EE 244: Fundamental Algorithms for System Modeling, Analysis, and Optimization Fall 2016

Model Checking

Stavros Tripakis
University of California, Berkeley

Recall: the model-checking problems for LTL and CTL

Given:

- the implementation: a transition system (Kripke structure) $M = (\mathsf{AP}, S, S_0, L, R)$
- the specification: a temporal logic (LTL or CTL) formula ϕ

check where M satisfies ϕ:

$$M \stackrel{?}{\models} \phi$$

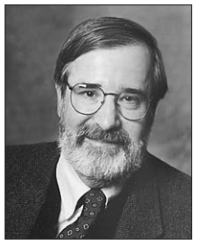
- If ϕ is LTL: **every** execution trace of M must satisfy ϕ.
- If ϕ is CTL: **every** initial state of M must satisfy ϕ.

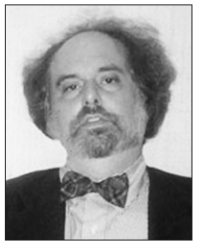
For finite-state M, the question can be answered fully automatically!

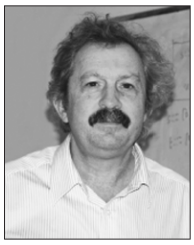
ACM Turing Award for Model-Checking

Clarke, Emerson, and Sifakis won the ACM Turing Award in 2007,

> *for their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries.*


Edmund M. Clarke


E. Allen Emerson


Joseph Sifakis

Simplest model-checking problem: checking **invariants**

Suppose ϕ is of the form

$$\mathbf{G}\psi \qquad \text{or} \qquad \mathbf{AG}\psi$$

where ψ is a propositional formula (boolean expression on atomic propositions).

E.g.,

$$\mathbf{G}(p \vee q), \qquad \mathbf{G}(p \rightarrow q), \qquad \cdots$$

Then ψ is **invariant**: it must hold at all **reachable** states.

Examples:

- "Whenever train is at intersection the gate must be lowered"
- "If the autopilot is off then the pilot must not believe it is on"

Reachability Analysis and State-Space Exploration

Suppose we want to model-check an invariant, i.e., check whether transition system (Kripke structure) M satisfies $\mathbf{G}\psi$, for boolean expression ψ.

Model checking such formulas is conceptually easy:

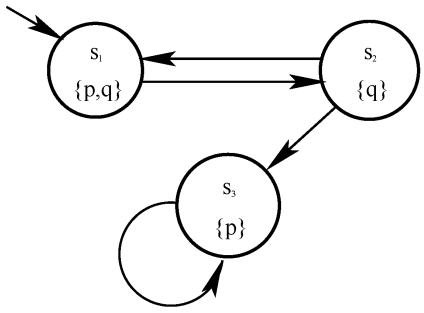
- Explore (generate) all reachable states of M.
- Check that every one of them satisfies ψ. (Is this easy? Why?)

This is called **reachability analysis**.

- For finite-state systems, it can be done exhaustively and fully automatically!
- ... at least in theory ... in practice, often **state explosion** ...

Recall: Transition System (Kripke Structure)

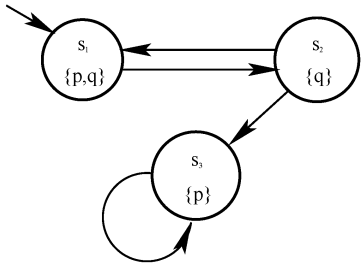
A tuple (P, S, S_0, L, R).



- P: set of atomic propositions, e.g., $P = \{p, q\}$.
- S: set of states, e.g., $S = \{s_1, s_2, s_3\}$.
- S_0: set of initial states, could be more than one, in this example just one: $S_0 = \{s_1\}$.
- $L : S \to 2^P$: labeling function, e.g., $L(s_1) = \{p, q\}$, $L(s_2) = \{q\}$, ...
- $R \subseteq S \times S$: transition relation, e.g., $R = \{(s_1, s_2), (s_2, s_1), (s_2, s_3), (s_3, s_3)\}$.

Reachable States

Given transition system (P, S, S_0, L, R).



A state $s \in S$ is called **reachable** if there exists a finite path (in the transition system) reaching that state:

$$s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_k, \text{ such that } k \geq 0 \text{ and } s_k = s.$$

The path is formed by initial state $s_0 \in S_0$, and transitions $(s_i, s_{i+1}) \in R$, for $i = 0, ..., k-1$.

