COMP4600 Advanced algorithms: Algorithms for verification (3 lectures)

Andreas Bauer

NICTA Software Systems Research Group & The Australian National University

http://baueran.multics.org/

Caveat

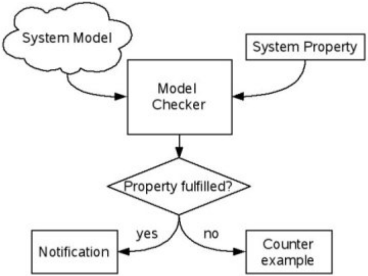
- Although model checking is my research area...
- ...this is the first time, I'm giving a comprehensive lecture on model checking.
- We will look at MC foremost from a technical/algorithmic point of view, not so much from a formal/logical one.
- However, there will be a wee bit of logic introduced/used that everyone should be able to follow who knows standard propositional logic.
- Let's see how we go...

What do we mean by verification?

- System is modelled as finite state-transition system.
- Properties are written down in propositional temporal logic.
- Verification = exhaustive state-space search of system model.
- Diagnostic counterexample, if any.

What do we mean by verification?

- System is modelled as finite state-transition system.
- Properties are written down in propositional temporal logic.
- Verification = exhaustive state-space search of system model.
- Diagnostic counterexample, if any.



Model checking

- Does system model M satisfy temporal logic property φ (written $M \models \varphi$)?
- Normally, checking of functional correctness (not error-freeness in the intuitive sense).
- System (model) only as good/reliable as its designers anticipated.
- Model checking cannot detect implementation errors (e.g., compiler bugs) $\Rightarrow$ Systems testing.

Model checking

- Does system model M satisfy temporal logic property φ (written $M \models \varphi$)?
- Normally, checking of functional correctness (not error-freeness in the intuitive sense).
- System (model) only as good/reliable as its designers anticipated.
- Model checking cannot detect implementation errors (e.g., compiler bugs) $\Rightarrow$ Systems testing.

Let's be more formal!

What is M, what is φ, what is "satisfy"?

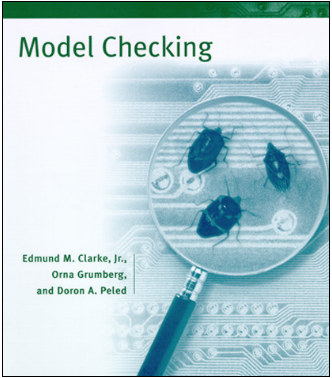
By the way. . .

- MC "won" Turing award in 2007 (Clarke, Emmerson, Sifakis):

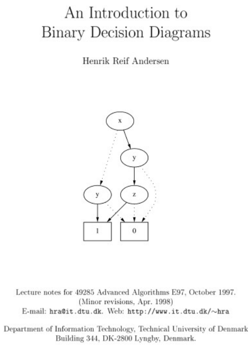
- Most widely used industrial design verification technique.
- Focus shifted from verification of simple designs (e.g., communication protocol specifications) to entire software systems (e.g., business information system).

By the way. . .

A lot (but not all) of the material in these lectures is based upon



(MIT Press, 2003)


An Introduction to
Binary Decision Diagrams

Henrik Reif Andersen

Lecture notes for 49285 Advanced Algorithms E97, October 1997.
(Minor revisions, Apr. 1998)
E-mail: hra@it.dtu.dk. Web: http://www.it.dtu.dk/~hra

Department of Information Technology, Technical University of Denmark
Building 344, DK-2800 Lyngby, Denmark.

(Henrik Reif Andersen '97)

Kripke structures

$M = (S, R, L)$ over set of propositions, AP, where

- S is set of states,
- $R \subseteq S \times S$ a transition relation,
- $L : S \rightarrow 2^{AP}$ a labelling function.

Kripke structures

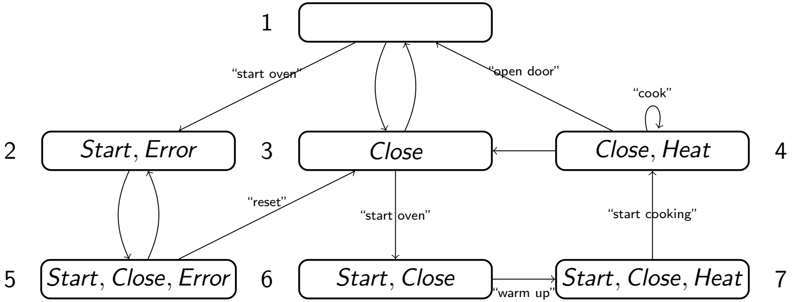
$M = (S, R, L)$ over set of propositions, AP, where

- S is set of states,
- $R \subseteq S \times S$ a transition relation,
- $L : S \to 2^{AP}$ a labelling function.

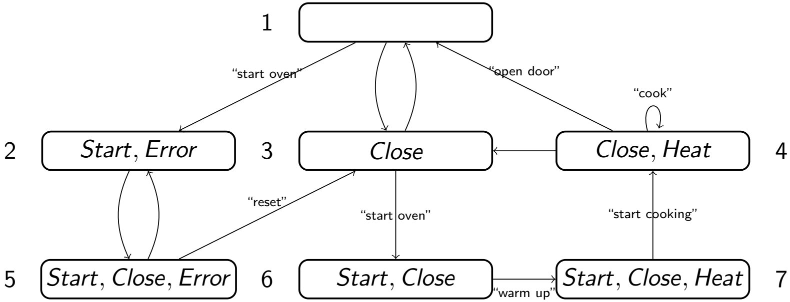
Modelling the behaviour of a microwave oven

- $AP = \{Start, Close, Heat, Error\}$
- $S = \{S_1, \ldots, S_7\}$
- $R = \{(S_1, S_3), (S_1, S_2), (S_3, S_1), \ldots\}$
- $L(S_1) = \emptyset, L(S_2) = \{Start, Error\}, \ldots$

Kripke structures



Kripke structures



Possible behaviour of microwave oven

Trace/word: $\{Close\}, \{Start, Close\}, \{Start, Close, Heat\}, \{Close, Heat\}, \{Close, Heat\}, \ldots$

Kripke structures

- Behaviour of microwave = all possible traces/words of M.
- Trace/word = linear Kripke structure.
- Traces typically infinite due to loops (i.e., reactive system never switched off).

Definition

Let $\Sigma = 2^{AP}$ be a finite alphabet. Let Σ^ω denote set of all infinite traces over Σ. Behaviour of M can be given as

$$\{w \in \Sigma^\omega \mid \text{for all } i \in \mathbb{N}_0 \text{ there are } m, n \in \mathbb{N} \text{ s.t. } (S_m, S_m) \in R \text{ and } w(i) = L(S_m) \text{ and } w(i+1) = L(S_n)\}$$

(We could also demand that $L(S_0) = w(0)$, had we an S_0.)

Kripke structures—where they come from

- If we model a system directly in terms of a Kripke structure, we are, sort of, performing the model checking by hand already.
- Model generation: Convert abstract system model (e.g., source code) into Kripke structure automatically.

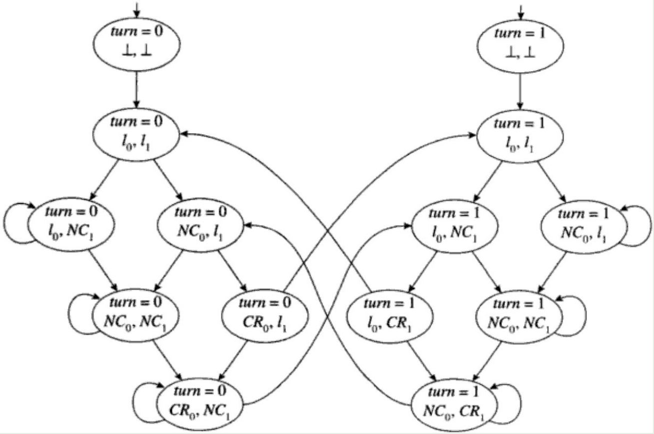
Example program: $P = m:\ \text{cobegin}\ P_0 || P_1\ \text{coend}\ m'$

$P_0 :: \quad l_0 : \quad$ **while** *True* **do**
$\qquad\qquad NC_0 :$ **wait**$(turn = 0)$;
$\qquad\qquad CR_0 : turn := 1$;
$\qquad$ **end while**;
$\quad l_0'$

$P_1 :: \quad l_1 : \quad$ **while** *True* **do**
$\qquad\qquad NC_1 :$ **wait**$(turn = 1)$;
$\qquad\qquad CR_1 : turn := 0$;
$\qquad$ **end while**;
$\quad l_1'$

Kripke structures—where they come from

The corresponding Kripke structure



cf. Clarke, Grumberg, Peled (2003): "Model Checking", MIT Press.

Linear-time temporal logic

- Pnueli, 1977; Turing award 1996.
- LTL = propositional logic + two temporal operators (**X**, **U**).
- Used as formal specification language for temporal order of events.

Propositional logic (recap)

- $\varphi = a \wedge \neg b \vee c$ has model $\{\alpha(a) = 1, \alpha(b) = 0, \alpha(c) = 1\}$
- We can write this as singleton "Kripke structure" $M = \{a, c\}$.
- Thus, $M \models \varphi$ ("M satisfies/is a model for φ.")

Linear-time temporal logic

LTL syntax

- Every propositional logic formula is also an LTL formula.
- If φ is an LTL formula, then so are $\mathbf{X}\varphi$ and $\varphi\mathbf{U}\varphi'$.
- BNF: $\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$.

Linear-time temporal logic

LTL syntax

- Every propositional logic formula is also an LTL formula.
- If φ is an LTL formula, then so are $\mathbf{X}\varphi$ and $\varphi\mathbf{U}\varphi'$.
- BNF: $\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{X}\varphi \mid \varphi\mathbf{U}\varphi$.

LTL semantics: w series of assignments/worlds, i position in w

$$
\begin{array}{lll}
w, i \models p & \text{iff} & p \in w(i) \\
w, i \models \neg\varphi & \text{iff} & w, i \models \varphi \text{ is not true} \\
w, i \models \varphi \wedge \psi & \text{iff} & w, i \models \varphi \text{ and } w, i \models \psi \\
w, i \models \mathbf{X}\varphi & \text{iff} & w, i+1 \models \varphi \\
w, i \models \varphi\mathbf{U}\psi & \text{iff} & \text{there is } k \geq i \text{ s.t. } w, k \models \psi, \text{ and for all } i \leq j < k \text{ we have } w, j \models \varphi
\end{array}
$$

More generally, note how models of $\varphi \in LTL$ are elements from Σ^ω ($\Sigma = 2^{AP}$ is our alphabet).
Let $\mathcal{L}(\varphi) = \{w \in \Sigma^\omega \mid w, 0 \models \varphi\}$ be the language of φ.

