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 $\varphi$ (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 $\varphi$ (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 $\varphi$, what is “satisfy”?
By the way…

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

  ![Image of award ceremony](image-url)

- 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)

(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

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

1. "start oven" -> "open door" -> "cook" -> "start cooking"
2. Start, Error
3. Close
4. Close, Heat
5. Start, Close, Error
6. Start, Close
7. Start, Close, Heat

Possible behaviour of microwave oven:
- Trace/word: {Close}, {Start, Close}, {Start, Close, Heat}
- Actions: "start oven", "open door", "reset", "start oven", "warm up", "cook", "start cooking"
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 $\Sigma = 2^\text{AP}$ be a finite alphabet. Let $\Sigma^\omega$ denote set of all infinite traces over $\Sigma$. 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: \( P = m : \text{cobegin } P_0 || P_1 \text{ coend } m' \)
Kripke structures—where they come from

The corresponding Kripke structure

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)

- \( \varphi = a \land \neg b \lor c \) 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 \( \varphi \).”)
Linear-time temporal logic

LTL syntax

- Every propositional logic formula is also an LTL formula.
- If \( \varphi \) is an LTL formula, then so are \( X\varphi \) and \( \varphi U \varphi' \).
- BNF: \( \varphi ::= p \in AP \mid \neg \varphi \mid \varphi \land \varphi \mid X\varphi \mid \varphi U \varphi \).
Linear-time temporal logic

**LTL syntax**

- Every propositional logic formula is also an LTL formula.
- If $\varphi$ is an LTL formula, then so are $X\varphi$ and $\varphi U \varphi'$.
- BNF: $\varphi ::= p \in AP | \neg \varphi | \varphi \land \varphi | X\varphi | \varphi U \varphi$.

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

<table>
<thead>
<tr>
<th>$w, i \models p$</th>
<th>iff</th>
<th>$p \in w(i)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$w, i \models \neg \varphi$</td>
<td>iff</td>
<td>$w, i \models \varphi$ is not true</td>
</tr>
<tr>
<td>$w, i \models \varphi \land \psi$</td>
<td>iff</td>
<td>$w, i \models \varphi$ and $w, i \models \psi$</td>
</tr>
<tr>
<td>$w, i \models X\varphi$</td>
<td>iff</td>
<td>$w, i + 1 \models \varphi$</td>
</tr>
<tr>
<td>$w, i \models \varphi U \psi$</td>
<td>iff</td>
<td>there is $k \geq i$ s.t. $w, k \models \psi$, and for all $i \leq j &lt; k$ we have $w, j \models \varphi$</td>
</tr>
</tbody>
</table>

More generally, note how models of $\varphi \in LTL$ are elements from $\Sigma^\omega$ ($\Sigma = 2^{AP}$ is our alphabet).

Let $L(\varphi) = \{w \in \Sigma^\omega \mid w, 0 \models \varphi\}$ be the **language** of $\varphi$. 

Andreas Bauer

NICTA & ANU

COMP4600 Advanced algorithms: Algorithms for verification

13 / 83
Linear-time temporal logic

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

- $true = p \lor \neg p$
- $false = \neg true$
- $\varphi \lor \psi = \neg (\neg \varphi \land \neg \psi)$
- $\varphi \rightarrow \psi = \neg \varphi \lor \psi$
- $\varphi \leftrightarrow \psi = (\varphi \rightarrow \psi) \land (\psi \rightarrow \varphi)$
- $F\varphi = true U \varphi$ (“eventually $\varphi$”)
- $G\varphi = \neg F \neg \varphi$ (“always $\varphi$”)
- $\varphi R \psi = \neg (\neg \varphi U \neg \psi)$ (“release $\psi$ when $\varphi$ becomes true”)

Andreas Bauer

COMP4600 Advanced algorithms: Algorithms for verification

14 / 83
Linear-time temporal logic

Some LTL specifications:

Invariants:
- $G \neg (crit_1 \land crit_2)$ (mutual exclusion)
- $G (preset_1 \lor \ldots \lor preset_n)$ (deadlock freedom)

Response, recurrence:
- $G (try_1 \rightarrow F crit_1)$ (eventual access to critical section)
- $GF \neg crit_1$ (no starvation in critical section)

Strong fairness:
- $GF (try_1 \land \neg crit_2) \rightarrow GF crit_1$ (strong fairness)
LTL model checking

Is the following decision problem:

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

Example: Microwave oven

$\mathcal{L}(M) \subseteq \mathcal{L}(G(Heat \to Close))$
LTL model checking

Key ideas:

- \( L(M) \subseteq L(\varphi) \iff L(M) \cap L(\neg\varphi) = \emptyset \)
- If \( L(M) \cap L(\neg\varphi) \neq \emptyset \), we have a counterexample.
LTL model checking

Key ideas:

- \( L(M) \subseteq L(\varphi) \iff L(M) \cap L(\neg \varphi) = \emptyset \)
- If \( L(M) \cap L(\neg \varphi) \neq \emptyset \), we have a counterexample.

