Formal Methods for System Design

Chapter 5: Symbolic model checking

Mickael Randour

Mathematics Department, UMONS

October 2021

UMONS
Université de Mons

The state(-space) explosion problem (excerpt from Ch. 2)

Verification techniques operate on TSs obtained from programs or program graphs. Their size can be **huge**, or they can even be **infinite**. Some sources:

- **Variables**
  - ▷ PG with 10 locations, three Boolean variables and five integers in $\{0, \ldots, 9\}$ already contains $10 \cdot 2^3 \cdot 10^5 = 8.000.000$ states.
  - ▷ Variable in infinite domain $\Rightarrow$ infinite TS!
- **Parallelism**
  - ▷ $\mathcal{T} = \mathcal{T}_1 \,|||\, \ldots \,|||\, \mathcal{T}_n \;\Rightarrow\; |S| = |S_1| \cdot \ldots \cdot |S_n|$.
    - $\hookrightarrow$ **Exponential blow-up!**

$\Rightarrow$ Need for (a lot of) **abstraction** and efficient **symbolic** techniques (Ch. 5) to keep the verification process tractable.

$\Longrightarrow$ **Well, here we are!**

What is *symbolic* model checking?

- Classical techniques rely on an **explicit**, enumerative, representation of TSs.
  - ▷ Each individual state is explicitly represented as well as its successor/predecessor lists.

$\Longrightarrow$ **Not adequate for very large TSs!**

- **Idea:** represent TSs in a **symbolic** way by considering *sets* of states and *sets* of transitions instead of individual ones.

$\Longrightarrow$ **Leads to more efficient techniques in practice.**

Symbolic CTL model checking

- There exist various symbolic approaches for different models and logics.
- In this chapter, we focus on a single one: CTL model checking via ROBDDs.

Our goal is to illustrate the concept and interest of such an approach without delving too deep into technical considerations.

- The symbolic set-based approach is **natural for CTL** as its semantics and model checking algorithm are based on *satisfactions sets* for subformulae.

$\Longrightarrow$ **We focus on a technique based on switching functions and ROBDDs (other techniques exist).**

Key concept: binary encoding of states

Let $\mathcal{T} = (S, \longrightarrow, I, AP, L)$ be a TS.

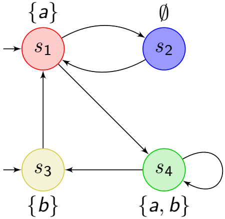
- ▷ We dropped Act as actions are irrelevant from now on.
- ▷ That is, $\longrightarrow \subseteq S \times S$.

We want to encode states as **bit vectors**.

- ▷ We assume that $|S| \geq 2$ and let $n \geq \lceil \log |S| \rceil$.
  - ↪ $n = \#$ bits used to represent S.
- ▷ Let $enc\colon S \rightarrow \{0, 1\}^n$ be an arbitrary *injective* encoding of states by bit vectors of length n.
  - ▷ We can make it *surjective* too w.l.o.g. by using dummy states in the TS (i.e., we now assume $n = \log |S|$).

Binary encoding of states

Example



- $|S| = 4 \implies n = \log|S| = 2$.
- We choose an encoding $enc \colon S \rightarrow \{0,1\}^2$:

$$enc(s_1) = 00 \qquad enc(s_2) = 01$$
$$enc(s_3) = 10 \qquad enc(s_4) = 11$$

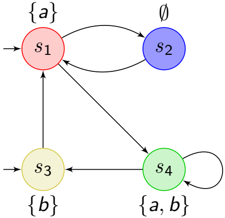
Characteristic function of a set of states

Any subset $T \subseteq S$ can be represented by its **characteristic function** $\chi_T \colon \{0,1\}^n \rightarrow \{0,1\}$ such that $\chi_T(\overline{b})$ evaluates to true (i.e., 1) iff the bit vector $\overline{b} \in \{0,1\}^n$ encodes a state $s \in T$.

E.g., $\chi_I(\overline{b}) = \begin{cases} 1 \text{ if } \overline{b} = 00 \ \vee\ \overline{b} = 10 \\ 0 \text{ otherwise} \end{cases}$ $\qquad \chi_{Sat(a \wedge b)}(\overline{b}) = \begin{cases} 1 \text{ if } \overline{b} = 11 \\ 0 \text{ otherwise} \end{cases}$

Binary encoding of transitions

Example



- $|S| = 4 \implies n = \log |S| = 2$.
- We choose an encoding $enc\colon S \to \{0,1\}^2$:

$$enc(s_1) = 00 \qquad enc(s_2) = 01$$
$$enc(s_3) = 10 \qquad enc(s_4) = 11$$

Same idea for *transitions*: $\longrightarrow \subseteq S \times S$ is represented by a Boolean function $\Delta\colon \{0,1\}^{2n} \to \{0,1\}$ assigning 1 to pairs of bit vectors $(\overline{b}, \overline{b}')$ such that $s = enc^{-1}(\overline{b})$, $s' = enc^{-1}(\overline{b}')$ and $s \to s'$.

In the following, we discuss how this encoding can be seen as switching functions and how CTL model checking can be formulated on them. Then we present compact representation of switching functions through ROBDDs.

Switching functions

Boolean variables and evaluations

Idea

Instead of considering functions $\{0,1\}^n \to \{0,1\}$, we will see bit vectors in $\{0,1\}^n$ as evaluations of Boolean variables and consider mappings from those evaluations to 0 or 1.

Let $Var = \{z_1, \ldots, z_m\}$ be a set of Boolean variables and $Eval(z_1, \ldots, z_m)$ be the set of *evaluations* for those variables, i.e., functions $\eta\colon (z_1, \ldots, z_m) \to \{0,1\}^m$. An evaluation is written as $[z_1 = b_1, \ldots, z_m = b_m]$ or, shortly, $[\overline{z} = \overline{b}]$ for $\overline{b} \in \{0,1\}^m$.

(Notice we use m instead of n here as FTM we forget about TSs and define those notions in full generality.)

Switching functions

Definition

Definition: switching function

A switching function for $\mathit{Var} = \{z_1, \ldots, z_m\}$ is a function $f\colon \mathit{Eval}(\mathit{Var}) \rightarrow \{0, 1\}$. For $m = 0$ (i.e., $\mathit{Var} = \emptyset$), possible switching functions are constants 0 and 1.

We often write $f(\overline{b})$ instead of $f([\overline{z} = \overline{b}])$ when context is clear.