Linear-time temporal logic

Some more useful LTL operators and shortcuts (syntactic "sugar"):

- $\mathit{true} = p \vee \neg p$
- $\mathit{false} = \neg \mathit{true}$
- $\varphi \vee \psi = \neg(\neg\varphi \wedge \neg\psi)$
- $\varphi \rightarrow \psi = \neg\varphi \vee \psi$
- $\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \wedge (\psi \rightarrow \varphi)$
- $\mathbf{F}\varphi = \mathit{true}\mathbf{U}\varphi$ ("eventually φ")
- $\mathbf{G}\varphi = \neg\mathbf{F}\neg\varphi$ ("always φ")
- $\varphi\mathbf{R}\psi = \neg(\neg\varphi\mathbf{U}\neg\psi)$ ("release ψ when φ becomes true")

Linear-time temporal logic

Some LTL specifications:

Invariants:

- $\mathbf{G}\neg(crit_1 \wedge crit_2)$ (mutual exclusion)
- $\mathbf{G}(preset_1 \vee \ldots \vee preset_n)$ (deadlock freedom)

Response, recurrence:

- $\mathbf{G}(try_1 \rightarrow \mathbf{F}crit_1)$ (eventual access to critical section)
- $\mathbf{GF}\neg crit_1$ (no starvation in critical section)

Strong fairness:

- $\mathbf{GF}(try_1 \wedge \neg crit_2) \rightarrow \mathbf{GF}crit_1$ (strong fairness)

LTL model checking

Is the following decision problem:

- **Input**: Kripke structure M, LTL formula φ.
- **Question**: Does $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$ hold (sometimes written as $M \models \varphi$)?

LTL model checking

Is the following decision problem:

- **Input**: Kripke structure M, LTL formula φ.
- **Question**: Does $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi)$ hold (sometimes written as $M \models \varphi$)?

Example: Microwave oven

$\mathcal{L}(M) \subseteq \mathcal{L}(\mathbf{G}(\mathit{Heat} \rightarrow \mathit{Close}))$

LTL model checking

Key ideas:

- $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi) \Leftrightarrow \mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) = \emptyset$
- If $\mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) \neq \emptyset$, we have a counterexample.

LTL model checking

Key ideas:

- $\mathcal{L}(M) \subseteq \mathcal{L}(\varphi) \Leftrightarrow \mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) = \emptyset$
- If $\mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) \neq \emptyset$, we have a counterexample.

How do we test if $\mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) = \emptyset$?

LTL model checking

Theorem

For every $\varphi \in$ LTL, there exists an ω-automaton, $\mathcal{A}$, s.t., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$.

LTL model checking

Theorem

For every $\varphi \in LTL$*, there exists an* ω*-automaton,* $\mathcal{A}$*, s.t.,* $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$.

Corollary

We can solve the LTL model checking problem by testing if $\mathcal{L}(M \times \mathcal{A}_{\neg\varphi}) = \emptyset$.

LTL model checking

Theorem

For every $\varphi \in LTL$, there exists an ω-automaton, $\mathcal{A}$, s.t., $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\varphi)$.

Corollary

We can solve the LTL model checking problem by testing if $\mathcal{L}(M \times \mathcal{A}_{\neg\varphi}) = \emptyset$.

Note that, $M \times \mathcal{A}_{\neg\varphi}$ is normally too big to be explicitly computed (but we disregard that fact for now).

LTL model checking—ω-automata

Definition

An ω-automaton is a five-tuple $\mathcal{A} = (\Sigma, Q, Q_0, \delta, \mathcal{F})$ where

- Σ is the input alphabet,
- Q a finite set of states,
- $Q_0 \subseteq Q$ a distinguished set of initial states,
- $\delta : Q \to 2^Q$ a transition relation, and
- $\mathcal{F}$ an acceptance condition.

A run ρ of $\mathcal{A}$ over a word $w \in \Sigma^\omega$ is a mapping $\mathbb{N}_0 \to Q$ s.t.

- $\rho(0) \in Q_0$, and
- $\rho(i+1) \in \delta(\rho(i), w(i))$ for all $i \in \mathbb{N}_0$.

LTL model checking—ω-automata

Generalised Büchi automaton (GBA): $\mathcal{F} = \{F_1, \ldots, F_n\}$

- $F_i \subseteq Q$ is an accepting set.
- ρ is accepting iff $\mathit{Inf}(\rho) \cap F_i \neq \emptyset$ for $1 \leq i \leq n$.

LTL model checking—ω-automata

Generalised Büchi automaton (GBA): $\mathcal{F} = \{F_1, \ldots, F_n\}$

- $F_i \subseteq Q$ is an accepting set.
- ρ is accepting iff $\mathit{Inf}(\rho) \cap F_i \neq \emptyset$ for $1 \leq i \leq n$.

Definition

A word w is accepted by an ω-automaton $\mathcal{A}$ iff $\mathcal{A}$ has an accepting run over w.

LTL model checking—ω-automata

Generalised Büchi automaton (GBA): $\mathcal{F} = \{F_1, \ldots, F_n\}$

- $F_i \subseteq Q$ is an accepting set.
- ρ is accepting iff $Inf(\rho) \cap F_i \neq \emptyset$ for $1 \leq i \leq n$.

Definition

A word w is accepted by an ω-automaton $\mathcal{A}$ iff $\mathcal{A}$ has an accepting run over w.

Büchi automaton (BA sometimes NBA): $\mathcal{F} = F$.

- $F \subseteq Q$ is a set of accepting states.
- ρ is accepting iff $Inf(\rho) \cap F \neq \emptyset$.

Streett automaton: $\mathcal{F} = \{(E_1, F_1), \ldots, (E_n, F_n)\}$

- $E_i, F_i \subseteq Q$.
- ρ is accepting iff $Inf(\rho) \cap F_i \neq \emptyset \rightarrow Inf(\rho) \cap E_i \neq \emptyset$ for $1 \leq i \leq n$.

LTL model checking—ω-automata

Recall: An automaton is deterministic iff for all $q \in Q$, and $\sigma \in \Sigma$, $\delta(q, \sigma)$ is a singleton; that is, if δ is, in fact, a function.

LTL model checking—ω-automata

Recall: An automaton is deterministic iff for all $q \in Q$, and $\sigma \in \Sigma$, $\delta(q, \sigma)$ is a singleton; that is, if δ is, in fact, a function.

Theorem

NBAs are strictly more expressive than DBAs.

LTL model checking—ω-automata

Recall: An automaton is deterministic iff for all $q \in Q$, and $\sigma \in \Sigma$, $\delta(q, \sigma)$ is a singleton; that is, if δ is, in fact, a function.

Theorem

NBAs are strictly more expressive than DBAs.

Proof.

$L = \mathcal{L}((a+b)^* a^\omega)$ NBA- but not DBA-definable. $\square$

LTL model checking—ω-automata

Recall: An automaton is deterministic iff for all $q \in Q$, and $\sigma \in \Sigma$, $\delta(q, \sigma)$ is a singleton; that is, if δ is, in fact, a function.

Theorem

NBAs are strictly more expressive than DBAs.

Proof.

$L = \mathcal{L}((a+b)^* a^\omega)$ NBA- but not DBA-definable. $\square$

Theorem

NBAs can encode every LTL property, but not vice versa.

LTL model checking—ω-automata

Recall: An automaton is deterministic iff for all $q \in Q$, and $\sigma \in \Sigma$, $\delta(q, \sigma)$ is a singleton; that is, if δ is, in fact, a function.

Theorem

NBAs are strictly more expressive than DBAs.

Proof.

$L = \mathcal{L}((a + b)^* a^\omega)$ NBA- but not DBA-definable. □

Theorem

NBAs can encode every LTL property, but not vice versa.

Proof.

"p occurs at least on even positions" □

LTL-to-automata translation—prerequisites

Definition

The syntactic closure of φ, $cl(\varphi)$, consists of all subformulas of ψ of φ and their negation $\neg\psi$.

LTL-to-automata translation—prerequisites

Definition

The syntactic closure of φ, $cl(\varphi)$, consists of all subformulas of ψ of φ and their negation $\neg\psi$.

Example: $\varphi = a\mathbf{U}(\neg a \wedge b)$

LTL-to-automata translation—prerequisites

Definition

The syntactic closure of φ, $cl(\varphi)$, consists of all subformulas of ψ of φ and their negation $\neg\psi$.

Example: $\varphi = a\mathbf{U}(\neg a \wedge b)$

$$cl(\varphi) = \{a, b, \neg a, \neg b, \neg a \wedge b, \neg(\neg a \wedge b), \varphi, \neg\varphi\}$$

LTL-to-automata translation

GBA for $\varphi \in LTL$:

- Q: elements of $cl(\varphi)$, promised to be true.
- Q_0: states containing φ.
- δ: repr. as graph $G = (V, E)$, where
  - V all complete subsets of $cl(\varphi)$ (i.e., $c \in V$ iff for all $\psi \in cl(\varphi)$ either $\psi \in c$ or $\neg\psi \in c$, and for all $\varphi' = \psi \wedge \psi' \in cl(\varphi)$ we have that $\varphi' \in c$ iff $\psi \in c$ and $\psi' \in c$.)
  - $(c, d) \in E$ iff
    - for any $\varphi' = \psi \mathbf{U} \psi' \in cl(\varphi)$, $\varphi' \in c$ iff either $\psi' \in c$, or $\psi \in c$ and $\varphi' \in d$;
    - for any $\varphi' = \mathbf{X}\psi \in cl(\varphi)$, $\varphi' \in c$ iff $\psi \in d$.
- $\mathcal{F} = \{\{q \in Q \mid \psi \mathbf{U} \psi' \notin q \text{ or } \psi' \in q\} \mid \psi \mathbf{U} \psi' \in cl(\varphi)\}$

LTL-to-automata translation—complexity considerations

How big is $|Q|$ (resp. $\mathcal{A}_\varphi$) at most?

LTL-to-automata translation—complexity considerations

How big is $|Q|$ (resp. $\mathcal{A}_\varphi$) at most?