Why would some states be unreachable?

E.g., a counter modulo 10, represented in 4 bits.

Caveat: Deadlocks

We have implicitly assumed that our system is **deadlock-free**.

Deadlock: a state with no successors:

$$s \text{ is a deadlock iff } \nexists s' : s \longrightarrow s'$$

Are deadlocks problematic for checking invariants? Why?

Only infinite paths count for the verification of a property such as $\mathbf{G}p$. If the system deadlocks after every time it violates p, then, formally speaking, it satisfies $\mathbf{G}p$!

How can we check that a given system is deadlock-free?

Use reachability analysis!

Reachability analysis: summary

- Generate all reachable states ...
- ... while at the same time checking that each of them is "OK", i.e.,
  - it is not a deadlock state
  - it does not violate an invariant
  - ...

Reachability Algorithms

- **Enumerative** (also called "explicit state").
  - These are basically search algorithms on directed graphs.
- **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).

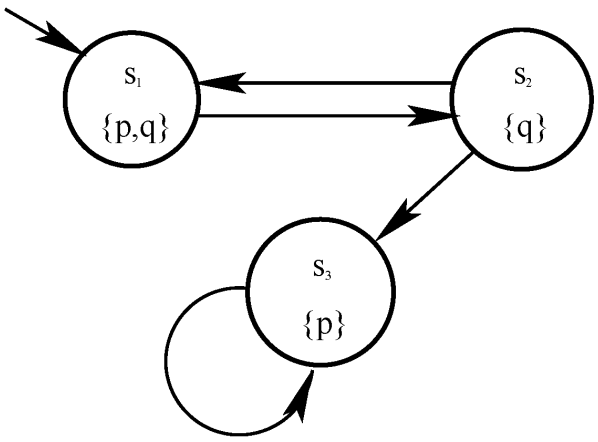
main:

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

DFS(s):

1: check s; /* is s a deadlock? is given $p \in L(s)$? ... */
2: $V := V \cup \{s\}$;
3: **for all** s' such that $(s, s') \in R$ **do**
4: **if** $s' \notin V$ **then**
5: DFS(s'); /* recursive call */
6: **end if**
7: **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? Yes, if state space is finite.
- Does it visit all reachable states? Yes: if s is reachable, then either $s \in S_0$, or s is the immediate successor of some s', which is itself reachable. In the first case, s is inserted into V because of the main loop. In the second case, assuming (by induction) that s' is inserted to V, s will also be inserted to V by loop in lines 3-6.
- Does it visit any unreachable states? No: following the "inverse" of the argument above, if s is inserted into V, either this is done because of the main loop, or because of the loop in lines 3-6. In the first case, s must be in S_0, so it's an initial state, so it's reachable. In the second case, s must be successor of some s', which by induction must be itself in V, therefore reachable.
- What is the complexity of the algorithm? $O(n+m)$ where n is number of nodes/states and m is number of edges/transitions in the graph. Every node and edge are visited at most once.

Other enumerative algorithms

Every search algorithm on finite graphs can be used for reachability analysis:

- DFS: depth-first search
- BFS: breadth-first search
- Best-first search:
  - every state is assigned a "value" (using some heuristic value function, e.g., how "close" we are likely to be to the goal – in our case a "bad" state) and then next state to explore is the one with the highest value.
- A*: classic search technique in artificial intelligence.
- ...

Other enumerative algorithms

Every search algorithm on finite graphs can be used for reachability analysis: DFS, BFS, A*, ...

- Most of these have been tried by researchers in verification.
- Basic complexity is the same for all: need to store all reachable states
  - in the "worst case" from the algorithmic point of view
  - but in fact "best case" from the verification point of view, since we are trying to prove that our system is correct! $\Rightarrow$ all reachable states must be correct
- **State explosion**: the number of reachable states is too large

State explosion

- How many states does a chip with 100 flip-flops have?
  - 2^{100} (potentially reachable) states.
  - That is $1267650600228229401496703205376$ states.
  - Even if each state costs 1 bit to store, this still makes $2^{100-60-3} = 2^{37} = 137,438,953,472$ exabytes ...
  - Even if only $\frac{1}{32}$ states are reachable, this still makes $2^{100-5} = 2^{95}$ states.
- How many states does a piece of concurrent software have? Assume n asynchronous processes (e.g., threads), with k states each.
  - k^n (potentially reachable) states.
- What if the processes also communicate with queues? Consider a single queue of size ℓ (i.e., can hold ℓ messages), and m possible types of messages.
  - m^ℓ (potentially reachable) states for just one queue.