$\Longrightarrow$ **Boolean connectives for switching functions are defined naturally.**

E.g., let f_1 and f_2 be switching functions for $\{z_1, \ldots, z_n, \ldots, z_m\}$ and $\{z_n, \ldots, z_m, \ldots, z_k\}$ respectively. Then, $f_1 \vee f_2$ is a switching function for $\{z_1, \ldots, z_k\}$ whose values are given by

$$(f_1 \vee f_2)([z_1 = b_1, \ldots, z_k = b_k]) = \max\Big\{f_1([z_1 = b_1, \ldots, z_m = b_m]), f_2([z_n = b_n, \ldots, z_k = b_k])\Big\}.$$

Switching functions

As Boolean connections of variables

Observation

Any switching function f for $Var = \{z_1, \ldots, z_m\}$ can be represented as a Boolean connection of the variables z_i (viewed as projection switching functions) and constants 0 and 1.

E.g., $z_1 \vee (z_2 \wedge z_3)$ is a switching function for $Var = \{z_1, z_2, z_3\}$.

Switching functions

Cofactors and essential variables (1/2)

Definition: cofactor

Let f be a switching function for $Var = \{z, y_1, \dots y_m\}$. The *positive cofactor* of f for variable z is the switching function $f|_{z=1} \colon Eval(Var) \rightarrow \{0, 1\}$ whose value is given by

$$f|_{z=1}(c, b_1, \dots, b_m) = f(1, b_1, \dots, b_m)$$

for any bit vector $(c, b_1, \dots, b_m) \in \{0, 1\}^{m+1}$.

Negative cofactors are defined similarly with 0 instead of 1.

Iterated cofactors are obtained by successive replacements and denoted $f|_{z_1=b_1,\dots,z_k=b_k}$.

Definition: essential variable

Variable z is *essential* for f iff $f|_{z=0} \neq f|_{z=1}$.

Switching functions

Cofactors and essential variables (2/2)

Example 1: let $f(z_1, z_2, z_3) = (z_1 \vee \neg z_2) \wedge z_3$.

- ▷ $f|_{z_1=1} = z_3$ and $f|_{z_1=0} = \neg z_2 \wedge z_3$.
- ↪ z_1 is essential for f.

Example 2: let $f(z_1, z_2, z_3) = z_1$ (projection function).

- ↪ z_1 is essential for f whereas z_2 and z_3 are not.

Example 3: let $f(z_1, z_2, z_3) = z_1 \vee \neg z_2 \vee (z_1 \wedge z_2 \wedge \neg z_3)$.

- ↪ z_1 and z_2 are essential.
- ↪ z_3 is not because $f|_{z_3=1} = z_1 \vee \neg z_2$ and $f|_{z_3=0} = z_1 \vee \neg z_2 \vee (z_1 \wedge z_2) = z_1 \vee \neg z_2$.

Switching functions

Decomposition into cofactors

Shannon expansion

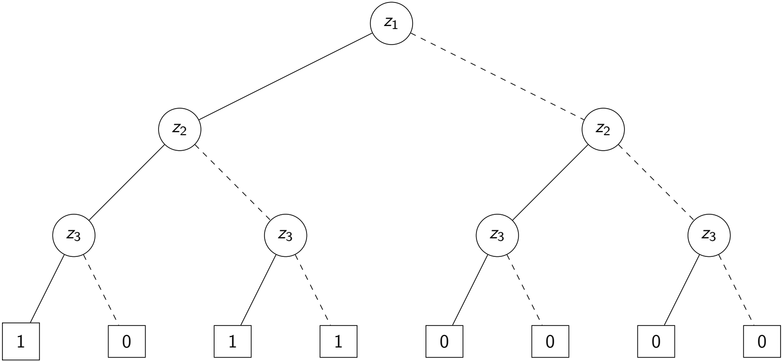
Let f be a switching function for Var. Then, for each $z \in Var$,

$$f = (\neg z \wedge f|_{z=0}) \vee (z \wedge f|_{z=1}).$$

$\Longrightarrow$ This decomposition is the cornerstone of the representation of switching functions as **binary decision trees**.

Switching functions

Binary decision trees



Binary decision tree for $f = z_1 \wedge (\neg z_2 \vee z_3)$. Solid edge leaving z_i means $z_i = 1$, dashed one means $z_i = 0$. Path from root to leaf represents an evaluation and its value for f. Subtrees = cofactors.

Switching functions

Quantification over variables

Existential and universal quantification

Let f be a switching function for Var and $z \in Var$. Then, $\exists z.f$ is the switching function $\exists z.f = f|_{z=0} \vee f|_{z=1}$ and $\forall z.f$ is the one defined by $\forall z.f = f|_{z=0} \wedge f|_{z=1}$.

Example: let $f = (z \vee y_1) \wedge (\neg z \vee y_2)$.

- ▷ Then $\exists z.f = f|_{z=0} \vee f|_{z=1} = y_1 \vee y_2$,
- ▷ and $\forall z.f = f|_{z=0} \wedge f|_{z=1} = y_1 \wedge y_2$.

Switching functions

Renaming

Let $\overline{z} = (z_1, \ldots, z_m)$, $\overline{y} = (y_1, \ldots, y_m)$, and $\overline{x} = (x_1, \ldots, x_k)$ such that $\overline{x}$ does not contain any z_i or y_i. Let $s = [\overline{y} = \overline{b}, \overline{x} = \overline{c}] \in \mathit{Eval}(\overline{y}, \overline{x})$ be an evaluation.

Then, $s\{\overline{z} \leftarrow \overline{y}\}$ denotes the evaluation in $\mathit{Eval}(\overline{z}, \overline{x})$ obtained through the renaming function $y_i \mapsto z_i$. That is, $s\{\overline{z} \leftarrow \overline{y}\}$ agrees with s for x_i and assigns to z_i the same value as s assigns to y_i.

Given $f\colon \mathit{Eval}(\overline{y}, \overline{x}) \to \{0, 1\}$, then the switching function $f\{\overline{z} \leftarrow \overline{y}\} : \mathit{Eval}(\overline{z}, \overline{x}) \to \{0, 1\}$ is given by $f\{\overline{z} \leftarrow \overline{y}\}(s) = f(s\{\overline{z} \leftarrow \overline{y}\})$. We simply write $f(\overline{z}, \overline{x})$ if the context is clear.

Encoding TSs by switching functions

Labeling

Back to TSs: we now use switching functions to give a symbolic representation of a TS $\mathcal{T} = (S, \longrightarrow, I, AP, L)$.