- $|cl(\varphi)| = O(|\varphi|)$.

LTL-to-automata translation—complexity considerations

How big is $|Q|$ (resp. $\mathcal{A}_\varphi$) at most?

- $|cl(\varphi)| = O(|\varphi|)$.
- There are at most $2^{O(|\varphi|)}$ many possible subsets of $cl(\varphi)$.

LTL-to-automata translation—complexity considerations

How big is $|Q|$ (resp. $\mathcal{A}_\varphi$) at most?

- $|cl(\varphi)| = O(|\varphi|)$.
- There are at most $2^{O(|\varphi|)}$ many possible subsets of $cl(\varphi)$.

That's why we do LTL model checking as $\mathcal{L}(M \times \mathcal{A}_{\neg\varphi}) = \emptyset$ rather than $\mathcal{L}(M) \cap \mathcal{L}(\overline{\mathcal{A}_\varphi}) = \emptyset$:

- Complementation of formula $O(1)$ vs.
- complementation of automaton $\approx O(2^{|Q|})$.

LTL-to-automata translation—optimisations

GBA acceptance more difficult to test than NBA acceptance:

- Turn all states into tuples (q, i), where i is counter.
- Initially, $i = 0$; counter counts modulo $|\mathcal{F}|$.
- $i = i + 1$ if the ith set F_i of $\mathcal{F}$ is reached (i.e., if q not accepting counter doesn't do anything).
- Now, we only need to check one accepting set, $F_0 \times \{0\}$.

LTL-to-automata translation—optimisations

More formally:

LTL-to-automata translation—optimisations

More formally:
From GBA $\mathcal{A} = (\Sigma, Q, Q_0, \delta, \mathcal{F} = F_1, \ldots, F_n)$, we construct NBA $\mathcal{B} = (\Sigma, Q', Q'_0, \delta', F')$:

- $Q' = Q \times \{1, \ldots, n\}$
- $\delta' \subseteq Q' \times Q'$, where $((q, i), (s, j)) \in \delta'$ iff $(q, s) \in \delta$ AND $q \notin F_i$ and $i = j$, or $q \in F_i$ and $j = (i + 1) \mod n$.
- $Q'_0 = \{(q, 0) \mid q \in Q_0\}$
- $F' = \{(q, 0) \mid q \in F_0\}$

LTL-to-automata translation—optimisations

More formally:
From GBA $\mathcal{A} = (\Sigma, Q, Q_0, \delta, \mathcal{F} = F_1, \ldots, F_n)$, we construct NBA $\mathcal{B} = (\Sigma, Q', Q_0', \delta', F')$:

- $Q' = Q \times \{1, \ldots, n\}$
- $\delta' \subseteq Q' \times Q'$, where $((q,i),(s,j)) \in \delta'$ iff $(q,s) \in \delta$ AND $q \notin F_i$ and $i = j$, or $q \in F_i$ and $j = (i+1) \mod n$.
- $Q_0' = \{(q,0) \mid q \in Q_0\}$
- $F' = \{(q,0) \mid q \in F_0\}$

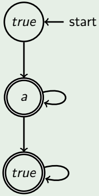
Edge-labelled vs. state-labelled NBA:

- Both used; arguably, edge-labelled more common.
- Easy translation between the two models.

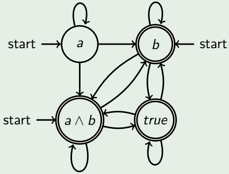
LTL-to-automata translation

Some example NBAs: (w/o redundant states)

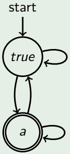
$\mathbf{X}a$:



$a\mathbf{U}b$:



$\mathbf{GF}a$:



The temporal formulae inside of states are just used for constructing automata. Later we can merely remember the Boolean formulae that are satisfied in order to enter a state as above. (You should convince yourself that this is an equivalent representation wrt. the accepted languages!)

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$?

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$? in P (i.e., linear-time algorithm)

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$? in P (i.e., linear-time algorithm)
- $\mathcal{L}(\mathcal{A}) = / \neq \Sigma^\omega$?

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$? in P (i.e., linear-time algorithm)
- $\mathcal{L}(\mathcal{A}) = / \neq \Sigma^\omega$? is PSpace-complete

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$? in P (i.e., linear-time algorithm)
- $\mathcal{L}(\mathcal{A}) = / \neq \Sigma^\omega$? is PSpace-complete
- $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ NBA representable (closure under intersection)

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$? in P (i.e., linear-time algorithm)
- $\mathcal{L}(\mathcal{A}) = / \neq \Sigma^\omega$? is PSpace-complete
- $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ NBA representable (closure under intersection)
- $\overline{\mathcal{L}(\mathcal{A})}$ NBA representable (closure under complement)

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$? in P (i.e., linear-time algorithm)
- $\mathcal{L}(\mathcal{A}) = / \neq \Sigma^\omega$? is PSpace-complete
- $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ NBA representable (closure under intersection)
- $\overline{\mathcal{L}(\mathcal{A})}$ NBA representable (closure under complement)
- NBAs are not closed under determinisation, i.e., there exists an NBA, $\mathcal{A}$, for which there is no DBA, $\mathcal{B}$, s.t. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

Important properties of NBAs

Let $\mathcal{A}$ be an NBA over Σ.

- $\mathcal{L}(\mathcal{A}) = / \neq \emptyset$? in P (i.e., linear-time algorithm)
- $\mathcal{L}(\mathcal{A}) = / \neq \Sigma^\omega$? is PSpace-complete
- $\mathcal{L}(\mathcal{A}) \cap \mathcal{L}(\mathcal{B})$ NBA representable (closure under intersection)
- $\overline{\mathcal{L}(\mathcal{A})}$ NBA representable (closure under complement)
- NBAs are not closed under determinisation, i.e., there exists an NBA, $\mathcal{A}$, for which there is no DBA, $\mathcal{B}$, s.t. $\mathcal{L}(\mathcal{A}) = \mathcal{L}(\mathcal{B})$.

Closure under complement and intersection are the prerequisites for what is known as automata-theoretic model checking.

Automata theoretic model checking

Given $M = (S, s_0, R, L)$ and $\mathcal{A}_\varphi = (\Sigma, Q, Q_0, \delta, F)$, we define the "product automaton" $M \times \mathcal{A}_\varphi = (\Sigma, Q', Q'_0, \delta', F')$ by

- $Q' = \{(s, q) \in S \times Q \mid L(s) \text{ satisfies } q\}$ (recall: q contains a Boolean formula!)
- $Q'_0 = \{(s_0, q) \in Q' \mid q \in Q_0\}$
- $\delta' = \{((s, q), (s', q')) \in Q' \times Q' \mid (s, s') \in R \text{ and } (q, q') \in \delta\}$
- $F' = \{(s, q) \in Q' \mid q \in F\}$

Automata theoretic model checking

Given $M = (S, s_0, R, L)$ and $\mathcal{A}_\varphi = (\Sigma, Q, Q_0, \delta, F)$, we define the "product automaton" $M \times \mathcal{A}_\varphi = (\Sigma, Q', Q'_0, \delta', F')$ by

- $Q' = \{(s, q) \in S \times Q \mid L(s) \text{ satisfies } q\}$ (recall: q contains a Boolean formula!)
- $Q'_0 = \{(s_0, q) \in Q' \mid q \in Q_0\}$
- $\delta' = \{((s, q), (s', q')) \in Q' \times Q' \mid (s, s') \in R \text{ and } (q, q') \in \delta\}$
- $F' = \{(s, q) \in Q' \mid q \in F\}$

What is the accepted language of this automaton?

Automata theoretic model checking

Given $M = (S, s_0, R, L)$ and $\mathcal{A}_\varphi = (\Sigma, Q, Q_0, \delta, F)$, we define the "product automaton" $M \times \mathcal{A}_\varphi = (\Sigma, Q', Q'_0, \delta', F')$ by

- $Q' = \{(s, q) \in S \times Q \mid L(s) \text{ satisfies } q\}$ (recall: q contains a Boolean formula!)
- $Q'_0 = \{(s_0, q) \in Q' \mid q \in Q_0\}$
- $\delta' = \{((s, q), (s', q')) \in Q' \times Q' \mid (s, s') \in R \text{ and } (q, q') \in \delta\}$
- $F' = \{(s, q) \in Q' \mid q \in F\}$

What is the accepted language of this automaton?

Lemma

$\mathcal{L}(M \times \mathcal{A}_\varphi) = \mathcal{L}(M) \cap \mathcal{L}(\mathcal{A}_\varphi)$

Automata theoretic model checking

Recall: we need to test if $\mathcal{L}(M \times \mathcal{A}_\varphi) = \emptyset$. (How do we do it?)

Automata theoretic model checking

Recall: we need to test if $\mathcal{L}(M \times \mathcal{A}_\varphi) = \emptyset$. (How do we do it?)

Theorem

$\mathcal{L}(M \times \mathcal{A}_\varphi) = \emptyset \Leftrightarrow$ *there is no reachable cycle containing a state from F.*

Automata theoretic model checking

Recall: we need to test if $\mathcal{L}(M \times \mathcal{A}_\varphi) = \emptyset$. (How do we do it?)

Theorem

$\mathcal{L}(M \times \mathcal{A}_\varphi) = \emptyset \Leftrightarrow$ *there is no reachable cycle containing a state from F.*

Polynomial-time algorithm (e.g., Tarjan's SCC finding alg.) does the job (cf. Knuth Vol. 3)

Automata theoretic model checking

Recall: we need to test if $\mathcal{L}(M \times \mathcal{A}_\varphi) = \emptyset$. (How do we do it?)

Theorem

$\mathcal{L}(M \times \mathcal{A}_\varphi) = \emptyset \Leftrightarrow$ *there is no reachable cycle containing a state from F.*

Polynomial-time algorithm (e.g., Tarjan's SCC finding alg.) does the job (cf. Knuth Vol. 3)

Corollary

LTL model checking is in PTime, if M and $\mathcal{A}_\varphi$ are given.

. . . which is never the case in practice. :-(

Detour (I): Tarjan's algorithm for SCC identification

Idea: Does a forward DFS to visit all nodes once to assign increasing index, and upon returning from the recursive calls, assigns low-indices that point to the node with the smallest index reachable from each respective node.
When low-index of a node = index of that node, we have a root of an SCC.