The real complexity of reachability

Searching a graph is linear in the size of the graph, which appears to be a very nice worst-case complexity ...

... until we realize that the size of the graph is **exponential** in the number of state variables, processes, etc.

This is not just a practical observation. There is theoretical complexity results about this, e.g., checking intersection emptiness of a **set** of DFA is PSPACE-complete.

Enumerative methods to remedy state explosion

- **Bit-state hashing**: instead of storing the entire state vector, just store 1 bit per state: its hash value [Holzmann, 1998].
  - Do you see a problem with this method?
  - **Incomplete**: two states may hash to the same value ⇒ only one will be visited ⇒ some reachable states may be missed!
  - And as we saw, even 1 bit per state may be too much already.
- **Partial-order reduction**: in asynchronous concurrent systems, transitions of different processes are often independent ⇒ no need to explore all interleavings [Valmari, 1990, Godefroid and Wolper, 1991].
- **Symmetry reduction**: many state spaces are symmetric ⇒ equivalence relation on states ⇒ suffices to explore just one state per equivalence class [Ip and Dill, 1996, Clarke et al., 1998, Sistla and Godefroid, 2004].
- ...

All these help, but don't eliminate the state-explosion problem.
Note: above references are representative, there is a lot more work on these topics.
In-depth discussion: Computer-Aided Verification course by Sanjit Seshia.

SYMBOLIC METHODS

Symbolic Methods: Why?

Motivation: attack the state explosion problem.

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

10^{20} is less than 2^{67}, so still not quite enough for modern circuits.

Nevertheless: 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.

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

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

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

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., for three state variables, x, i, b:

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

Symbolic Representation of Transition Relations

Examples:

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

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

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

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

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

   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:

$$(P, Init, Trans)$$

where

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

Basis of the language of NuSMV.