- We use $n = \log |S|$ Boolean variables $x_1, \ldots, x_n$ to represent S: we identify any evaluation $[\overline{x} = \overline{b}] \in Eval(\overline{x})$ with the unique state $s \in S$ such that $enc(s) = \overline{b}$.
  - $\hookrightarrow$ We assume $S = Eval(\overline{x})$.
- Recall that any set $T \subseteq S$ can be described by a characteristic function χ_T. Actually, χ_T is the switching function

$$\chi_T \colon Eval(\overline{x}) \rightarrow \{0, 1\},\ \chi_T(s) = \begin{cases} 1 \text{ if } s \in T \\ 0 \text{ otherwise.} \end{cases}$$

$\Longrightarrow$ **The labeling function can be represented by a family $(f_a)_{a \in AP}$ of switching functions for $\overline{x}$ such that $f_a = \chi_{Sat(a)}$.**

Encoding TSs by switching functions

Transitions

Same idea: we identify $\longrightarrow$ with its characteristic function $S \times S \to \{0, 1\}$ assigning 1 to (s, s') iff $s \to s'$.

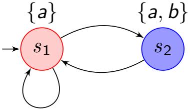
- ▷ We use *two variable tuples*, $\overline{x}$ and $\overline{x}'$, to encode s and s'.
- ▷ We encode $\longrightarrow$ as the switching function

$$\Delta\colon \mathit{Eval}(\overline{x}, \overline{x}') \to \{0, 1\},\ \Delta(s, s'\{\overline{x}' \leftarrow \overline{x}\}) = \begin{cases} 1 \text{ if } s \to s' \\ 0 \text{ otherwise} \end{cases}$$

$\Longrightarrow$ Here we see that the renaming is used because formally s' is an evaluation for $\overline{x}$ and we need to map it to $\overline{x}'$.

Encoding TSs by switching functions

Example



Need a single Boolean variable x for the encoding: $enc(s_1) = 0$, $enc(s_2) = 1$.

Labeling?

- ▷ $f_a = \chi_{Sat(a)} = \chi_S = \neg x \vee x = 1$.
- ▷ $f_b = \chi_{Sat(b)} = \chi_{\{s_2\}} = x$.

Transitions?

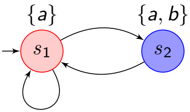
- ▷ $\Delta = (\neg x \wedge \neg x') \vee (\neg x \wedge x') \vee (x \wedge \neg x') = \neg x \vee \neg x'$.

$\Longrightarrow$ **Such a symbolic representation can be obtained for any TS.**

$\Longrightarrow$ **Exercise: construct the symbolic representation of the TS in slide 8.**

Encoding TSs by switching functions

Example (contd.)



Need a single Boolean variable x for the encoding: $enc(s_1) = 0$, $enc(s_2) = 1$.

Transitions:

- ▷ $\Delta = \neg x \vee \neg x'$.

Successors?

- ▷ $Post(s_1) = \{s_1, s_2\}$ obtained symbolically by

$$\Delta|_{x=0}\{x' \leftarrow x\} = (\neg x \vee \neg x')|_{x=0}\{x' \leftarrow x\} = 1.$$

↪ Constant 1 means all states of S belong to the Post.

- ▷ $Post(s_2) = \{s_1\}$ obtained symbolically by

$$\Delta|_{x=1}\{x' \leftarrow x\} = (\neg x \vee \neg x')|_{x=1}\{x' \leftarrow x\} = \neg x.$$

↪ Only s_1 belongs to the Post (because it is encoded as 0, hence $\neg 0 = 1$, whereas s_2 is 1 hence $\neg 1 = 0$).

CTL model checking using switching functions

Backward reachability

Consider a TS represented by the switching function $\Delta(\overline{x}, \overline{x}')$ and a set $B \subseteq S$ given by its characteristic function χ_B. We want to compute $Pre^*(B) = \{s \in S \mid s \models \exists\Diamond B\}$ using switching functions.

- $f_0 = \chi_B$ represents $T_0 = B$.
- We compute $f_{j+1} = \chi_{T_{j+1}}$ for $T_{j+1} = T_j \cup \{s \in S \mid \exists s' \in S,\ s' \in Post(s) \wedge s' \in T_j\}$ as

$$f_{j+1} = f_j \vee \exists \overline{x}'.\big(\underbrace{\Delta(\overline{x}, \overline{x}')}_{s' \in Post(s)} \wedge \underbrace{f_j(\overline{x}')}_{s' \in T_j}\big)$$

$\Longrightarrow$ **Blackboard illustration on previous example for s_2.**

CTL model checking using switching functions

Adaptation to $\exists(C \cup B)$

Input: $\Delta(\overline{x}, \overline{x}')$, χ_B and χ_C
Output: $f_j(\overline{x})$ representing $Sat(\exists(C \cup B))$

$f_0(\overline{x}) := \chi_B(\overline{x})$
$j := 0$
repeat
  $f_{j+1}(\overline{x}) := f_j(\overline{x}) \vee \Big(\chi_C(\overline{x}) \wedge \exists \overline{x}'.\big(\Delta(\overline{x}, \overline{x}') \wedge f_j(\overline{x}')\big)\Big)$
  $j := j + 1$
until $f_j(\overline{x}) = f_{j-1}(\overline{x})$
return $f_j(\overline{x})$

$\hookrightarrow$ The additional conjunction ensures that we only add states from C.

CTL model checking using switching functions

Adaptation to $\exists\bigcirc B$

Recall that $Sat(\exists\bigcirc B) = \{s \in S \mid Post(s) \cap B \neq \emptyset\}$.

$\hookrightarrow$ **Here no iteration is needed:**

$$\chi_{Sat(\exists\bigcirc B)}(\overline{x}) = \exists \overline{x}'.\big(\Delta(\overline{x}, \overline{x}') \wedge \chi_B(\overline{x}')\big).$$

CTL model checking using switching functions

Adaptation to $\exists\Box B$

We need to mimic $T_0 = B$ and $T_{j+1} = T_j \cap \{s \in S \mid \exists s' \in S,\ s' \in Post(s) \wedge s' \in T_j\}$ by a symbolic computation.

Input: $\Delta(\overline{x}, \overline{x}')$ and χ_B
Output: $f_j(\overline{x})$ representing $Sat(\exists\Box B)$

$f_0(\overline{x}) := \chi_B(\overline{x})$
$j := 0$
repeat
  $f_{j+1}(\overline{x}) := f_j(\overline{x}) \wedge \exists \overline{x}'.\big(\Delta(\overline{x}, \overline{x}') \wedge f_j(\overline{x}')\big)$
  $j := j + 1$
until $f_j(\overline{x}) = f_{j-1}(\overline{x})$
return $f_j(\overline{x})$

$\Longrightarrow$ **Blackboard illustration on last example for $\exists\Box b$.**

CTL model checking using switching functions

Wrap-up

We now have **all the necessary ingredients to deal with CTL formulae in ENF symbolically**: algorithms for atomic propositions, conjunctions, negations, $\exists \bigcirc \Phi$, $\exists(\Phi \mathsf{U} \Psi)$ and $\exists \Box \Phi$ based on switching functions.