Detour (I): Tarjan's algorithm for SCC identification

```
algorithm tarjan is
  input: graph G = (V, E)
  output: set of strongly connected components (sets of vertices)

  index := 0
  S := empty
  for each v in V do
    if (v.index is undefined) then
      strongconnect(v)
    end if
  repeat

  function strongconnect(v)
    // Set the depth index for v to the smallest unused index
    v.index := index
    v.lowlink := index
    index := index + 1
    S.push(v)

    // Consider successors of v
    for each (v, w) in E do
      if (w.index is undefined) then
        // Successor w has not yet been visited; recurse on it
        strongconnect(w)
        v.lowlink := min(v.lowlink, w.lowlink)
      else if (w is in S) then
        // Successor w is in stack S and hence in the current SCC
        v.lowlink := min(v.lowlink, w.index)
      end if
    repeat

    // If v is a root node, pop the stack and generate an SCC
    if (v.lowlink = v.index) then
      start a new strongly connected component
      repeat
        w := S.pop()
        add w to current strongly connected component
      until (w = v)
      output the current strongly connected component
    end if
  end function
```

Some observations:

- strongconnect(x) is called once for every node.
- The for-each-loop at most considers each edge twice (to find neighbours of all nodes)
- (But not all nodes have necessarily an outgoing edge.)
- That is, runtime of $O(|V| + |E|)$.

Detour (II): On-The-Fly Bad-Cycle-Detection

Idea:

- Often M not given, so one needs to construct M from an abstract model (e.g., code, call it $\mathcal{M}$).
- Instead of doing it all at once, one can construct M on-the-fly (cf. Vardi et al, CAV'90).
- Observe, it is easy to obtain initial states (i.e., initial in M and $\mathcal{A}_\varphi$)
- Algorithm proceeds by expanding more states in an "as needed" manner, and looks if a cycle can be found which hosts an accepting state from $\mathcal{A}_\varphi$.
- In practice, there's a fair chance it will find an accepting cycle before having expanded all nodes of M.

Detour (II): On-The-Fly Bad-Cycle-Detection

```
Input: M and Aφ,
Initialize:  Stack1:=∅ , Stack2:=∅ ,
             Table1:=∅ , Table2:=∅ ;
procedure Main() {
  foreach  s ∈ Init⊗
    { if  s ∉ Table1  then DFS1(s); }
  output("no bad cycle");
  exit;
}
procedure DFS1(s)  {
   push(s,Stack1);
   hash(s,Table1);
   foreach  t ∈ Succ⊗(s)
     { if t ∉ Table1 then DFS1(t); }
   if  s ∈ F⊗ then {  DFS2(s);  }
   pop(Stack1);
}
```

```
procedure DFS2(s){
  push(s,Stack2);
  hash(s,Table2) ;
  foreach t ∈ Succ⊗(s) do {
    if t ∉ Table2 then
           DFS2(t)
    else  if t is on Stack1 {
      output("bad cycle:");
      output(Stack1,Stack2,t);
      exit;
      }
    }
   pop(Stack2);
}
```

(Slide shamelessly stolen from Kousha Etessami.)

Complexity of LTL model checking

Recall: Input to the LTL model checking problem is a KS, M, and φ. The question to be answered is, does $\mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) \neq \emptyset$ hold?

Theorem

The LTL model checking problem can be answered

- *in time $O(2^{O(|\varphi|)} \cdot |M|)$ (cf. size of NBA), or*
- *in PSpace (but potentially ExpTime; cf. on-the-fly alg.).*

Complexity of LTL model checking

Recall: Input to the LTL model checking problem is a KS, M, and φ. The question to be answered is, does $\mathcal{L}(M) \cap \mathcal{L}(\neg\varphi) \neq \emptyset$ hold?

Theorem

The LTL model checking problem can be answered

- *in time $O(2^{O(|\varphi|)} \cdot |M|)$ (cf. size of NBA), or*
- *in PSpace (but potentially ExpTime; cf. on-the-fly alg.).*

The latter explains why model checking works in practice: the NBA can be fixed for most formulae, and the subsequent state-space exploration optimised.

Complexity of LTL model checking

Theorem

LTL model checking is PSpace-complete.

Proof.

Hardness: Reduction from LTL satisfiability, which is also PSpace-complete: $\mathcal{L}(\varphi) = \emptyset \Leftrightarrow \mathcal{L}(\varphi) \cap \Sigma^\omega = \emptyset \Leftrightarrow \Sigma^\omega \models \neg\varphi$.
Membership: Nondeterministic algorithm: Expand NBA on-the-fly (similar to expansion of M earlier) and guess

- a path through M, and
- a state, l, in the NBA which lies on an accepting loop.

Each expansion step of the NBA can be done in PTime, and to check whether l is visited again is constant. If guessed path goes through l twice, we know that we have a counterexample. □

Computation Tree Logic (CTL)

CTL syntax

$$\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{AX}\varphi \mid \mathbf{EX}\varphi \mid \mathbf{A}(\varphi\mathbf{U}\varphi) \mid \mathbf{E}(\varphi\mathbf{U}\varphi)$$

Computation Tree Logic (CTL)

CTL syntax

$$\varphi ::= p \in AP \mid \neg\varphi \mid \varphi \wedge \varphi \mid \mathbf{AX}\varphi \mid \mathbf{EX}\varphi \mid \mathbf{A}(\varphi\mathbf{U}\varphi) \mid \mathbf{E}(\varphi\mathbf{U}\varphi)$$

- Note, there's no arbitrary nesting of path quantifiers (cf. CTL*).
- For example, you can't say $\mathbf{XAF}\varphi$ in CTL.
- But $\mathbf{EFEG}\varphi$ is OK.

CTL—syntactic sugar and equalities

- $\mathbf{AX}\varphi = \neg\mathbf{EX}(\neg\varphi)$
- $\mathbf{EF}\varphi = \mathbf{E}(\mathit{true}\mathbf{U}\varphi)$
- $\mathbf{AG}\varphi = \neg\mathbf{EF}(\neg\varphi)$
- $\mathbf{AF}\varphi = \neg\mathbf{EG}(\neg\varphi)$
- $\mathbf{A}(\varphi\mathbf{U}\psi) = \neg\mathbf{E}(\neg\psi\mathbf{U}(\neg\varphi \wedge \neg\psi)) \wedge \neg\mathbf{EG}\neg\psi$
- $\mathbf{A}(\varphi\mathbf{R}\psi) = \neg\mathbf{E}(\neg\varphi\mathbf{U}\neg\psi)$
- $\mathbf{E}(\varphi\mathbf{R}\psi) = \neg\mathbf{A}(\neg\varphi\mathbf{U}\neg\psi)$

CTL—syntactic sugar and equalities

- $\mathbf{AX}\varphi = \neg\mathbf{EX}(\neg\varphi)$
- $\mathbf{EF}\varphi = \mathbf{E}(\mathit{true}\mathbf{U}\varphi)$
- $\mathbf{AG}\varphi = \neg\mathbf{EF}(\neg\varphi)$
- $\mathbf{AF}\varphi = \neg\mathbf{EG}(\neg\varphi)$
- $\mathbf{A}(\varphi\mathbf{U}\psi) = \neg\mathbf{E}(\neg\psi\mathbf{U}(\neg\varphi \wedge \neg\psi)) \wedge \neg\mathbf{EG}\neg\psi$
- $\mathbf{A}(\varphi\mathbf{R}\psi) = \neg\mathbf{E}(\neg\varphi\mathbf{U}\neg\psi)$
- $\mathbf{E}(\varphi\mathbf{R}\psi) = \neg\mathbf{A}(\neg\varphi\mathbf{U}\neg\psi)$

Corollary

Any CTL formula can be expressed in terms of $\neg$, $\vee$, **EX**, **EU** *and* **EG** *alone.*

CTL—semantics

CTL semantics: Let $M = (S, R, L)$ be defined as usual; $s \in S$.

$M, s \models p$	iff	$p \in L(s)$
$M, s \models \neg\varphi$	iff	$M, s \models \varphi$ is not true
$M, s \models \varphi \wedge \psi$	iff	$M, s \models \varphi$ and $M, s \models \psi$
$M, s \models \mathbf{AX}\varphi$	iff	for all $s \to s_1$, $M, s_1 \models \varphi$
$M, s \models \mathbf{EX}\varphi$	iff	there is a $s \to s_1$, s.t. $M, s_1 \models \varphi$
$M, s \models \mathbf{A}(\varphi \mathbf{U} \psi)$	iff	for all $s_1 \to s_2 \to \ldots,$ where $s_1 = s$, there is a s_k, s.t. $M, s_k \models \psi$, and $M, s_j \models \varphi$ for all s_j, where $0 \leq j < k$
$M, s \models \mathbf{E}(\varphi \mathbf{U} \psi)$	iff	there is a $\ldots$

CTL—examples

Some CTL specifications:

- $\mathbf{EF}(\mathit{Start} \wedge \neg \mathit{Ready})$: It is possible to reach a state in which Start but not Ready holds.
- $\mathbf{AG}(\mathit{Req} \rightarrow \mathbf{AF}\mathit{Ack})$: Every req. is eventually answered.
- $\mathbf{AG}(\mathbf{AF}\mathit{DeviceEnabled})$: The device is enabled infinitely often on all paths.
- $\mathbf{AG}(\mathbf{EF}\mathit{Restart})$: From any state it is possible to reach a state in which Restart holds.

CTL model checking—labelling algorithm

"Labelling algorithm"—what it does:

- **Input:** A CTL formula, φ, and a Kripke structure, $M = (S, s_0, R, L)$ over a set AP.
- **Output:** A set of formulae, $label(s_0)$, that are true in s_0 (i.e., $M, s_0 \models \varphi$ iff $\varphi \in label(s)$).

CTL model checking—labelling algorithm

"Labelling algorithm"—what it does:

- **Input:** A CTL formula, φ, and a Kripke structure, $M = (S, s_0, R, L)$ over a set AP.
- **Output:** A set of formulae, $label(s_0)$, that are true in s_0 (i.e., $M, s_0 \models \varphi$ iff $\varphi \in label(s)$).
- Initially, $label(s_0) = L(s_0)$; algorithm goes through states, at stage i, CTL subformulae with $i - 1$ nested temporal operators are processed.
- When a formula is processed it is added to the labelling of those states where it is true.

