variables

Example:

$$\begin{array}{rcl}
\phi & = & 0 \leq x \leq 5 \\
\mathit{Trans} & = & x \leq x' \leq x + 1 \\
\mathbf{succ}(\phi) & = & (\exists x : 0 \leq x \leq 5 \wedge x \leq x' \leq x + 1)[x' \rightsquigarrow x] \\
 & = & (\exists x : 0 \leq x \leq 5 \wedge 0 \leq x' \leq 5 + 1)[x' \rightsquigarrow x]
\end{array}$$

Symbolic Reachability: Predicate Transformer

Then:

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

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \leadsto \vec{x}]$$

  - $\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')$: successors of states in ϕ
  - $[\vec{x}' \leadsto \vec{x}]$: renames variables so that resulting predicate is over current state variables

Example:

$$\begin{array}{rcl} \phi & = & 0 \leq x \leq 5 \\ \mathit{Trans} & = & x \leq x' \leq x + 1 \\ \mathbf{succ}(\phi) & = & (\exists x : 0 \leq x \leq 5 \wedge x \leq x' \leq x + 1)[x' \leadsto x] \\ & = & (\exists x : 0 \leq x \leq 5 \wedge 0 \leq x' \leq 5 + 1)[x' \leadsto x] \\ & = & (0 \leq x' \leq 6)[x' \leadsto x] \end{array}$$

Symbolic Reachability: Predicate Transformer

Then:

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

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \leadsto \vec{x}]$$

  - $\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')$: successors of states in ϕ
  - $[\vec{x}' \leadsto \vec{x}]$: renames variables so that resulting predicate is over current state variables

Example:

$$\begin{array}{rcl}
\phi & = & 0 \le x \le 5 \\
\mathit{Trans} & = & x \le x' \le x + 1 \\
\mathbf{succ}(\phi) & = & (\exists x : 0 \le x \le 5 \wedge x \le x' \le x + 1)[x' \leadsto x] \\
 & = & (\exists x : 0 \le x \le 5 \wedge 0 \le x' \le 5 + 1)[x' \leadsto x] \\
 & = & (0 \le x' \le 6)[x' \leadsto x] \\
 & = & 0 \le x \le 6
\end{array}$$

Symbolic Reachability: Predicate Transformer

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \land \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \leadsto \vec{x}]$$

How do we implement $\mathbf{succ}$ in practice?

In particular, how to do quantifier elimination automatically?

In the case of propositional logic, quantifier elimination becomes a simple procedure:

$$\exists x : \phi(x) \quad \equiv$$

Symbolic Reachability: Predicate Transformer

$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \land \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \leadsto \vec{x}]$$

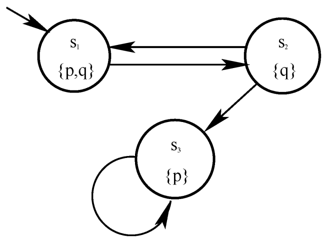
How do we implement $\mathbf{succ}$ in practice?

In particular, how to do quantifier elimination automatically?

In the case of propositional logic, quantifier elimination becomes a simple procedure:

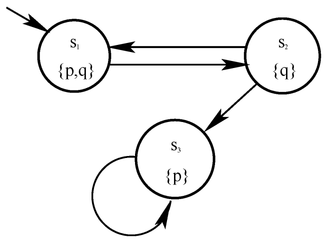
$$\exists x : \phi(x) \quad \equiv \quad \phi(x)[x \leadsto 0] \lor \phi(x)[x \leadsto 1]$$

Predicate Transformer: Another Example



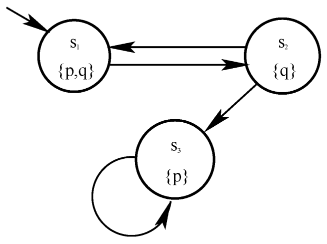
$$\mathbf{succ}(p \wedge q) \quad = \quad (\exists p, q : p \wedge q \wedge \mathit{Trans})[p' \rightsquigarrow p, q' \rightsquigarrow q]$$

Predicate Transformer: Another Example



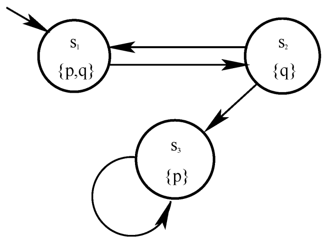
$$
\begin{aligned}
\mathbf{succ}(p \wedge q) &= (\exists p, q : p \wedge q \wedge \mathit{Trans})[p' \rightsquigarrow p, q' \rightsquigarrow q] \\
&= (\exists p, q : p \wedge q \wedge \overline{p}' \wedge q')[p' \rightsquigarrow p, q' \rightsquigarrow q]
\end{aligned}
$$

Predicate Transformer: Another Example



$$
\begin{aligned}
\mathbf{succ}(p \wedge q) &= (\exists p, q : p \wedge q \wedge \mathit{Trans})[p' \rightsquigarrow p, q' \rightsquigarrow q] \\
&= (\exists p, q : p \wedge q \wedge \overline{p}' \wedge q')[p' \rightsquigarrow p, q' \rightsquigarrow q] \\
&= (\overline{p}' \wedge q')[p' \rightsquigarrow p, q' \rightsquigarrow q]
\end{aligned}
$$

Predicate Transformer: Another Example



$$
\begin{aligned}
\mathbf{succ}(p \wedge q) &= (\exists p, q : p \wedge q \wedge \mathit{Trans})[p' \rightsquigarrow p, q' \rightsquigarrow q] \\
&= (\exists p, q : p \wedge q \wedge \overline{p}' \wedge q')[p' \rightsquigarrow p, q' \rightsquigarrow q] \\
&= (\overline{p}' \wedge q')[p' \rightsquigarrow p, q' \rightsquigarrow q] \\
&= \overline{p} \wedge q
\end{aligned}
$$