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

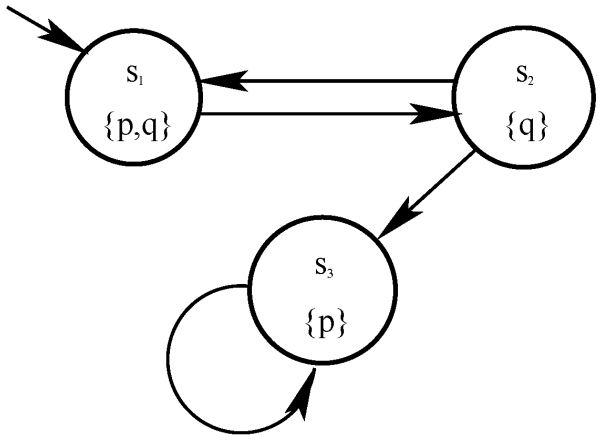
Example: NuSMV model

```
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?

What about P and L?

Example: Kripke Structure



Represent this symbolically.

SYMBOLIC REACHABILITY ANALYSIS

Recall: Symbolic Representation of Kripke Structures

$$(P, Init, Trans)$$

where

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

Recall: Symbolic Representation

- Set of states = predicate $\phi(\vec{x})$ on vector of state variables $\vec{x}$. E.g.:
  - $Init(x, y, z) : x \wedge \neg y$
  - $Bad(x_1, x_2) : x_1 = crit \wedge x_2 = crit$
- Transition relation = predicate $Trans(\vec{x}, \vec{x}')$ on state variables and next-state variables. E.g.:
  - $Trans(x, y, x', y') : x' = x + 1 \wedge (y' = 0 \vee y' = 1)$
- How do we perform set-theoretic operations with predicates?
  - Union of two sets represented by ϕ_1 and ϕ_2: $\phi_1 \vee \phi_2$.
  - Intersection of two sets represented by ϕ_1 and ϕ_2: $\phi_1 \wedge \phi_2$.
  - Complement of a set represented by ϕ: $\neg\phi$.

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 $\phi(\vec{x})$.

We want to compute a new predicate ϕ', representing the set of **all 1-step successors** of states in $\phi(\vec{x})$.

Predicate Transformer

- 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:

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

Predicate Transformer

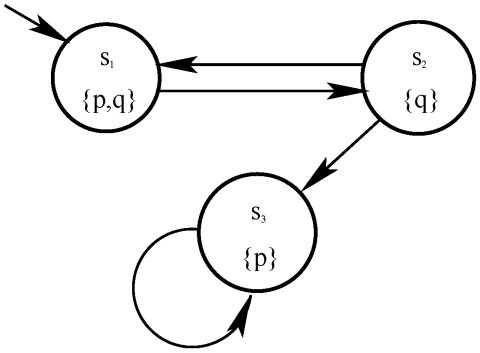
$$\mathbf{succ}\big(\phi(\vec{x})\big) := \big(\exists \vec{x} : \phi(\vec{x}) \wedge \mathit{Trans}(\vec{x}, \vec{x}')\big)[\vec{x}' \rightsquigarrow \vec{x}]$$

How to do quantifier elimination automatically?

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

$$\exists x : \phi \quad \Leftrightarrow \quad \phi[x \rightsquigarrow 0] \vee \phi[x \rightsquigarrow 1]$$

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 Analysis Algorithm

1: $\mathit{Reachable} := \mathit{Init}$;
2: terminate := false;
3: **repeat**
4: $\quad \mathit{tmp} := \mathit{Reachable} \vee \mathbf{succ}(\mathit{Reachable})$;
5: $\quad$ **if** $\mathit{tmp} \Leftrightarrow \mathit{Reachable}$ **then**
6: $\quad\quad$ terminate := true;
7: $\quad$ **else**
8: $\quad\quad \mathit{Reachable} := \mathit{tmp}$;
9: $\quad$ **end if**
10: **until** terminate
11: return $\mathit{Reachable}$;

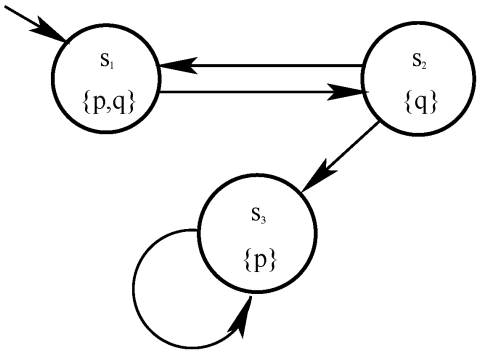
Does the algorithm terminate? Why?

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

1: $Reachable := Init$;
2: terminate := false;
3: error := false;
4: **repeat**
5: $\quad tmp := Reachable \vee \mathbf{succ}(Reachable)$;
6: $\quad$ **if** $tmp \Leftrightarrow Reachable$ **then**
7: $\quad\quad$ terminate := true;
8: $\quad$ **else**
9: $\quad\quad Reachable := tmp$;
10: $\quad$ **end if**
11: $\quad$ **if** SAT$(Reachable \wedge Bad)$ **then**
12: $\quad\quad$ error := true;
13: $\quad$ **end if**
14: **until** terminate or error
15: return $(Reachable$,error);

Symbolic Reachability: Example



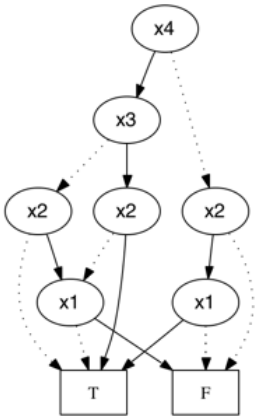
Let's check this system symbolically!
We want to check that all reachable states satisfy $p \vee q$.
In temporal logic parlance:

CTL: $\mathbf{AG}(p \vee q)$
LTL: $\mathbf{G}(p \vee q)$

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)** [Bryant, 1992] (paper available in bcourses - follow link from lectures web page)