CTL model checking—labelling algorithm

By strucutral induction[1] (that is, algorithm starts with innermost formulae and works its way "outwards"):

- $\Phi = \neg\varphi$: label all states with Φ that are not labelled by φ.
- $\Phi = \varphi \vee \psi$: label all states with Φ that are labelled by either φ or ψ.
- $\Phi = \mathbf{EX}\varphi$: label all states with Φ that have a successor labelled by φ.
- $\Phi = \mathbf{E}(\varphi \mathbf{U} \psi)$: find all states labelled by ψ; then work backwards until you hit a state labelled by φ; all intermediate states on these paths should be labelled by Φ.

[1] Only few cases due to earlier corollary!

CTL model checking—labelling algorithm

procedure $CheckEU(f_1, f_2)$
    $T := \{\, s \mid f_2 \in label(s) \,\}$;
    for all $s \in T$ **do** $label(s) := label(s) \cup \{\, \mathbf{E}[f_1\ \mathbf{U}\ f_2] \,\}$;
    while $T \neq \emptyset$ **do**
        choose $s \in T$;
        $T := T \setminus \{s\}$;
        for all t **such that** $R(t, s)$ **do**
            if $\mathbf{E}[f_1\ \mathbf{U}\ f_2] \notin label(t)$ **and** $f_1 \in label(t)$ **then**
                $label(t) := label(t) \cup \{\, \mathbf{E}[f_1\ \mathbf{U}\ f_2] \,\}$;
                $T := T \cup \{t\}$;
            end if;
        end for all;
    end while;
end procedure

Runs in $O(|S| + |R|)$.

CTL model checking—labelling algorithm

- $\Phi = \mathbf{EG}\varphi$ slightly more complicated; needs notion of SCC:
  - First create $M' = (S', s'_0, R', L')$, where
    - $S' = \{s \in S' \mid M, s \models \varphi\}$ (i.e., remove all nodes from M, where φ does not hold)
    - $R' = R|_{S' \times S'}$
    - $L' = L|_{S'}$

CTL model checking—labelling algorithm

- $\Phi = \mathbf{EG}\varphi$ slightly more complicated; needs notion of SCC:
  - First create $M' = (S', s'_0, R', L')$, where
    - $S' = \{s \in S' \mid M, s \models \varphi\}$ (i.e., remove all nodes from M, where φ does not hold)
    - $R' = R|_{S' \times S'}$
    - $L' = L|_{S'}$

Lemma

$M, s \models \mathbf{EG}\varphi$ iff the following two conditions are satisfied:

1. $s \in S'$
2. *There is a path in M', starting in s, to some node t in some SCC of graph (S', R').*

CTL model checking—labelling algorithm

Proof.

($\Rightarrow$) As for 1.: Clearly, $s \in S'$.
Now we need to show 2. Let $w' = uw$ be a path in M such that φ is true in each state. u is the prefix and w the infinite suffix. For w to repeat, it must lie inside a SCC. And since φ is true along the path, we have for u and w that they're both contained in S' by the construction of M'.

($\Leftarrow$) Every path that in M' is also a path in M. And if there is a path that loops infinitely through some SCC, and on which φ holds, then it is a model for $\mathbf{EG}\varphi$. Since the initial state of that path, $s \in S'$ is clearly also in S, the lemma follows. $\square$

CTL model checking—labelling algorithm

procedure $CheckEG(f_1)$
  $S' := \{\, s \mid f_1 \in label(s) \,\}$;
  $SCC := \{\, C \mid C$ is a nontrivial SCC of $S' \,\}$;
  $T := \bigcup_{C \in SCC} \{\, s \mid s \in C \,\}$;
  for all $s \in T$ **do** $label(s) := label(s) \cup \{\, \mathbf{EG}\ f_1 \,\}$;
  while $T \neq \emptyset$ **do**
    choose $s \in T$;
    $T := T \setminus \{s\}$;
    for all t **such that** $t \in S'$ **and** $R(t, s)$ **do**
      if $\mathbf{EG}\ f_1 \notin label(t)$ **then**
        $label(t) := label(t) \cup \{\, \mathbf{EG}\ f_1 \,\}$;
        $T := T \cup \{t\}$;
      end if;
    end for all;
  end while;
end procedure

Runs in $O(|S'| + |R'|)$.

CTL model checking—labelling algorithm

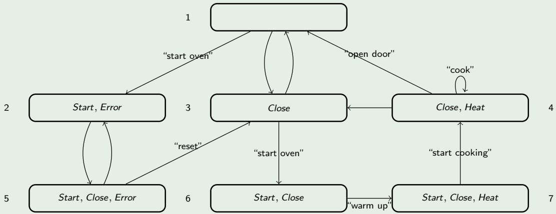
Since we have at most $|\varphi|$ subformulae, CTL model checking against a Kripke structure takes time $O(|\varphi| \cdot (|S| + |R|))$.

Theorem

To decide the CTL model checking problem one only needs an algorithm that runs in PTime.

CTL model checking—example

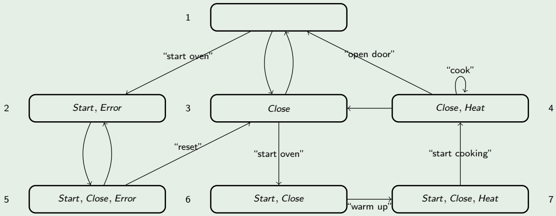
Same Kripke structure we used earlier:



, 1 $\models$ $\mathbf{AG}(\mathit{Start} \rightarrow \mathbf{AF}\mathit{Heat})$?

CTL model checking—example

Same Kripke structure we used earlier:



, 1 $\models$ $\mathbf{AG}(\mathit{Start} \rightarrow \mathbf{AF}\mathit{Heat})$?

Observe:

- $\mathbf{AG}(\mathit{Start} \rightarrow \mathbf{AF}\mathit{Heat})$ equiv. to $\neg\mathbf{EF}(\mathit{Start} \wedge \mathbf{EG}\neg\mathit{Heat})$
- We use $\mathbf{EF}\varphi$ as shorthand for $\mathbf{E}(\mathit{true}\mathbf{U}\varphi)$.

CTL model checking—example

How the algorithm proceeds:

- Let $S(\psi)$ be the set of states in which ψ holds.
- Initially, $S(\mathit{Start}) = \{2, 5, 6, 7\}$, $S(\neg \mathit{Heat}) = \{1, 2, 3, 5, 6\}$.
- For $S(\mathbf{EG}\neg \mathit{Heat})$ we first find SCCs wrt. $\neg \mathit{Heat}$.

[2] not 6, because you can reach 7 from 6, where *Heat* is true

CTL model checking—example

How the algorithm proceeds:

- Let $S(\psi)$ be the set of states in which ψ holds.
- Initially, $S(\mathit{Start}) = \{2, 5, 6, 7\}$, $S(\neg \mathit{Heat}) = \{1, 2, 3, 5, 6\}$.
- For $S(\mathbf{EG}\neg \mathit{Heat})$ we first find SCCs wrt. $\neg \mathit{Heat}$. I.e., $S' = \{1, 2, 3, 5, 6\}$, and SCC in S' is $\{1, 2, 3, 5\} = S(\mathbf{EG}\neg \mathit{Heat})$[2]

[2] not 6, because you can reach 7 from 6, where *Heat* is true

CTL model checking—example

How the algorithm proceeds:

- Let $S(\psi)$ be the set of states in which ψ holds.
- Initially, $S(Start) = \{2, 5, 6, 7\}$, $S(\neg Heat) = \{1, 2, 3, 5, 6\}$.
- For $S(\mathbf{EG}\neg Heat)$ we first find SCCs wrt. $\neg Heat$. I.e., $S' = \{1, 2, 3, 5, 6\}$, and SCC in S' is $\{1, 2, 3, 5\} = S(\mathbf{EG}\neg Heat)$[2]
- $S(Start \wedge \mathbf{EG}\neg Heat)$

[2] not 6, because you can reach 7 from 6, where *Heat* is true

CTL model checking—example

How the algorithm proceeds:

- Let $S(\psi)$ be the set of states in which ψ holds.
- Initially, $S(\mathit{Start}) = \{2, 5, 6, 7\}$, $S(\neg \mathit{Heat}) = \{1, 2, 3, 5, 6\}$.
- For $S(\mathbf{EG}\neg \mathit{Heat})$ we first find SCCs wrt. $\neg \mathit{Heat}$. I.e., $S' = \{1, 2, 3, 5, 6\}$, and SCC in S' is $\{1, 2, 3, 5\} = S(\mathbf{EG}\neg \mathit{Heat})$[2]
- $S(\mathit{Start} \wedge \mathbf{EG}\neg \mathit{Heat}) = \{2, 5\}$.
- To compute $S(\mathbf{EF}(\mathit{Start} \wedge \mathbf{EG}\neg \mathit{Heat})$, set $T = S(\mathbf{EG}\neg \mathit{Heat})$ and find all states from which states from T can be reached,

[2] not 6, because you can reach 7 from 6, where *Heat* is true

CTL model checking—example

How the algorithm proceeds:

- Let $S(\psi)$ be the set of states in which ψ holds.
- Initially, $S(Start) = \{2, 5, 6, 7\}$, $S(\neg Heat) = \{1, 2, 3, 5, 6\}$.
- For $S(\mathbf{EG}\neg Heat)$ we first find SCCs wrt. $\neg Heat$. I.e., $S' = \{1, 2, 3, 5, 6\}$, and SCC in S' is $\{1, 2, 3, 5\} = S(\mathbf{EG}\neg Heat)$[2]
- $S(Start \wedge \mathbf{EG}\neg Heat) = \{2, 5\}$.
- To compute $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)$, set $T = S(\mathbf{EG}\neg Heat)$ and find all states from which states from T can be reached, i.e., $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat) = S$.

[2]not 6, because you can reach 7 from 6, where *Heat* is true

CTL model checking—example

How the algorithm proceeds:

- Let $S(\psi)$ be the set of states in which ψ holds.
- Initially, $S(Start) = \{2,5,6,7\}$, $S(\neg Heat) = \{1,2,3,5,6\}$.
- For $S(\mathbf{EG}\neg Heat)$ we first find SCCs wrt. $\neg Heat$. I.e., $S' = \{1,2,3,5,6\}$, and SCC in S' is $\{1,2,3,5\} = S(\mathbf{EG}\neg Heat)$[2]
- $S(Start \wedge \mathbf{EG}\neg Heat) = \{2,5\}$.
- To compute $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)$, set $T = S(\mathbf{EG}\neg Heat)$ and find all states from which states from T can be reached, i.e., $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat) = S$.
- Finally, $S(\neg\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat) = \overline{S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)} = \emptyset$.