Symbolic Reachability

$Reachable := Init$;
terminate := false;
repeat
    $tmp := Reachable \vee \mathbf{succ}(Reachable)$;
    if $tmp \equiv Reachable$ **then**
        terminate := true;
    else
        $Reachable := tmp$;
    end if
until terminate
return $Reachable$;

Symbolic Reachability

$Reachable := Init$;
terminate := false;
repeat
    $tmp := Reachable \vee \mathbf{succ}(Reachable)$;
    if $tmp \equiv Reachable$ **then**
        terminate := true;
    else
        $Reachable := tmp$;
    end if
until terminate
return $Reachable$;

Does the algorithm terminate? Why?

$Reachable := Init$;
terminate := false;
repeat
    $tmp := Reachable \vee \textbf{succ}(Reachable)$;
    if $tmp \equiv Reachable$ **then**
        terminate := true;
    else
        $Reachable := tmp$;
    end if
until terminate
return $Reachable$;

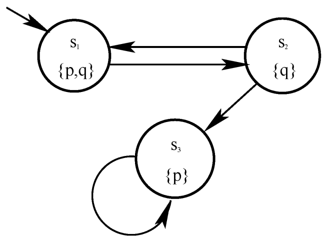
Does the algorithm terminate? Why?

Quiz: modify the algorithm to make it check reachability of a set of bad states characterized by predicate Bad.

Symbolic Reachability: checking for Bad states

```
Reachable := Init;
terminate := false;
error := false;
repeat
    tmp := Reachable ∨ succ(Reachable);
    if tmp ≡ Reachable then
        terminate := true;
    else
        Reachable := tmp;
    end if
    if SAT(Reachable ∧ Bad) then
        error := true;
    end if
until terminate or error
return (Reachable,error);
```

Symbolic Model-Checking: Example



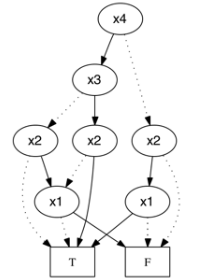
Let's model-check this symbolically!
We want to check that all reachable states satisfy $p \vee q$.
In temporal logic parlance:

$$\begin{array}{rl} \text{CTL:} & AG(p \vee q) \\ \text{LTL:} & \Box(p \vee q) \end{array}$$

Symbolic Model-Checking: Implementation

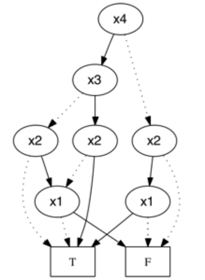
- For finite-state systems, boolean variables can be used to encode state.
- All predicates then become boolean expressions.
- Efficient data structures for boolean expressions:
  - BDDs (Binary Decision Diagrams)
- Efficient algorithms for implementing logical operations (conjunction, disjunction, satisfiability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

Example: BDD



Can you guess which boolean expression this BDD represents?

Example: BDD



Can you guess which boolean expression this BDD represents?

$$\overline{x_2}\ \overline{x_3}x_4 + \overline{x_1}(x_2\overline{x_3}x_4 + \overline{x_2}x_3x_4) + x_2x_3x_4 + x_1x_2\overline{x_4}$$

Symbolic Model-Checking

Homework: Assuming that the system only has boolean variables, express deadlock as a predicate on state (or next state) variables. Note: the predicate must have no quantifiers.

Bibliography I

Baier, C. and Katoen, J.-P. (2008).
Principles of Model Checking.
MIT Press.

Barrett, C. W., Sebastiani, R., Seshia, S. A., and Tinelli, C. (2009).
Satisfiability modulo theories.
In Biere, A., Heule, M., van Maaren, H., and Walsh, T., editors, *Handbook of Satisfiability*, volume 185, pages 825–885. IOS Press.

Biere, A., Cimatti, A., Clarke, E. M., Strichman, O., and Zhu, Y. (2003).
Bounded model checking.
Advances in Computers, 58:117–148.

Burch, J., Clarke, E., Dill, D., Hwang, L., and McMillan, K. (1990).
Symbolic model checking: 10^{20} states and beyond.
In *5th LICS*, pages 428–439. IEEE.

Clarke, E., Grumberg, O., and Peled, D. (2000).
Model Checking.
MIT Press.

Godefroid, P. and Wolper, P. (1991).
Using partial orders for the efficient verification of deadlock freedom and safety properties.
In *4th CAV*.

Holzmann, G. (1998).
An analysis of bitstate hashing.
In *Formal Methods in System Design*, pages 301–314. Chapman & Hall.

Bibliography II

Huth, M. and Ryan, M. (2004).
Logic in Computer Science: Modelling and Reasoning about Systems.
Cambridge University Press.

Jain, H. and Clarke, E. M. (2009).
Efficient SAT solving for non-clausal formulas using DPLL, graphs, and watched cuts.
In *46th Annual Design Automation Conference*, DAC '09, pages 563–568. ACM.

Robinson, J. (1965).
A machine-oriented logic based on the resolution principle.
Journal of the ACM, 12(1).

Tseitin, G. S. (1968).
On the complexity of derivations in the propositional calculus.
Studies in Mathematics and Mathematical Logic, Part II:115–125.

Valmari, A. (1990).
Stubborn sets for reduced state space generation.
LNCS 483.

Zhang, L. and Malik, S. (2002).
The quest for efficient boolean satisfiability solvers.
In *14th International Conference on Computer Aided Verification*, CAV '02, pages 17–36.