- Efficient algorithms for implementing logical operations (conjunction, disjunction, satisfiability check, ...) on BDDs.
- Note: logical operations correspond to set-theoretic operations:
  - Conjunction: intersection
  - Disjunction: union
  - Satisfiability check: emptiness check
  - ...

Example: BDD



Can you guess which boolean expression this BDD represents?

$$x_4\big(\overline{x_3}(\overline{x_2} + x_2\overline{x_1}) + x_3(\overline{x_2}\,\overline{x_1} + x_2)\big) + \overline{x_4}x_2x_1$$

BDDs

Binary decision trees

Binary decision tree:

- A tree representing all possible variable assignments, and corresponding truth values of a boolean expression.
- For n variables, the tree has $1 + 2 + 2^2 + \cdots + 2^n = 2^{n+1} - 1$ nodes (including the leaves).

Let's draw the binary decision tree for

$$(z_1 \wedge z_3) \vee (z_2 \wedge z_3)$$

(assuming the order of variables z_1, z_2, z_3).

From binary decision trees to BDDs

Main idea: make the representation compact (i.e., smaller) by eliminating redundant nodes.

- If two subtrees (including leaves) T_1 and T_2 are identical then keep only T_1. All incoming links to T_2 are redirected to T_1.
- If both the true-branch and the false-branch of a node v lead to the same node v', then node v is redundant: v can be removed, with its incoming links being redirected to v'.

The result is a **reduced ordered binary decision diagram** (ROBDD).
It is a **DAG**: directed acyclic graph.
We often use BDD to mean ROBDD.

Let's try this on the following formulas:

$$a + b, \qquad \text{and} \qquad (z_1 \wedge z_3) \vee (z_2 \wedge z_3)$$

From binary decision trees to BDDs

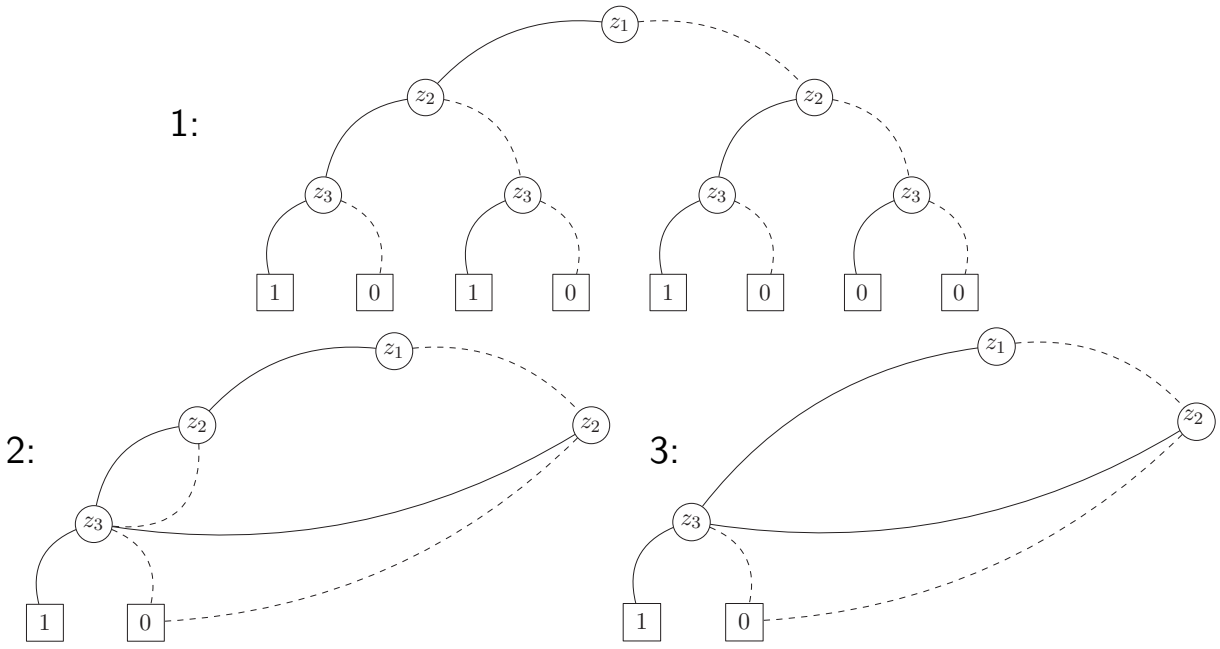


Figure taken from [Baier and Katoen, 2008].

BDDs: a canonical representation of boolean functions

ROBDDs are a **canonical** representation of boolean functions.

This means that two boolean functions (or expressions) f_1 and f_2 are equivalent iff their corresponding ROBDDs (for the same variable ordering) are identical.

Is this an important property? What is an example where it is useful?

Recall the symbolic reachability algorithm stopping criterion:

$$tmp \Leftrightarrow Reachable$$

If B and B' are the BDDs representing tmp and $Reachable$, respectively, then $tmp \Leftrightarrow Reachable$ holds iff B and B' are identical.

The bad news: variable ordering matters greatly

- BDD size depends on variable ordering
  - For the same boolean function, different variable orderings may result BDDs which are very different in size.
  - For example, consider the function

    $$(x_1 \wedge y_1) \vee (x_2 \wedge y_2) \vee (x_3 \wedge y_3)$$

    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.
- Deciding whether a given order is optimal is NP-hard.
- Land of heuristics ...

Operations on BDDs

We want to compute set-theoretic, or equivalently, logical, operations on BDDs:

- Check for emptiness / satisfiability.
- Check for universality / validity.
- Intersection / conjunction.
- Union / disjunction.
- Complementation / negation.

Which of these operations are easy to perform on ROBDDs?

Operations on BDDs

- Check for emptiness / satisfiability.
  - Check whether the BDD is the leaf 0. If yes $\Rightarrow$ empty / unsat.
- Check for universality / validity.
  - Check whether the BDD is the leaf 1. If yes $\Rightarrow$ valid.
- Complementation / negation.
  - Replace the leaf 0 with 1, and 1 with 0.

We next look at conjunction and disjunction.

Shannon expansion