[2] not 6, because you can reach 7 from 6, where *Heat* is true

CTL model checking—example

How the algorithm proceeds:

- Let $S(\psi)$ be the set of states in which ψ holds.
- Initially, $S(Start) = \{2, 5, 6, 7\}$, $S(\neg Heat) = \{1, 2, 3, 5, 6\}$.
- For $S(\mathbf{EG}\neg Heat)$ we first find SCCs wrt. $\neg Heat$. I.e., $S' = \{1, 2, 3, 5, 6\}$, and SCC in S' is $\{1, 2, 3, 5\} = S(\mathbf{EG}\neg Heat)$[2]
- $S(Start \wedge \mathbf{EG}\neg Heat) = \{2, 5\}$.
- To compute $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)$, set $T = S(\mathbf{EG}\neg Heat)$ and find all states from which states from T can be reached, i.e., $S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat) = S$.
- Finally, $S(\neg\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat) = \overline{S(\mathbf{EF}(Start \wedge \mathbf{EG}\neg Heat)} = \emptyset$.
- Property does not hold. :-(

[2] not 6, because you can reach 7 from 6, where *Heat* is true

Binary decision diagrams

- Popular data structure for compactly and uniquely representing Boolean functions.
- Efficient algorithms known to manipulate BDDs according to the operations in Boolean logic.
- **Applications:** there are many! In our context: to compactly represent Kripke structures.

Binary decision diagrams

Let $x \to y_0, y_1$ be the if-then-else operator defined by

$$x \to y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

All other Boolean operations can be expressed in terms of this operator:

- $\neg x =$

Binary decision diagrams

Let $x \to y_0, y_1$ be the if-then-else operator defined by

$$x \to y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

All other Boolean operations can be expressed in terms of this operator:

- $\neg x = (x \to 0, 1)$
- $x \Leftrightarrow y =$

Binary decision diagrams

Let $x \to y_0, y_1$ be the if-then-else operator defined by

$$x \to y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

All other Boolean operations can be expressed in terms of this operator:

- $\neg x = (x \to 0, 1)$
- $x \Leftrightarrow y = x \to (y \to 1, 0), (y \to 0, 1)$
- etc.

Binary decision diagrams

Let $x \to y_0, y_1$ be the if-then-else operator defined by

$$x \to y_0, y_1 = (x \wedge y_0) \vee (\neg x \wedge y_1)$$

All other Boolean operations can be expressed in terms of this operator:

- $\neg x = (x \to 0, 1)$
- $x \Leftrightarrow y = x \to (y \to 1, 0), (y \to 0, 1)$
- etc.

Definition

The ITE-normal form (INF) is a Boolean expression built entirely from the ITE-operator. (You may have heard of other normal forms.)

Binary decision diagrams—how to obtain INF?

Definition

Shannon expansion: Given Boolean expression t,

$$t = x \to t[1/x], t[0/x] \quad (\text{"Shannon expansion of } t \text{ wrt. } x\text{"}).$$

- If t contains no variables, it is equivalent to 0 or 1, i.e., in INF.
- Otherwise, perform Shannon expansion of t wrt. any of its variables x.
- Since $t[0/x]$ and $t[1/x]$ contain one variable less than t, one can recursively find INFs for both of these new terms; call them t_0 and t_1.
- INF for t is thus $x \to t_1, t_0$.

Binary decision diagrams—how to obtain INF?

Theorem

Any Boolean expression is equivalent to an expression in INF.

Proof.

See inductive INF construction. □

Binary decision diagrams—how to obtain INF?

Example: $t = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$

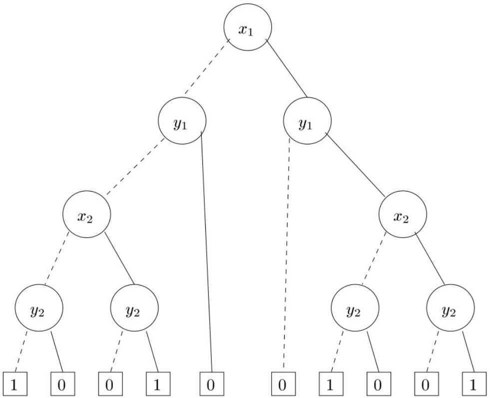
Perform SE on variables ordered by x_1, y_1, x_2, y_2, then

$$\begin{array}{rcl}
t & = & x_1 \rightarrow t_1, t_0 \\
t_0 & = & y_1 \rightarrow 0, t_{00} \\
t_1 & = & y_1 \rightarrow t_{11}, 0 \\
t_{00} & = & x_2 \rightarrow t_{001}, t_{000} \\
t_{11} & = & x_2 \rightarrow t_{111}, t_{110} \\
t_{000} & = & y_2 \rightarrow 0, 1 \\
t_{001} & = & y_2 \rightarrow 1, 0 \\
t_{110} & = & y_2 \rightarrow 0, 1 \\
t_{111} & = & y_2 \rightarrow 1, 0
\end{array}$$

Binary decision diagrams—how to obtain BDD?

Example: $t = (x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$

Corresponding binary decision tree:



(Source: Henrik Reif Andersen's lecture notes.)

Binary decision diagrams—how to obtain BDD?

Consider again:

$$
\begin{aligned}
t &= x_1 \to t_1, t_0 \\
t_0 &= y_1 \to 0, t_{00} \\
t_1 &= y_1 \to t_{11}, 0 \\
t_{00} &= x_2 \to t_{001}, t_{000} \\
t_{11} &= x_2 \to t_{111}, t_{110} \\
t_{000} &= y_2 \to 0, 1 \\
t_{001} &= y_2 \to 1, 0 \\
t_{110} &= y_2 \to 0, 1 \\
t_{111} &= y_2 \to 1, 0
\end{aligned}
$$

Note:

- Instead of t_{110} we could use t_{000}.
- Substitute t_{110} for t_{000} on RHS of t_{11}.

Binary decision diagrams—how to obtain BDD?

$$\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{111}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0 \\
t_{111} &= y_2 \rightarrow 1, 0
\end{aligned}$$

Binary decision diagrams—how to obtain BDD?

$$
\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{001/111}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0 \\
t_{111} &= y_2 \rightarrow 1, 0
\end{aligned}
$$

Binary decision diagrams—how to obtain BDD?

$$
\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{00/11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0
\end{aligned}
$$

Binary decision diagrams—how to obtain BDD?

$$
\begin{array}{rcl}
t & = & x_1 \to t_1, t_0 \\
t_0 & = & y_1 \to 0, t_{00} \\
t_1 & = & y_1 \to t_{00}, 0 \\
t_{00} & = & x_2 \to t_{001}, t_{000} \\
t_{000} & = & y_2 \to 0, 1 \\
t_{001} & = & y_2 \to 1, 0
\end{array}
$$

Binary decision diagrams—how to obtain BDD?

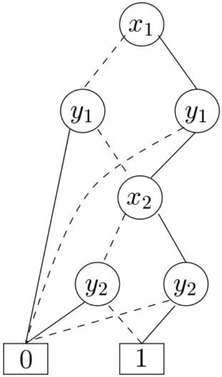
$$
\begin{aligned}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{00}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0
\end{aligned}
$$

Let us now view each subexpression as a node of a graph, where 0 and 1 are the only "terminal" nodes:

Binary decision diagrams—how to obtain BDD?

$$
\begin{aligned}
t &= x_1 \to t_1, t_0 \\
t_0 &= y_1 \to 0, t_{00} \\
t_1 &= y_1 \to t_{00}, 0 \\
t_{00} &= x_2 \to t_{001}, t_{000} \\
t_{000} &= y_2 \to 0, 1 \\
t_{001} &= y_2 \to 1, 0
\end{aligned}
$$

Let us now view each subexpression as a node of a graph, where 0 and 1 are the only "terminal" nodes:



Binary decision diagrams

Definition

A BDD is a rooted, directed acyclic graph (DAG) with

- one or two terminal nodes of out-degree zero labeled 0 or 1 and,
- a set of variable nodes u of out-degree two. The two outgoing edges are given by two functions $low(u)$ and $high(u)$. (In pictures, these are shown as dotted and solid lines, respectively). A variable $var(u)$ is associated with each variable node.

Binary decision diagrams

Definition

A BDD is ordered (OBDD) if on all paths through the graph the variables respect a given linear order $x_1 < x_2 < \ldots < x_n$. An OBDD is reduced if

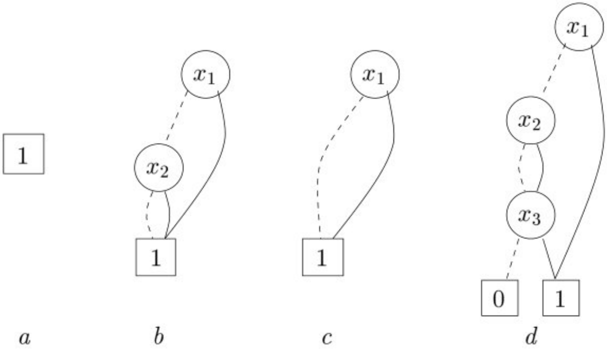
- (uniqueness) no two distinct nodes u and v have the same variable name and low- and high-successor, i.e.,

  $$var(u) = var(v), low(u) = low(v), high(u) = high(v) \Rightarrow u = v$$

- (no redundancy) no variable node u has identical low- and high-successor, i.e., $low(u) \neq high(u)$.

Binary decision diagrams

Various OBDDs. Which ones are reduced, which ones are not? What Boolean functions are expressed in those?



Binary decision diagrams

ROBDDs are canonical.

Binary decision diagrams

ROBDDs are canonical.

Let $f : \mathbb{B}^n \to \mathbb{B}$. Nodes u of ROBDD for f inductively define Boolean expressions t^u:

- $t^0 = 0$
- $t^1 = 1$
- $t^u = var(u) \to t^{high(u)}, t^{low(u)}$

Let $x_1 < \ldots < x_n$ be var. ordering, then f^u maps $(b_1, \ldots, b_n) \in \mathbb{B}^n$ to the truth value of $t^u[b_1/x_1, \ldots, b_n/x_n]$.

Binary decision diagrams

ROBDDs are canonical.