How do we test if \( L(M) \cap L(\neg \varphi) = \emptyset \)?
LTL model checking

Theorem

For every \( \varphi \in LTL \), there exists an \( \omega \)-automaton, \( A \), s.t.,
\[
L(A) = L(\varphi).
\]
LTL model checking

**Theorem**

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

**Corollary**

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

LTL model checking

**Theorem**

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

**Corollary**

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

Note that, $M \times A_{\neg \varphi}$ is normally too big to be explicitly computed (but we disregard that fact for now).
LTL model checking—$\omega$-automata

Definition

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

- $\Sigma$ 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
- $F$ an acceptance condition.

A run $\rho$ of $A$ 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—$\omega$-automata

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

- $F_i \subseteq Q$ is an accepting set.
- $\rho$ is accepting iff $\text{Inf}(\rho) \cap F_i \neq \emptyset$ 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.
- \( \rho \) is accepting iff \( \text{Inf}(\rho) \cap F_i \neq \emptyset \) 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.
- \( \rho \) is accepting iff \( \text{Inf}(\rho) \cap F_i \neq \emptyset \) 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.
- \( \rho \) is accepting iff \( \text{Inf}(\rho) \cap F \neq \emptyset \).

Streett automaton: \( \mathcal{F} = \{(E_1, F_1), \ldots, (E_n, F_n)\} \)
- \( E_i, F_i \subseteq Q \).
- \( \rho \) is accepting iff \( \text{Inf}(\rho) \cap F_i \neq \emptyset \rightarrow \text{Inf}(\rho) \cap E_i \neq \emptyset \) for \( 1 \leq i \leq n \).
LTL model checking—\(\omega\)-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 \(\delta\) is, in fact, a function.
**Recall:** An automaton is deterministic iff for all \( q \in Q \), and \( \sigma \in \Sigma \), \( \delta(q, \sigma) \) is a singleton; that is, if \( \delta \) is, in fact, a function.

**Theorem**

*NBAs are strictly more expressive than DBAs.*
LTL model checking—\(\omega\)-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 \(\delta\) 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.
Recall: An automaton is deterministic iff for all \( q \in Q \), and \( \sigma \in \Sigma \), \( \delta(q, \sigma) \) is a singleton; that is, if \( \delta \) is, in fact, a function.

**Theorem**

NBAs are strictly more expressive than DBAs.

**Proof.**

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

**Theorem**

NBAs can encode every LTL property, but not vice versa.
Recall: An automaton is deterministic iff for all \( q \in Q \), and \( \sigma \in \Sigma \), \( \delta(q, \sigma) \) is a singleton; that is, if \( \delta \) is, in fact, a function.

**Theorem**

*NBAs are strictly more expressive than DBAs.*

**Proof.**

\[ L = \mathcal{L}((a + b)^* a^\omega) \] is NBA-definable 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 \( \varphi \), \( cl(\varphi) \), consists of all subformulas of \( \psi \) of \( \varphi \) and their negation \( \neg \psi \).
Definition

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

Example: $\varphi = aU(\neg a \land b)$
**Definition**

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

**Example:** $\varphi = a U (\neg a \land b)$

$cl(\varphi) = \{a, b, \neg a, \neg b, \neg a \land b, \neg(\neg a \land b), \varphi, \neg\varphi\}$
LTL-to-automata translation

GBA for $\varphi \in LTL$:

- **Q**: elements of $cl(\varphi)$, promised to be true.
- **$Q_0$**: states containing $\varphi$.
- **$\delta$**: 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 $\varphi' = \psi \land \psi' \in cl(\varphi)$ we have that $\varphi' \in c$ iff $\psi \in c$ and $\psi' \in c$.)
  - $(c, d) \in E$ iff
    - for any $\varphi' = \psi U \psi' \in cl(\varphi)$, $\varphi' \in c$ iff either $\psi' \in c$, or $\psi \in c$ and $\varphi' \in d$;
    - for any $\varphi' = X \psi \in cl(\varphi)$, $\varphi' \in c$ iff $\psi \in d$.
- $\mathcal{F} = \{ \{q \in Q \mid \psi U \psi' \not\in q \text{ or } \psi' \in q\} \mid \psi U \psi' \in cl(\varphi)\}$
LTL-to-automata translation—complexity considerations

How big is $|Q|$ (resp. $A_\varphi$) at most?
LTL-to-automata translation—complexity considerations

How big is $|Q|$ (resp. $A_\varphi$) at most?

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

How big is $|Q|$ (resp. $A_\varphi$) at most?