E.g., $Sat(\Phi \land \neg\Psi)$ is given by $f_\Phi \land \neg f_\Psi$.

At the end of the model checking process, we must check that $I \subseteq Sat(\Phi)$: this can be achieved by checking that $\chi_I \rightarrow f_\Phi$ holds, i.e., that $\neg\chi_I \lor f_\Phi = 1$ in terms of switching functions.

$\Longrightarrow$ **It remains to find appropriate data structures to encode the switching functions!**

The problem

We are looking for an appropriate data structure to encode the switching functions. It needs to:

1. yield **compact representations** of the satisfaction sets and the transition relation,
2. **support Boolean connectives and comparison of switching functions** (to implement the model checking procedures based on switching functions directly).

Regarding compactness, recall that for a system with $|S| = 2^n$ states, we only need n Boolean variables ($2n$ for transitions).

$\Longrightarrow$ Unfortunately, it can be proved that *no* data structure can ensure polynomial-size representations of *all* switching functions (because there are doubly-exponentially-many switching functions for m variables).

Possible data structures

We know that no structure can avoid exponential representation for *some* switching functions.

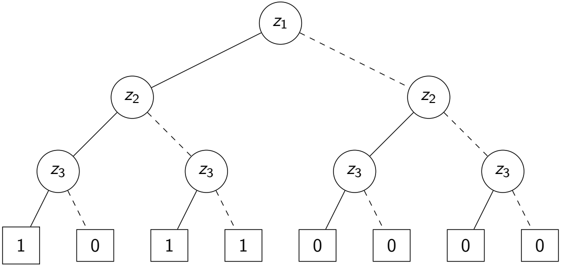
$\Longrightarrow$ **Does not mean that all structures are equally good!**

- *Truth tables* are not efficient as they *always* require 2^m entries for m variables (i.e., whatever the switching function).
- Same for *binary decision trees*: always $2^{m+1} - 1$ nodes.
- *Conjunctive and disjunctive normal forms* are not well-suited because equivalence checking is hard.
- A good choice is **ordered binary decision diagrams (OBDDs)**.
  - Yields compact representations *for many switching functions appearing in practical applications*.
  - Boolean connectives in linear time (in the input OBDDs).
  - Equivalence checking in constant time with appropriate implementation.

OBDDs

Intuition

Main idea: compactification of binary decision trees.

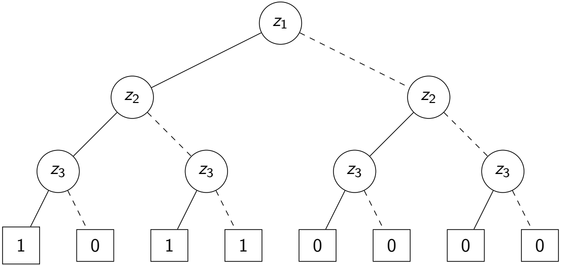


Recall BDT for $f = z_1 \wedge (\neg z_2 \vee z_3)$. Our goal is to *skip redundant fragments of this tree*.

OBDDs

Intuition

Main idea: compactification of binary decision trees.

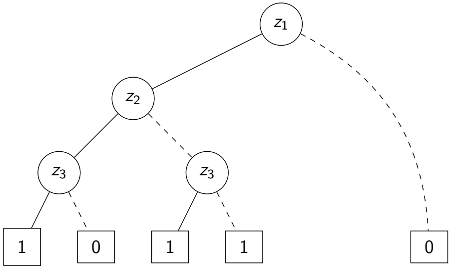


The right subtree corresponds to cofactor $f|_{z_1=0}$, modeling the constant function 0: tests on z_2 and z_3 are useless as all terminal nodes have value 0.

OBDDs

Intuition

Main idea: compactification of binary decision trees.



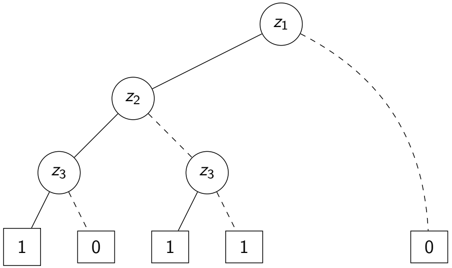
The right subtree corresponds to cofactor $f|_{z_1=0}$, modeling the constant function 0: tests on z_2 and z_3 are useless as all terminal nodes have value 0.

$\Longrightarrow$ We replace this subtree by a single terminal node of value 0.

OBDDs

Intuition

Main idea: compactification of binary decision trees.

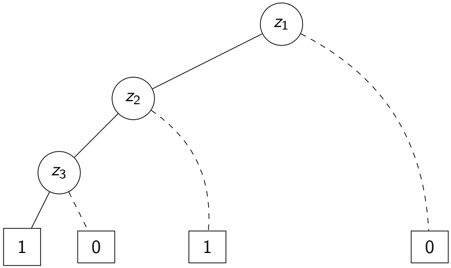


The subtree corresponding to cofactor $f|_{z_1=1,z_2=0}$ is also constant.

OBDDs

Intuition

Main idea: compactification of binary decision trees.



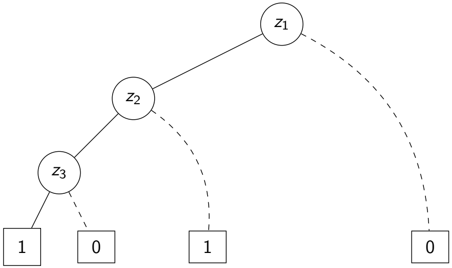
The subtree corresponding to cofactor $f|_{z_1=1,z_2=0}$ is also constant.

$\implies$ We replace this subtree by a single leaf of value 1.

OBDDs

Intuition

Main idea: compactification of binary decision trees.

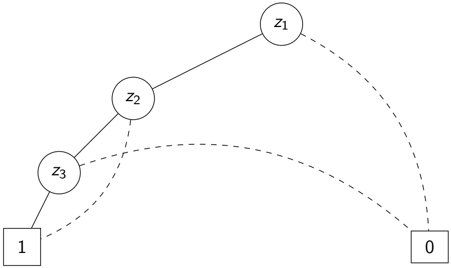


Now, we finally observe that it is useless to keep several leaves with the same value.

OBDDs

Intuition

Main idea: compactification of binary decision trees.