Let $f : \mathbb{B}^n \to \mathbb{B}$. Nodes u of ROBDD for f inductively define Boolean expressions t^u:

- $t^0 = 0$
- $t^1 = 1$
- $t^u = var(u) \to t^{high(u)}, t^{low(u)}$

Let $x_1 < \ldots < x_n$ be var. ordering, then f^u maps $(b_1, \ldots, b_n) \in \mathbb{B}^n$ to the truth value of $t^u[b_1/x_1, \ldots, b_n/x_n]$.

Theorem

For any function $f : \mathbb{B}^n \to \mathbb{B}$ there is exactly one ROBDD u with variable ordering $x_1 < x_2 < \ldots < x_n$ s.t. $f^u = f(x_1, \ldots, x_n)$.

Binary decision diagrams

Proof.

By induction (cf. Andersen lecture notes p. 13f.). □

Binary decision diagrams

What to do with ROBDDs? Let $f, g : \mathbb{B}^n \rightarrow \mathbb{B}$

- How do you check validity of f if given as ROBDD?

Binary decision diagrams

What to do with ROBDDs? Let $f, g : \mathbb{B}^n \rightarrow \mathbb{B}$

- How do you check validity of f if given as ROBDD? (compare to non-terminal node; $O(1)$ vs coNP for formulae)

Binary decision diagrams

What to do with ROBDDs? Let $f, g : \mathbb{B}^n \to \mathbb{B}$

- How do you check validity of f if given as ROBDD? (compare to non-terminal node; $O(1)$ vs coNP for formulae)
- How do you check equivalence of f and g if given as ROBDDs?

Binary decision diagrams

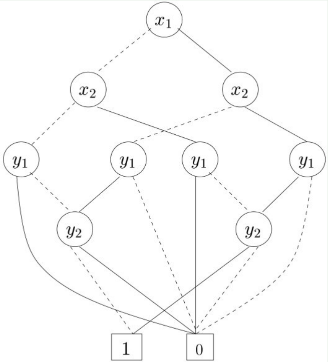
What to do with ROBDDs? Let $f, g : \mathbb{B}^n \to \mathbb{B}$

- How do you check validity of f if given as ROBDD? (compare to non-terminal node; $O(1)$ vs coNP for formulae)
- How do you check equivalence of f and g if given as ROBDDs? (compare nodes; $O(n)$ vs coNP for formulae)

Binary decision diagrams—variable orderings

Consider ROBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$

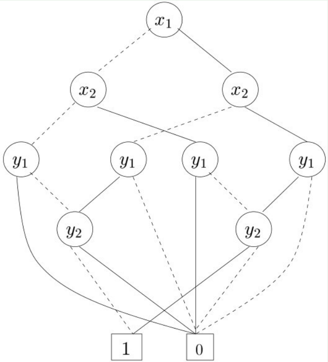
...but different var. ordering of $x_1 < x_2 < y_1 < y_2$:



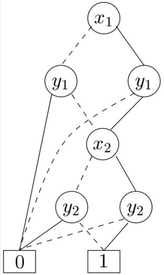
Binary decision diagrams—variable orderings

Consider ROBDD for $(x_1 \Leftrightarrow y_1) \wedge (x_2 \Leftrightarrow y_2)$

...but different var. ordering of $x_1 < x_2 < y_1 < y_2$:



vs



ROBDDs—construction

We saw how to construct OBDD, but how to construct ROBBD?

ROBDDs—construction

We saw how to construct OBDD, but how to construct ROBBD?

- "Construct OBDD and reduce it until you can't anymore."

ROBDDs—construction

We saw how to construct OBDD, but how to construct ROBBD?

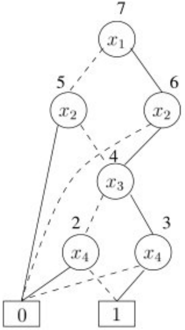
- “Construct OBDD and reduce it until you can’t anymore.”
- **Reduce OBDD on-the-fly (i.e., during construction).**

ROBDDs—construction

- Let $T : u \mapsto (i, l, h)$ be a table which maps every node to an index, a low- and high-index.
- Let $H : (i, l, h) \mapsto u$ be the inverse of T to look up nodes (i.e., $T(u) = (i, l, h)$ iff $H(i, l, h) = u$)

ROBDDs—construction

- Let $T : u \mapsto (i, l, h)$ be a table which maps every node to an index, a low- and high-index.
- Let $H : (i, l, h) \mapsto u$ be the inverse of T to look up nodes (i.e., $T(u) = (i, l, h)$ iff $H(i, l, h) = u$)



$T : u \mapsto (i, l, h)$

u	var	low	$high$
0	5		
1	5		
2	4	1	0
3	4	0	1
4	3	2	3
5	2	4	0
6	2	0	4
7	1	5	6

ROBDDs—construction

Lookup a node i in H and return it, or create new one and return handle to it:

$\textsc{Mk}[T,H](i,l,h)$
1: **if** $l = h$ **then return** l
2: **else if** $member(H,i,l,h)$ **then**
3: **return** $lookup(H,i,l,h)$
4: **else** $u \leftarrow add(T,i,l,h)$
5: $insert(H,i,l,h,u)$
6: **return** u

($MK[T,H]$ means that MK uses data structures T and H.)

ROBDDs—construction

Lookup a node i in H and return it, or create new one and return handle to it:

$\text{MK}[T,H](i,l,h)$
1: **if** $l = h$ **then return** l
2: **else if** $member(H,i,l,h)$ **then**
3: **return** $lookup(H,i,l,h)$
4: **else** $u \leftarrow add(T,i,l,h)$
5: $insert(H,i,l,h,u)$
6: **return** u

($MK[T,H]$ means that MK uses data structures T and H.)

What is the running time of MK?

ROBDDs—construction

Lookup a node i in H and return it, or create new one and return handle to it:

$\textsc{Mk}[T, H](i, l, h)$
1: **if** $l = h$ **then return** l
2: **else if** $member(H, i, l, h)$ **then**
3: **return** $lookup(H, i, l, h)$
4: **else** $u \leftarrow add(T, i, l, h)$
5: $insert(H, i, l, h, u)$
6: **return** u

($MK[T, H]$ means that MK uses data structures T and H.)

What is the running time of MK?

Can be implemented in $O(1)$ using hash tables.

ROBDDs—construction

Input: t be Boolean expression of n var (with fixed var. ordering).
Output: ROBBD of t.

$\text{Build}[T, H](t)$

function $\text{BUILD'}(t, i) =$
  if $i > n$ **then**
    if t is false **then return** 0 **else return** 1
  else $v_0 \leftarrow \text{BUILD'}(t[0/x_i], i + 1)$
    $v_1 \leftarrow \text{BUILD'}(t[1/x_i], i + 1)$
    return $\text{MK}(i, v_0, v_1)$
end BUILD'

return $\text{BUILD'}(t, 1)$

ROBDDs—construction

Input: t be Boolean expression of n var (with fixed var. ordering).
Output: ROBBD of t.

BUILD$[T, H](t)$

    function BUILD'(t, i) =
        if $i > n$ **then**   Evaluation
            if t is false **then return** 0 **else return** 1
        else $v_0 \leftarrow$ BUILD'$(t[0/x_i], i+1)$
            $v_1 \leftarrow$ BUILD'$(t[1/x_i], i+1)$ Shannon expansion
            return MK(i, v_0, v_1)
    end BUILD'

    return BUILD'$(t, 1)$

ROBDDs—construction

Input: t be Boolean expression of n var (with fixed var. ordering).
Output: ROBBD of t.

```
BUILD[T, H](t)
    function BUILD'(t, i) =
        if i > n then                                   Evaluation
            if t is false then return 0 else return 1
        else v0 ← BUILD'(t[0/xi], i + 1)
             v1 ← BUILD'(t[1/xi], i + 1)               Shannon expansion
             return MK(i, v0, v1)
    end BUILD'

    return BUILD'(t, 1)
```

What is the running time of *BUILD*?

ROBDDs—construction

Input: t be Boolean expression of n var (with fixed var. ordering).
Output: ROBBD of t.

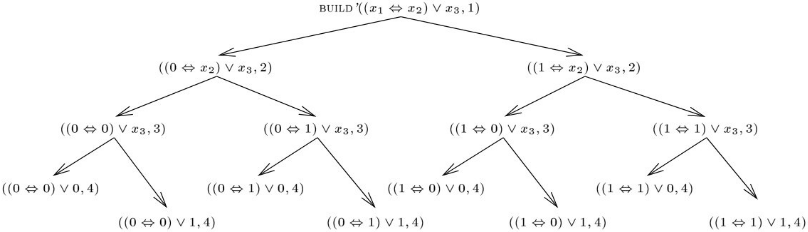
```
BUILD[T, H](t)
    function BUILD'(t, i) =
        if i > n then                 Evaluation
            if t is false then return 0 else return 1
        else v0 ← BUILD'(t[0/x_i], i + 1)    }
             v1 ← BUILD'(t[1/x_i], i + 1)    }  Shannon
             return MK(i, v0, v1)            }  expansion
    end BUILD'

    return BUILD'(t, 1)
```

What is the running time of *BUILD*?

It's bad: $O(2^n)$.

ROBDDs—construction

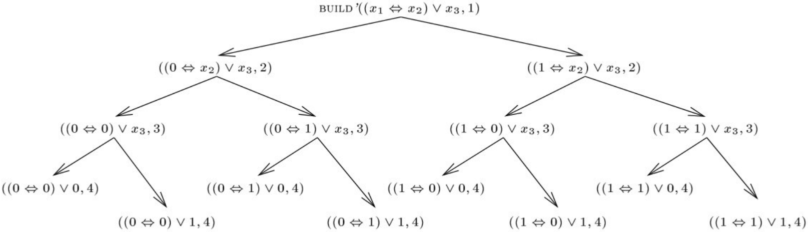
Intuitive explanation for bad running time:
BUILD callgraph on $(x_1 \Leftrightarrow x_2) \vee x_3$:



Can we do better?

ROBDDs—construction

Intuitive explanation for bad running time:
BUILD callgraph on $(x_1 \Leftrightarrow x_2) \vee x_3$:



Can we do better?

One can optimise using divide & conquer, etc. But worst-case no. of calls unavoidable as validity is $O(1)$, yet in coNP for formulae.