Let f be a boolean expression and x be a boolean variable.

Recall:

$$f[x \rightsquigarrow 0]$$

denotes the new formula f' obtained by replacing any occurrence of x in f by 0.

Similarly for $f[x \rightsquigarrow 1]$.

$f[x \rightsquigarrow 1]$ and $f[x \rightsquigarrow 0]$ are called the (positive and negative) **cofactors** of f, and are denoted f_x and $f_{\overline{x}}$.

Then

$$f \quad \Leftrightarrow \quad \underbrace{\overline{x} \cdot f_{\overline{x}} \; + \; x \cdot f_x}_{\text{this is called the Shannon expansion of } f}$$

Shannon expansion

$$f \quad \Leftrightarrow \quad \overline{x} \cdot f_{\overline{x}} \; + \; x \cdot f_x$$

This is the essence of binary decision trees and BDDs: if f is the root, then

- f_x is the sub-tree rooted at the 0-branch ("*false*"-branch) child of f
- $f_{\overline{x}}$ is the sub-tree rooted at the 1-branch ("*true*"-branch) child of f

Recursive application of boolean operations based on Shannon expansion

Suppose $\odot$ is some boolean operation (e.g., conjunction or disjunction).

Let f and g be two boolean expressions, and x be a boolean variable (usually f and g refer to x, but they don't have to).

Then

$$f \odot g \quad \Leftrightarrow \quad \overline{x} \cdot (f_{\overline{x}} \odot g_{\overline{x}}) \; + \; x \cdot (f_x \odot g_x)$$

For instance, if $\odot$ is conjunction:

$$f \cdot g \quad \Leftrightarrow \quad \overline{x} \cdot f_{\overline{x}} \cdot g_{\overline{x}} \; + \; x \cdot f_x \cdot g_x$$

This leads to the `apply` function.

The `apply` function

- Takes as input:
  - A boolean operation $\odot$ (e.g., conjunction or disjunction).
  - Two BDDs B_f and B_g (with the same variable ordering) representing two boolean functions f and g.
- Computes as output:
  - A BDD B representing $f \odot g$:

$$B = \texttt{apply}(\odot, B_f, B_g) \quad \text{such that} \quad B \Leftrightarrow B_{f \odot g}$$

- Operates recursively based on Shannon expansion.
- Resulting BDD may not be reduced, so needs to be generally reduced afterwards.

The apply function

We are computing $\texttt{apply}(\odot, B_f, B_g)$. Let v_f and v_g be the root nodes of B_f and B_g respectively.

There are the following cases to consider:

1. Both v_f and v_g are leaves (i.e., 0 or 1). Then, apply returns the leaf BDD with truth value $v_f \odot v_g$.
2. Both v_f and v_g are internal x-*nodes*, i.e., corresponding to variable x. Then, let B_f^x, B_g^x be the *positive sub-BDDs* (i.e., positive cofactors, i.e., BDDs rooted at the *true*-branch children) of v_f and v_g, respectively; and similarly with $B_f^{\overline{x}}, B_g^{\overline{x}}$. Then:
   1. Recursively compute BDD $B_x := \texttt{apply}(\odot, B_f^x, B_g^x)$.
   2. Recursively compute BDD $B_{\overline{x}} := \texttt{apply}(\odot, B_f^{\overline{x}}, B_g^{\overline{x}})$.
   3. Create and return a new BDD with root x and B_x as positive sub-BDD and $B_{\overline{x}}$ as negative sub-BDD.

   The justification for this comes directly from

$$f \odot g \quad \Leftrightarrow \quad \overline{x} \cdot (f_{\overline{x}} \odot g_{\overline{x}}) \; + \; x \cdot (f_x \odot g_x)$$

The apply function (continued)

3. v_f is an internal x-node, but v_g is either a leaf (0 or 1) or an internal y-node, with $y > x$, i.e., variable y is *after* x in the ordering (y is lower in the tree). Then we know, since B_f and B_g must follow the same variable ordering, that B_g is independent from x at this point in the tree. So we proceed as follows:
   1. Recursively compute BDD $B_x := \texttt{apply}(\odot, B_f^x, B_g)$.
   2. Recursively compute BDD $B_{\overline{x}} := \texttt{apply}(\odot, B_f^{\overline{x}}, B_g)$.
   3. Create and return a new BDD with root x and B_x as positive sub-BDD and $B_{\overline{x}}$ as negative sub-BDD.

   Do you see room for optimization here?

   E.g., when $\odot$ is $+$ and v_g is 0 or 1. If 0, return v_f. If 1, return 1.
4. Symmetric to case 3 above, but with v_g being higher in the tree than v_f instead of lower.

The apply function: example

Let's try apply(+) on the two BDDs below:

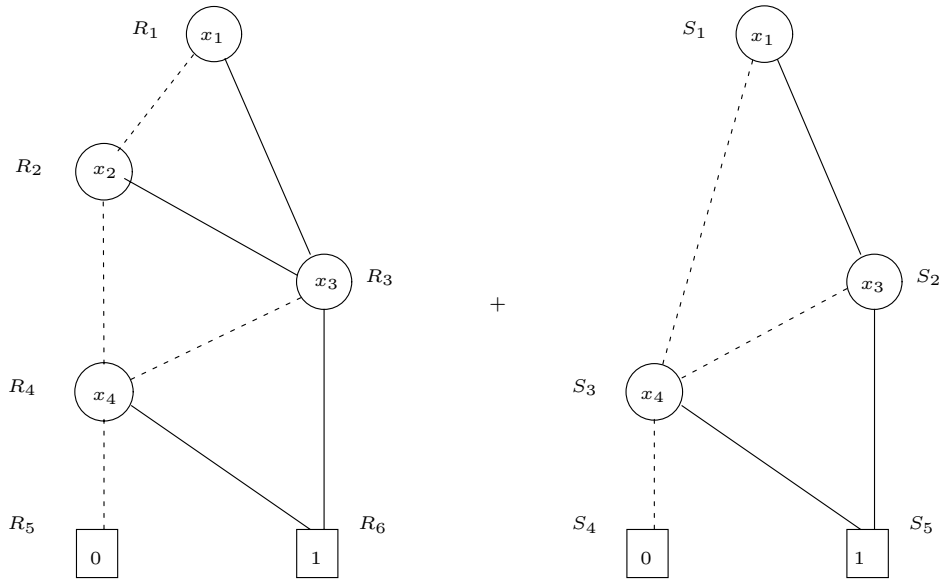


Figure taken from [Huth and Ryan, 2004].

Existential quantifier elimination

Recall that if x is a boolean variable then:

$$\exists x : f \quad \Leftrightarrow \quad f[x \rightsquigarrow 0] \vee f[x \rightsquigarrow 1] \quad \Leftrightarrow \quad f_{\overline{x}} \vee f_x$$

Let B_f be the BDD for f. How to compute the BDD for $\exists x : f$?

We know how to compute disjunction of BDDs already. It suffices to be able to compute substitutions like $f[x \rightsquigarrow 0]$.

This is simple:

- For every x-node v in B_f, eliminate v and redirect all incoming links to the 0-child of v.
- (If we wanted $f[x \rightsquigarrow 1]$ instead, we would redirect them to the 1-child of v.)
- We must then reduce the resulting BDD.

Putting it all together

Recall: Symbolic Reachability Analysis Algorithm

1: $Reachable := Init$;
2: terminate := false;
3: **repeat**
4: $\quad tmp := Reachable \vee \mathbf{succ}(Reachable)$;
5: $\quad$ **if** $tmp \Leftrightarrow Reachable$ **then**
6: $\quad\quad$ terminate := true;
7: $\quad$ **else**
8: $\quad\quad Reachable := tmp$;
9: $\quad$ **end if**
10: **until** terminate
11: return $Reachable$;

where

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

We have all the ingredients to implement this algorithm using BDDs:

- $Init, Reachable, tmp$ are each represented as a BDD on state variables $\vec{x}$.
- $Trans$ is represented as another BDD on $\vec{x}, \vec{x}'$.
- We know how to compute $\wedge, \vee, \exists$ on BDDs.
- Renaming variables $[\vec{x}' \rightsquigarrow \vec{x}]$ is straightforward also.
- We know how to check $\Leftrightarrow$ on BDDs.

FINITE-HORIZON REACHABILITY

(a.k.a. BOUNDED MODEL-CHECKING)

Bounded reachability

Question:

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

i.e., given a transition system (P, S, S_0, L, R) and a set of states $Bad \subseteq S$, does there exist a path

$$s_0 \longrightarrow s_1 \longrightarrow \cdots \longrightarrow s_k$$

in the transition system such that $s_0 \in S_0$ and $s_k \in Bad$, and $k \leq n$.

Key idea:

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

- SAT problem NP-complete for propositional logic.
- In practice, today's SAT solvers can handle formulas with thousands of variables (or more!): see [Malik and Zhang, 2009].
- BMC (**bounded model-checking**) has emerged thanks to the advances in SAT solver technology.

Bounded reachability

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

How to use them for bounded reachability?

- Bad state reachable in 0 steps iff

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

- Bad state reachable in 1 step iff

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

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