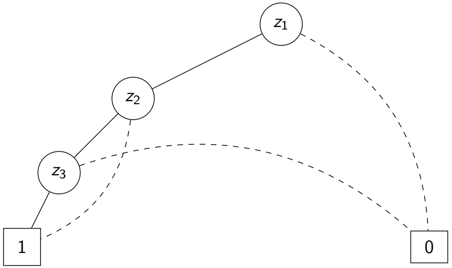
Now, we finally observe that it is useless to keep several leaves with the same value.

$\Longrightarrow$ We only keep one of each.

OBDDs

Intuition

Main idea: compactification of binary decision trees.



At the end, we obtain a *directed acyclic graph (DAG)* where inner nodes have outdegree 2. As in BDTs, inner nodes are labeled with variables and outgoing edges (solid or dashed) correspond to their evaluations; leaves are labeled with the function value.

$\Longrightarrow$ **Much more compact.**

OBDDs

Other example

$\Longrightarrow$ **Blackboard example for $f = (z_1 \wedge z_3) \vee (z_2 \wedge z_3)$.**

OBDDs

Variable ordering

Definition: variable ordering

Let Var be a finite set of variables. A *variable ordering* for Var is any tuple $\wp = (z_1, \ldots, z_m)$ such that $Var = \{z_1, \ldots, z_m\}$ and $z_i \neq z_j$ for $1 \leq i < j \leq m$.

$\hookrightarrow$ It induces a total order and operators $<_\wp$ and $\leq_\wp$:
i.e., $z_i <_\wp z_j$ iff $i < j$.

$\Longrightarrow$ **We will see that different orderings yield different (R)OBDDs!**

OBDDs

Definition (1/2)

Definition: ordered binary decision diagram (OBDD)

Let $\wp$ be a variable ordering for Var. A $\wp$-OBDD is a tuple $\mathfrak{B} = (V, V_I, V_T, succ_0, succ_1, var, val, v_0)$ consisting of

- a finite set of nodes V partitioned into V_I and V_T, i.e., inner nodes and terminal nodes;
- successor functions $succ_0, succ_1 \colon V_I \to V$ assigning 0- and 1-successors to inner nodes;
- a variable labeling function $var \colon V_I \to Var$ assigning a variable to each inner node;
- a value function $val \colon V_T \to \{0, 1\}$ assigning to each terminal node a function value 0 or 1;
- a root node $v_0 \in V$.

OBDDs

Definition (2/2)

The variable labeling function must be **consistent with the ordering**: if *v* is an inner node and *w* is both a successor of *v* and an inner node, then $var(v) < var(w)$ must hold.

↪ Intuitively, branches must respect the variable ordering.

When referring to the *size* of an OBDD, we consider its number of nodes.

⚠ **Observe that the definition of OBDDs does not enforce the reducing operations we have discussed before. In particular, BDTs are valid OBDDs.**

OBDDs

Semantics

As observed intuitively, the semantics of a $\wp$-OBDD $\mathfrak{B}$ is the **switching function** $f_{\mathfrak{B}}$ for Var where $f_{\mathfrak{B}}([z_1 = b_1, \ldots, z_m = b_m])$ is the value of the terminal node reached by following the corresponding branch of $\mathfrak{B}$ from the root v_0.

$\Longrightarrow$ In the following, we will see how to go from OBDDs to Reduced OBDDs (ROBDDs): for that, we need to introduce a few more concepts.

OBDDs

Bottom-up characterization of switching functions for nodes

Let $\mathfrak{B}$ be a $\wp$-OBDD. The switching functions f_v for the nodes $v \in V$ are given as follows:

- if v is a leaf, then f_v is the constant switching function with value $val(v)$;
- if v is a z-node, then $f_v = (\neg z \wedge f_{succ_0(v)}) \vee (z \wedge f_{succ_1(v)})$.

Furthermore, $f_{\mathfrak{B}} = f_{v_0}$ for the root v_0 of $\mathfrak{B}$.

$\Longrightarrow$ **Observe the Shannon expansion!**

$\hookrightarrow$ All concepts of OBDD-based approaches are based on this decomposition into cofactors.

OBDDs

$\wp$-consistent cofactors (1/2)

Definition: $\wp$-consistent cofactor

Let f be a switching function for Var and $\wp = (z_1, \ldots, z_m)$ be an ordering for Var. A switching function f' for Var is a $\wp$-consistent cofactor of f if there exists $i \in \{0, \ldots, m\}$ such that $f' = f|_{z_1=b_1,\ldots,z_i=b_i}$.

E.g., let $f = z_1 \wedge (z_2 \vee \neg z_3)$ and $\wp = (z_1, z_2, z_3)$. Consistent cofactors are:

- f itself;
- $f|_{z_1=0} = 0$ and $f|_{z_1=1} = z_2 \vee \neg z_3$;
- $f|_{z_1=1,z_2=0} = \neg z_3$ and $f|_{z_1=1,z_2=1} = 1$;
- $f|_{z_1=1,z_2=0,z_3=0} = 1$ and $f|_{z_1=1,z_2=0,z_3=1} = 0$ (redundant).

$\Longrightarrow$ $\wp$-consistent cofactors: f, $z_2 \vee \neg z_3$, $\neg z_3$, 0 and 1.

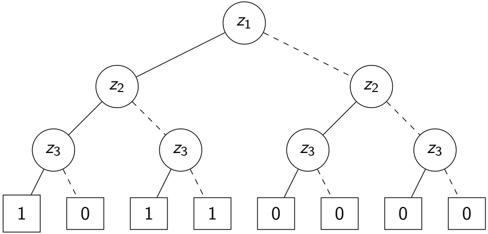
OBDDs

$\wp$-consistent cofactors (2/2)

Observation

For each node v in a $\wp$-OBDD $\mathfrak{B}$, the switching function f_v is a $\wp$-consistent cofactor of $f_{\mathfrak{B}}$; and for each $\wp$-consistent cofactor f' of f, there is *at least* one node v in $\mathfrak{B}$ such that $f_v = f'$.

$\Longrightarrow$ At least one, but **there can be many more**! E.g., BDTs bearing redundant information.



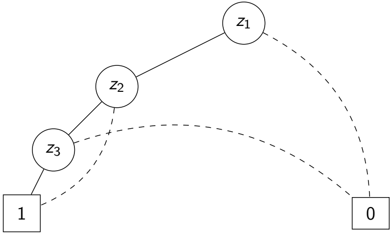
OBDDs

$\wp$-consistent cofactors (2/2)

Observation

For each node v in a $\wp$-OBDD $\mathfrak{B}$, the switching function f_v is a $\wp$-consistent cofactor of $f_{\mathfrak{B}}$; and for each $\wp$-consistent cofactor f' of f, there is *at least* one node v in $\mathfrak{B}$ such that $f_v = f'$.