ROBDDs—Boolean operations

```
APPLY[T, H](op, u1, u2)
init(G)

function APP(u1, u2) =
  if G(u1, u2) ≠ empty then return G(u1, u2)
  else if u1 ∈ {0, 1} and u2 ∈ {0, 1} then u ← op(u1, u2)
  else if var(u1) = var(u2) then
        u ← MK(var(u1), APP(low(u1), low(u2)), APP(high(u1), high(u2)))
  else if var(u1) < var(u2) then
        u ← MK(var(u1), APP(low(u1), u2), APP(high(u1), u2))
  else (* var(u1) > var(u2) *)
        u ← MK(var(u2), APP(u1, low(u2)), APP(u1, high(u2)))
  G(u1, u2) ← u
  return u
end APP

return APP(u1, u2)
```

Uses Shannon expansion:

- $t = x \rightarrow t[1/x], t[0/x]$
- $(x \rightarrow t_1, t_2)\ op\ (x \rightarrow t_1', t_2') = x \rightarrow t_1\ op\ t_1', t_2\ op\ t_2'$
- $(x \rightarrow t_1, t_2)\ op\ t_3 = x \rightarrow t_1\ op\ t_3, t_2\ op\ t_3$

ROBDDs—SatCount

Task: Count satisfying assignments for ROBBD u

Idea: Given some node, u ...

- determine $\#sat(low(u))$ and $\#sat(high(u))$ first;
- let there be $n \geq 0$ nodes in between u and $low(u)$ (resp. $high(u)$); these n nodes can be assigned truth values arbitrality, but add 2^n more assignments in total, respectively.

ROBDDs—SatCount

Task: Count satisfying assignments for ROBBD u

Idea: Given some node, u ...

- determine $\#sat(low(u))$ and $\#sat(high(u))$ first;
- let there be $n \geq 0$ nodes in between u and $low(u)$ (resp. $high(u)$); these n nodes can be assigned truth values arbitrality, but add 2^n more assignments in total, respectively.

SATCOUNT$[T](u)$

1: **function** $count(u)$
2: **if** $u = 0$ **then** $res \leftarrow 0$
3: **else if** $u = 1$ **then** $res \leftarrow 1$
4: **else** $res \leftarrow 2^{var(low(u))-var(u)-1} * count(low(u)) + 2^{var(high(u))-var(u)-1} * count(high(u))$
5: **return** res
6: **end** $count$

ROBDDs—AnySat & AllSat

$\textsc{AnySat}(u)$

if $u = 0$ **then** Error
else if $u = 1$ **then return** $[\,]$
else if $low(u) = 0$ **then return** $[x_{var(u)} \mapsto 1, \textsc{AnySat}(high(u))]$
else return $[x_{var(u)} \mapsto 0, \textsc{AnySat}(low(u))]$

$\textsc{AllSat}(u)$

if $u = 0$ **then return** $\langle\ \rangle$
else if $u = 1$ **then return** $\langle\ [\ ]\ \rangle$
else return
$\langle$add $[x_{var(u)} \mapsto 0]$ in front of all
truth-assignments in $\textsc{AllSat}(low(u))$,
add $[x_{var(u)} \mapsto 1]$ in front of all
truth-assignments in $\textsc{AllSat}(high(u))\rangle$

ROBDDs—algorithm running times

$\text{MK}(i, u_0, u_1)$	$O(1)$
$\text{BUILD}(t)$	$O(2^n)$
$\text{APPLY}(op, u_1, u_2)$	$O(\|u_1\|\,\|u_2\|)$
$\text{RESTRICT}(u, j, b)$	$O(\|u\|)$
$\text{SATCOUNT}(u)$	$O(\|u\|)$
$\text{ANYSAT}(u)$	$O(\|p\|)$
$\text{ALLSAT}(u)$	$O(\|r\| * n)$
$\text{SIMPLIFY}(d, u)$	$O(\|d\|\|u\|)$

Symbolic model checking—why?/what?

- Typically, one doesn't directly model system in terms of Kripke structure.

Symbolic model checking—why?/what?

- Typically, one doesn't directly model system in terms of Kripke structure.
- Translation of system model $\mathcal{M} \to M$ (cf. on-the-fly alg.)

Symbolic model checking—why?/what?

- Typically, one doesn't directly model system in terms of Kripke structure.
- Translation of system model $\mathcal{M} \rightarrow M$ (cf. on-the-fly alg.)
- However, M can be huge! (State explosion.)

Symbolic model checking—why?/what?

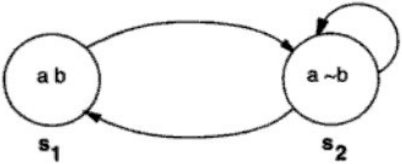
- Typically, one doesn't directly model system in terms of Kripke structure.
- Translation of system model $\mathcal{M} \to M$ (cf. on-the-fly alg.)
- However, M can be huge! (State explosion.)
- Represent states/transition system of M symbolically using ROBDDs (i.e., one ROBDD encodes multiple states/transitions of M).

Symbolic model checking—why?/what?

- Typically, one doesn't directly model system in terms of Kripke structure.
- Translation of system model $\mathcal{M} \to M$ (cf. on-the-fly alg.)
- However, M can be huge! (State explosion.)
- Represent states/transition system of M symbolically using ROBDDs (i.e., one ROBDD encodes multiple states/transitions of M).
- Expand state space inductively in a stepwise manner using ROBDD operations.

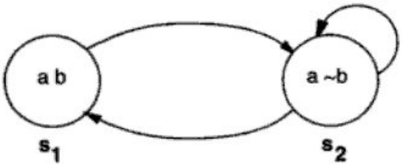
Symbolic model checking—basic idea

- For example:



Symbolic model checking—basic idea

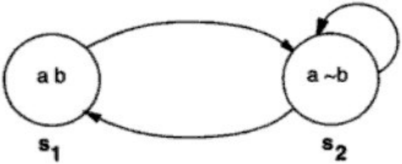
- For example:



- Transition $s_1 \rightarrow s_2$ is $a \wedge b \wedge a' \wedge \neg b'$

Symbolic model checking—basic idea

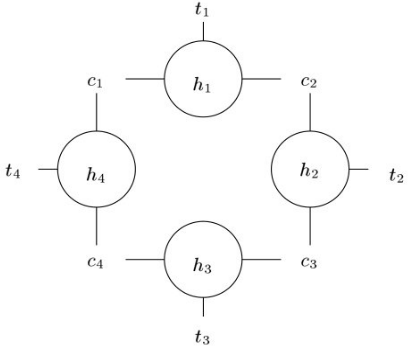
- For example:



- Transition $s_1 \to s_2$ is $a \wedge b \wedge a' \wedge \neg b'$
- Whole TS:
  $(a \wedge b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge \neg b') \vee (a \wedge \neg b \wedge a' \wedge b')$

Symbolic model checking—example

Milner's scheduler:



- $t_i = 1$ iff task i is running
- $h_i = 1$ iff task i has token
- $c_i = 1$ iff task $i - 1$ has released token (and i not picked it up yet)

Scheduler job: start at task 1, and schedule all tasks such that all are executed. Tasks can terminate in any order.

Symbolic model checking—example

- Each task can be described as an individual state-transition system over variables t_i, h_i, c_i, respectively.
- First, formalise behaviour:
  - if $c_i = 1 \wedge t_i = 0$ then $t_i, c_i, h_i := 1, 0, 1$
  - if $h_i = 1$ then $c_{(i \bmod N)+1}, h_i := 1, 0$

S subset of unprimed vars. Useful to state something about vars that changed:

$$unchanged_S = \bigwedge_{x \in S} x = x'$$

(Or, $assigned_{S'} = unchanged_{\vec{x} \setminus S'}$, i.e., all vars not in S' are unchanged.)

Symbolic model checking—example

We can now define P_i, the transitions of task i over the vars $\vec{x}, \vec{x'}$ as:

$$
\begin{aligned}
P_i = \quad & (c_i \wedge \neg t_i \wedge t_i' \wedge \neg c_i' \wedge h_i' \wedge assigned_{\{c_i, t_i, h_i\}}) \\
& \vee (h_i \wedge c'_{(i \mod N)+1} \wedge \neg h_i' \wedge assigned_{\{(c_i \mod N)+1, h_i\}})
\end{aligned}
$$

Termination of task:

$$E_i = t_i \wedge \neg t_i' \wedge assigned_{\{t_i\}}$$

All possible transitions:

$$T = P_1 \vee \ldots \vee P_n \vee E_1 \vee \ldots \vee E_n$$

Initial state (only c_1 has token):

$$I = \neg \vec{t} \wedge \neg \vec{h} \wedge c_1 \wedge \neg c_2 \wedge \ldots \wedge \neg c_N$$

Symbolic model checking—example

We can now start asking questions like

- Is it the case that all reachable states only ever have one token?
- Is task t_i always scheduled after t_{i-1}?
- Deadlock: can we reach a state where no more transitions can be taken?
- ...

Symbolic model checking—example

We can now start asking questions like

- Is it the case that all reachable states only ever have one token?
- Is task t_i always scheduled after t_{i-1}?
- Deadlock: can we reach a state where no more transitions can be taken?
- ...

Need to compute predicate over the unprimed vars, R, characterising exactly the set of states reachable from I.

Symbolic model checking—how to compute R

Some observations:

- R needs to satisfiy I or within finite number of transitions can be reached from I.

Symbolic model checking—how to compute R

Some observations:

- R needs to satisfiy I or within finite number of transitions can be reached from I.
- Suggests iterative process: R^0, R^1, ...

Symbolic model checking—how to compute R

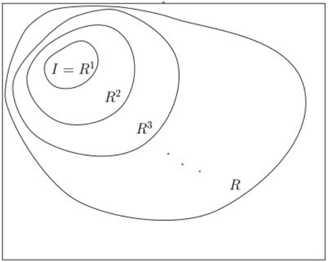
Some observations:

- R needs to satisfiy I or within finite number of transitions can be reached from I.
- Suggests iterative process: R^0, R^1, ...
- Let $R^0 = 0$ and compute R^{k+1} as disjunction of I and the set of states reachable from R^k.

Symbolic model checking—how to compute R

Some observations:

- R needs to satisfiy I or within finite number of transitions can be reached from I.
- Suggests iterative process: R^0, R^1, ...
- Let $R^0 = 0$ and compute R^{k+1} as disjunction of I and the set of states reachable from R^k.



REACHABLE-STATES$(I, T, \vec{x}, \vec{x}')$

1: $R \leftarrow \boxed{0}$
2: **repeat**
3: $\quad R' \leftarrow R$
4: $\quad R \leftarrow I \vee (\exists \vec{x}.\ T \wedge R)[\vec{x}/\vec{x}']$
5: **until** $R' = R$
6: **return** R