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

Bounded reachability algorithm – outer loop

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

Bounded reachability: soundness and completeness

1: **for all** $k = 0, 1, ..., n$ **do**
2: $\quad \phi := Init(\vec{x}_0) \wedge Trans(\vec{x}_0, \vec{x}_1) \wedge \cdots \wedge Trans(\vec{x}_{k-1}, \vec{x}_k) \wedge Bad(\vec{x}_k)$;
3: $\quad$ **if** SAT(ϕ) **then**
4: $\quad\quad$ print "Bad state reachable in k steps";
5: $\quad\quad$ output solution as counter-example;
6: $\quad$ **end if**
7: **end for**
8: 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 up to n steps" then there is no path of length $\leq n$ that reaches a bad state.

Can we make BMC **complete**?

- It should report unreachable iff there are no reachable bad states (w.r.t. any bound).
- Is this even possible in general? For finite-state systems? Yes!

Complete BMC: "brute-force" threshold

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

A finite-state transition system is essentially a finite graph.

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

If we know $|S|$ (the number of all possible states) then we can set $n := |S|$.
Because no acyclic path can have length greater than $|S|$, and we only care about acyclic paths.

But: with 100 boolean variables, $|S| = 2^{100}$, so this isn't practical ... (formulas become too big).

Complete BMC: a better threshold

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

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

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 |Reach|, therefore how to compute d?

Complete BMC: the Completeness Threshold

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

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

Claim: $d \leq r$. Why?

$\Rightarrow$ using r instead of d is safe. Why?

Can we compute r? How?

Use a SAT solver!

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

Bibliography I


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


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


Bryant, R. E. (1992).
Symbolic boolean manipulation with ordered binary-decision diagrams.
ACM Comput. Surv., 24(3):293–318.


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., Emerson, E., Jha, S., and Sistla, A. (1998).
Symmetry reductions in model checking.
In *CAV'98*, pages 147–158. Springer.


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


Courcoubetis, C., Vardi, M., Wolper, P., and Yannakakis, M. (1992).
Memory efficient algorithms for the verification of temporal properties.
Formal Methods in System Design, 1:275–288.

Bibliography II


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.


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


Ip, C. and Dill, D. (1996).
Better verification through symmetry.
Formal Methods in System Design, 9(1-2):41–75.


Latvala, T., Biere, A., Heljanko, K., and Junttila, T. (2004).
Simple Bounded LTL Model Checking.
In *Formal Methods in Computer-Aided Design*, volume 3312 of *LNCS*, pages 186–200. Springer.


Malik, S. and Zhang, L. (2009).
Boolean satisfiability: From theoretical hardness to practical success.
Communications of the ACM, 52(8):76–82.


Sistla, A. P. and Godefroid, P. (2004).
Symmetry and reduced symmetry in model checking.
ACM Trans. Program. Lang. Syst., 26(4):702–734.

Bibliography III


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