$\Longrightarrow$ What about the OBDD obtained after "reduction"?



$\Longrightarrow$ **It is free of redundancies: every $\wp$-consistent cofactor is represented by a single node.**

Reduced OBDDs

Definition

Definition: reduced OBDD (ROBDD)

Let $\mathfrak{B}$ be a $\wp$-OBDD. It is said to be *reduced* if for every pair of nodes (v, w) in $\mathfrak{B}$:

$$v \neq w \implies f_v \neq f_w.$$

The fact that each $\wp$-consistent cofactor corresponds to exactly one node of a $\wp$-ROBDD is the crux to obtain the next theorem.

Reduced OBDDs

Universality and canonicity

Theorem: universality and canonicity of ROBDDs

Let Var be a finite set of variables and $\wp$ an ordering for Var. Then:

(*a*) for each switching function f for Var, there exists a $\wp$-ROBDD $\mathfrak{B}$ with $f_{\mathfrak{B}} = f$;

(*b*) given two $\wp$-ROBDDs $\mathfrak{B}$ and $\mathfrak{C}$ with $f_{\mathfrak{B}} = f_{\mathfrak{C}}$, then $\mathfrak{B}$ and $\mathfrak{C}$ are isomorphic, i.e., they agree up to renaming of the nodes.

$\Longrightarrow$ **It is possible to talk about "the $\wp$-ROBDD" for a switching function f (up to isomorphism): this ROBDD is also the minimal $\wp$-OBDD for f.**

Reduced OBDDs

Simple construction procedure based on consistent cofactors

Let f be the switching function for Var to represent and $\wp$ the ordering of the variables.

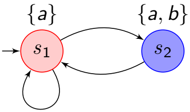
- If f is constant, then the ROBDD $\mathfrak{B}$ contains a single terminal node of the corresponding value. Else we proceed as follows.
- Let V be the set of $\wp$-consistent cofactors of f.
  - ▷ The *root* of $\mathfrak{B}$ is f and the constant cofactors are the *leaves* with corresponding values.
  - ▷ For $f' \in V \setminus \{0, 1\}$, let $var(f') = \min\{z \in Var \mid z$ is essential for $f'\}$ (resp. to the ordering).
  - ▷ Successors functions are $succ_0(f') = f'|_{z=0}$ and $succ_1(f') = f'|_{z=1}$ where $z = var(f')$.

By construction, this yields a $\wp$-OBDD $\mathfrak{B}$, and by Shannon expansion, its semantics is indeed $f_{\mathfrak{B}} = f$ and it is reduced (as each node represents a different cofactor).

Reduced OBDDs

Example

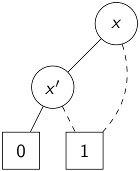
By universality of ROBDDs, any switching function can be encoded: let us encode the transition relation of the previous TS.



Need a single Boolean variable x for the encoding: $enc(s_1) = 0$, $enc(s_2) = 1$.

- ▷ Transitions: $\Delta = \neg x \vee \neg x'$.

$\Longrightarrow$ **Blackboard construction.**



ROBDD $\wp = (x, x')$.

Reduced OBDDs

Consequences of canonicity

The canonicity of ROBDDs yields interesting properties.

- *Absence of redundant vertices*: if $f_{\mathfrak{B}}$ does not depend on x_i, then $\mathfrak{B}$ does not contain an x_i-node.
- *Test for equivalence* between two switching functions $f(\overline{x})$ and $f'(\overline{x})$ can be done by generating ROBDDs $\mathfrak{B}_f$ and $\mathfrak{B}_{f'}$ and checking their isomorphism.
- *Test for validity*: checking if $f(\overline{x}) = 1$ can be done by generating $\mathfrak{B}_f$ and checking that it only consists of a 1-leaf.
- *Test for implication*: checking if $f(\overline{x}) \rightarrow f'(\overline{x})$ by generating $\mathfrak{B}_{f \wedge \neg f'}$ and checking that it only consists of a 0-leaf.
- *Test for satisfiability* of a Boolean expression f by checking that $\mathfrak{B}_f$ contains a reachable 1-leaf.

⚠ The SAT problem is NP-complete! Further proof that ROBDDs cannot ensure polynomial encoding of all switching functions.

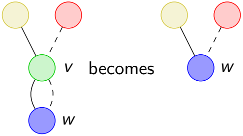
Reduced OBDDs

From OBDDs to ROBDDs (1/2)

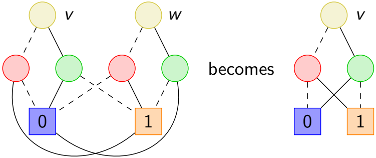
Reduction rules

Any $\wp$-OBDD can be transformed into a canonical $\wp$-ROBDD for the same switching function by successive applications of two simple *local reduction rules*.

1. *Elimination rule*: if $v \in V_I$ is s.t. $succ_0(v) = succ_1(v) = w$, then remove v and redirect all its incoming edges to w.
2. *Isomorphism rule*: if $v \neq w$ are the roots of isomorphic trees, remove w and redirect all its incoming edges to v.



Elimination rule.



Isomorphism rule.

Reduced OBDDs

From OBDDs to ROBDDs (2/2)

This reduction scheme is

- **sound**: if $\mathfrak{C}$ is a $\wp$-OBDD obtained by reduction from $\mathfrak{B}$, then $f_{\mathfrak{C}} = f_{\mathfrak{B}}$;
- **complete**: the $\wp$-OBDD $\mathfrak{B}$ is a $\wp$-ROBDD iff no reduction rule can be applied to $\mathfrak{B}$.

It can be implemented in *linear time* in the size of $\mathfrak{B}$.

Reduced OBDDs

The variable ordering problem

ROBDDs are canonical... **for a fixed variable ordering!**

$\Longrightarrow$ The size of the ROBDD *crucially depends on the ordering* (recall that $|V| = \#$ of $\wp$-consistent cofactors of f).

Some functions have

- *both* linear and exponential ROBDDs depending on the chosen ordering: e.g., the *stable function*;

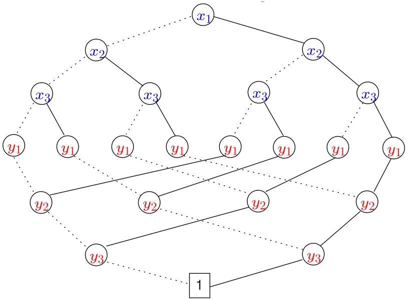
  $\hookrightarrow$ See next slide.

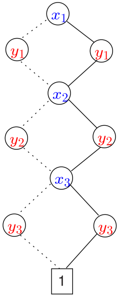
- *only* polynomial ROBDDs: e.g., *symmetric functions* such as $f(\overline{x}) = x_1 \oplus \ldots \oplus x_n$ or $f(\overline{x}) = 1$ iff $\geq k$ variables x_i are true;
- *only* exponential ROBDDs: e.g., the middle bit of the multiplication function.

  $\hookrightarrow$ See book.

Reduced OBDDs

The stable function: exponential vs. linear ROBDDs





Example from [Kat10]: $f_{stab}(\overline{x}, \overline{y}) = (x_1 \leftrightarrow y_1) \wedge \ldots \wedge (x_n \leftrightarrow y_n)$.

- ▷ $\wp = (x_1, \ldots, x_n, y_1, \ldots, y_n)$ yields $\mathcal{O}(2^n)$ nodes.
- ▷ $\wp = (x_1, y_1, \ldots, x_n, y_n)$ yields $\mathcal{O}(n)$ nodes.

$\Longrightarrow$ Intuitively, the second ordering checks each conjunct sequentially whereas the first one needs to recall the values of all variables x_i before being able to check the first conjunct.

Reduced OBDDs

Finding a good ordering

- The size of ROBDDs drastically depends on the ordering.
- Can we determine which is the best ordering, i.e., yielding the minimal ROBDD?
  - ▷ **Not efficiently**: checking if a variable ordering is optimal is NP-hard.

$\Longrightarrow$ In practice, efficient heuristics are used to improve the current ordering and rearrange the ROBDD. Beyond the scope of this course.

$\Longrightarrow$ For transition relations, the **interleaved ordering** usually yields compact ROBDDs:
for $\Delta(\overline{x}, \overline{x}')$, use $\wp = (x_1, x_1', \ldots, x_n, x_n')$.

Back to CTL model checking

We already established

1. a symbolic CTL model checking procedure based on *switching functions*,
2. that switching functions can be represented by *ROBDDs* which, for practical cases, are often compact.

The missing piece is how to *actually* implement the model checking blocks based on ROBDD-representations of the switching functions.

$\Longrightarrow$ We need to be able to **synthesize an ROBDD for a switching function** f (as sketched before), but also to **implement Boolean connectives at the ROBDD level**.
I.e., given $\wp$-ROBDDs for f_1 and f_2, we must be able to build a $\wp$-ROBDD for f_1 *op* f_2 where *op* is a Boolean connective (conjunction, implication, etc).

Synthesis of ROBDDs

We concentrate on the problem of synthesizing $\mathfrak{B}_{f_1 \; op \; f_2}$ from $\mathfrak{B}_{f_1}$ and $\mathfrak{B}_{f_2}$.

$\Longrightarrow$ We do not address the problem in full details but sketch the key steps of an approach based on **shared OBDDs**.

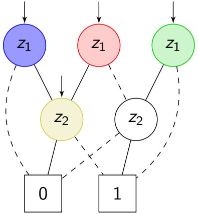
- ▷ The idea is to use a *single* ROBDD with global variable ordering $\wp$ to represent *several* switching functions.
- ▷ The shared OBDD can be seen as the combination of several ROBDDs obtained by *sharing nodes for common $\wp$-consistent cofactors*.

$\Longrightarrow$ **Increased compactness**: in the worst case, the shared OBDD will have its size bounded by the sum of sizes for all combined ROBDDs, but in practice it is often much more compact as shared cofactors are common.

Shared OBDDs

Definition: shared OBDD (SOBDD)

A $\wp$-SOBDD is simply a $\wp$-ROBDD with *possibly multiple roots* instead of a single one.



SOBDD representing $f_1 = z_1 \wedge \neg z_2$, $f_2 = \neg z_2$, $f_3 = z_1 \oplus z_2$ *and* $f_4 = \neg z_1 \vee z_2$ *for ordering* $\wp = (z_1, z_2)$.

Using SOBDDs for model checking a CTL formula Φ

Sketch (1/2)

We use a *single* SOBDD to encode:

- $\Delta(\overline{x}, \overline{x}')$ for the transition function,
- functions $(f_a)_{a \in AP}$ for the satisfaction sets $Sat(a)$ where $a \in AP$,
- the satisfaction sets $Sat(\Psi)$ for the subformulae Ψ of Φ.

In practice, the *interleaved ordering* gives good results.

Using SOBDDs for model checking a CTL formula Φ

Sketch (2/2)

Model checking process:

1. At start, we synthesize an SOBDD representing Δ and functions $(f_a)_{a \in AP}$.
2. During the procedure, we *extend it with new root nodes* for characteristic functions $\chi_{Sat(\Psi)}$ for subformulae Ψ of Φ.
   - E.g., if $\Phi = a \wedge \neg b$, then we first have to insert a root for $f_{\neg b} = \neg f_b$, and then a root for $f_a \wedge f_{\neg b}$.
   - For formulae like $\exists\Box\Psi$, we need to compute a sequence of iterations f_i, and each of them must be added to the SOBDD.
   - Each root addition *may induce the addition of consistent cofactors* not already present in the SOBDD.

Operations on the SOBDD are interleaved with reduction rules to ensure redundance-freedom at any time, making the comparison between two functions easy (it boils down to node equality).

Synthesizing SOBDDs

Two tables: *unique* and *computed*

The synthesis process relies on **two tables** for its computations.

- The **unique** table.
  - ▷ Keeps track of created nodes.
  - ▷ Each inner node v has an entry $\langle var(v), succ_1(v), succ_0(v)\rangle$.
  - ▷ Access via `find_or_add`(z, v_1, v_0) with $v_1 \neq v_0$:
    - returns v if there is a node $v = \langle z, v_1, v_0\rangle$ in the SOBDD,

      $\Longrightarrow$ Isomorphism reduction rule.
    - if not, creates a new z-node v with $succ_1(v) = v_1$ and $succ_0(v) = v_0$.
  - ▷ Implemented via *hash functions* (expected access in $\mathcal{O}(1)$).
- The **computed** table.
  - ▷ Keeps track of already computed tuples for upcoming function ITE (*memoization*).
  - ▷ Avoids redundant computations.

Synthesizing SOBDDs

The ITE operator (1/3)

We deal with *all* Boolean connectives through a *single* ternary operator, ITE (if-then-else):

$$\mathtt{ITE}(g, f_1, f_2) = (g \wedge f_1) \vee (\neg g \wedge f_2).$$

$\Longrightarrow$ If g then f_1 else f_2.

Link with the *unique* table representation of node v:

$$f_v = \mathtt{ITE}(z, f_{succ_1(v)}, f_{succ_0(v)}).$$

$\Longrightarrow$ Again the Shannon expansion!

Synthesizing SOBDDs

The ITE operator (2/3)

The $\mathtt{ITE}(g, f_1, f_2) = (g \wedge f_1) \vee (\neg g \wedge f_2)$ operator can give us:

$$f_1 = \mathtt{ITE}(1, f_1, f_2) \qquad f_2 = \mathtt{ITE}(0, f_1, f_2) \qquad \neg f = \mathtt{ITE}(f, 0, 1)$$

$$f_1 \vee f_2 = \mathtt{ITE}(f_1, 1, f_2) \quad f_1 \wedge f_2 = \mathtt{ITE}(f_1, f_2, 0) \quad f_1 \rightarrow f_2 = \mathtt{ITE}(f_1, f_2, 1)$$

$$f_1 \oplus f_2 = \mathtt{ITE}(f_1, \neg f_2, f_2) = \mathtt{ITE}(f_1, \mathtt{ITE}(f_2, 0, 1), f_2).$$

Key observation

Let g, f_1, f_2 be switching functions for Var, $z \in Var$ and $b \in \{0, 1\}$. Then:

$$\mathtt{ITE}(g, f_1, f_2)|_{z=b} = \mathtt{ITE}(g|_{z=b}, f_1|_{z=b}, f_2|_{z=b}).$$

$\Longrightarrow$ This observation is at the basis of the upcoming algorithm.

Synthesizing SOBDDs

The ITE operator (3/3)

From this observation, it follows that a node representing $\texttt{ITE}(g, f_1, f_2)$ is a node $w = \langle z, w_1, w_0 \rangle$ where

- z is the $\wp$-minimal essential variable of $\texttt{ITE}(g, f_1, f_2)$,
- w_b is a node with $f_{w_b} = \texttt{ITE}(g|_{z=b}, f_1|_{z=b}, f_2|_{z=b})$.

This suggests a **recursive algorithm**:

- determine z,
- recursively compute the nodes for ITE applied to the cofactors of g, f_1 and f_2.

Synthesizing SOBDDs

Basic algorithm for $\texttt{ITE}(u, v_1, v_2)$

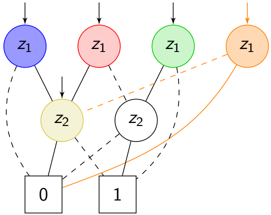
Input: u, v_1 and v_2 three $\wp$-SOBDD nodes
Output: w the $\wp$-SOBDD node whose subtree represents $f_w = \texttt{ITE}(u, v_1, v_2)$

if u is terminal **then**
  if $val(u) = 1$ **then**
    $w := v_1$ $\{\texttt{ITE}(1, f_{v_1}, f_{v_2}) = f_{v_1}\}$
  else
    $w := v_2$ $\{\texttt{ITE}(0, f_{v_1}, f_{v_2}) = f_{v_2}\}$
else
  $z := \min\{var(u), var(v_1), var(v_2)\}$ {z is the minimal essential variable}
  $w_1 := \texttt{ITE}(u|_{z=1}, v_1|_{z=1}, v_2|_{z=1})$
  $w_0 := \texttt{ITE}(u|_{z=0}, v_1|_{z=0}, v_2|_{z=0})$
  if $w_0 = w_1$ **then**
    $w := w_1$ {elimination rule}
  else
    $w := \texttt{find\_or\_add}(z, w_1, w_0)$ {isomorphism rule}
return w

$\Longrightarrow$ **Blackboard illustration on example from slide 52.**

Synthesizing SOBDDs

Basic algorithm for $\texttt{ITE}(u, v_1, v_2)$: illustration



SOBDD representing $f_1 = z_1 \wedge \neg z_2$, $f_2 = \neg z_2$, $f_3 = z_1 \oplus z_2$ and $f_4 = \neg z_1 \vee z_2$ for ordering $\wp = (z_1, z_2)$.

We can compute $f_2 \wedge f_4 = \neg z_2 \wedge \neg z_1$ by applying $\texttt{ITE}(f_2, f_4, 0)$: the result is the added orange part on the SOBDD. We can easily check that it indeed represents the desired switching function.

Synthesizing SOBDDs

Going further

Many optimizations exist to make this approach more efficient.

- ▷ A crucial one is the use of the *computed* table for memoization.

To actually achieve CTL model checking, we still need some more operators to deal with *renaming* and *preimage computation*.

DON'T PANIC![1]

We have seen enough for this course: check the book for more.

[1] And remember your towel. . .

Efficiency of the BDD-based CTL model checking

The BDD-based approach

- does not reduce the *worst-case* complexity of CTL model checking;
- *greatly helps in many practical applications*.
  - ▷ Depending on the underlying structure of the TS, gains can be huge.
  - ▷ In some applications, TSs with 10^{120} states could be verified through this technique [Kat10] (and could not by explicit techniques).

1 Symbolic model checking: why, what and how?

2 CTL model checking through switching functions

3 Efficient encoding through ROBDDs

4 A glance at other approaches for efficient model checking

Beyond CTL and BDDs

There are many other techniques and approaches to tackle the state-space explosion problem

- ▷ not only for CTL,
- ▷ not only via symbolic techniques.

$\Longrightarrow$ We briefly mention some of them in the next slides (non-exhaustive list).

Model checking with partial order reduction

Consider a TS arising from the interleaving between different processes and an LTL or CTL* formula to check.

- In general, one must consider an exponential # of orderings of actions: all possible interleavings must be checked.
- Now, if the actions of the processes are "independent" (e.g., one executes $x := x + 2$ while another one does $y := y - 3$), different orderings can be considered equivalent w.r.t. the property to check.

$\Longrightarrow$ **Partial order reduction techniques aim at reducing the state space by reducing the # of orderings to consider.**

$\Longrightarrow$ **Can lead to huge gain since the state space grows exponentially with the # of processes.**

Model checking with symmetry reduction

Various methods considering quotient state spaces based on symmetry reduction.

- The most aggressive methods may lead to false negatives (i.e., unreported errors) but in practice permit to deal with systems otherwise not checkable at all.

$\Longrightarrow$ **Still useful in practice even if not complete.**

Bounded model checking

Instead of considering infinite executions of the system, we settle for executions up to K steps.

- No guarantee that the system will work as intended beyond K steps.
- In practice, most errors can be detected after a reasonable number of steps.
- Technically based on extremely optimized SAT solvers.
  - ▷ NP-hard problem but efficiently solvable in most practical cases.

$\Longrightarrow$ **Quite efficient in practice.**

References I

J.-P. Katoen.
Reduced ordered binary decision diagrams, lecture #13 of advanced model checking, December 2010.