- $|\text{cl}(\varphi)| = O(|\varphi|)$.
- There are at most $2^{O(|\varphi|)}$ many possible subsets of $\text{cl}(\varphi)$. 
How big is $|Q|$ (resp. $A_\varphi$) 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 $\mathcal{L}(M \times A_{\neg \varphi}) = \emptyset$ rather than $\mathcal{L}(M) \cap \mathcal{L}(A_\varphi) = \emptyset$:
  - Complementation of formula $O(1)$ vs.
  - complementation of automaton $\approx O(2^{\vert Q\vert})$. 

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 \(i\)th 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—optimisation

More formally:

From $G = (\Sigma, Q, Q_0, \delta, F)$, we construct $B = (\Sigma, Q', Q'_0, \delta', F')$:

$q' = Q \times \{1, \ldots, n\}$

$\delta' \subseteq Q' \times Q'$, where $(q_i, s_j) \in \delta'$ if $q \not\in F_i$ and $i = j$, or $q \in F_i$ and $j = (i + 1) \mod n$.

$Q'_0 = \{ (q_0, 0) \mid q_0 \in Q_0 \}$

$F'_0 = \{ (q_0, 0) \mid q_0 \in F_0 \}$

Edge-labelled vs. state-labelled NBA:
Both used; arguably, edge-labelled more common.
Easy translation between the two models.
LTL-to-automata translation—optimisations

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

- \( Q' = Q \times \{1, \ldots, n\} \)
- \( \delta' \subseteq Q' \times Q' \), 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 $q \not\in F_i$ 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.
Some example NBAs: (w/o redundant states)

\[ Xa: \]
\[ aU b: \]
\[ GF a: \]

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!)*
Let $A$ be an NBA over $\Sigma$.

- $L(A) = / \neq \emptyset$?
Important properties of NBAs

Let $\mathcal{A}$ be an NBA over $\Sigma$.

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

Let $\mathcal{A}$ be an NBA over $\Sigma$.

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

Let $A$ be an NBA over $\Sigma$.

- $L(A) = / \neq \emptyset$? in $P$ (i.e., linear-time algorithm)
- $L(A) = / \neq \Sigma^\omega$? is $P$Space-complete
Important properties of NBAs

Let \( A \) be an NBA over \( \Sigma \).

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

Let $\mathcal{A}$ be an NBA over $\Sigma$.

- $\mathcal{L}(\mathcal{A}) \neq \emptyset$ 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 $A$ be an NBA over $\Sigma$.

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

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

Let $\mathcal{A}$ be an NBA over $\Sigma$.

- $\mathcal{L}(\mathcal{A}) \neq \emptyset$ 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 $A_\varphi = (\Sigma, Q, Q_0, \delta, F)$, we define the “product automaton” $M \times A_\varphi = (\Sigma, Q', Q'_0, \delta', F')$ 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

$L(M \times A_\varphi) = L(M) \cap L(A_\varphi)$
Automata theoretic model checking

Given \( M = (S, s_0, R, L) \) and \( A_\varphi = (\Sigma, Q, Q_0, \delta, F) \), we define the “product automaton” \( M \times A_\varphi = (\Sigma, Q', Q'_0, \delta', F') \) 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 \( A_\varphi = (\Sigma, Q, Q_0, \delta, F) \), we define the "product automaton" \( M \times A_\varphi = (\Sigma, Q', Q'_0, \delta', F') \) 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**

\[ \mathcal{L}(M \times A_\varphi) = \mathcal{L}(M) \cap \mathcal{L}(A_\varphi) \]
Automata theoretic model checking

**Recall:** we need to test if $L(M \times A_{\varphi}) = \emptyset$. (How do we do it?)
Automata theoretic model checking

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

**Theorem**

$L(M \times A_{\varphi}) = \emptyset \iff$ *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 $A_{\varphi}$ are given.

... which is never the case in practice. :-(
Recall: we need to test if $\mathcal{L}(M \times A_\varphi) = \emptyset$. (How do we do it?)

Theorem

$\mathcal{L}(M \times A_\varphi) = \emptyset \iff \text{there is no reachable cycle containing a state from } F$.

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 $\mathcal{L}(M \times A_\varphi) = \emptyset$. (How do we do it?)

**Theorem**

$\mathcal{L}(M \times A_\varphi) = \emptyset \iff$ 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 $A_\varphi$ 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

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 $O(|V| + |E|)$. 

```
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
end 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
    end for

    // 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
```
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: \( \mathcal{M} \) and \( \mathcal{A}_\varphi \),

Initialize: \( \text{Stack1} := \emptyset \), \( \text{Stack2} := \emptyset \),
\( \text{Table1} := \emptyset \), \( \text{Table2} := \emptyset \);

procedure DFS2(s) {
    push(s, Stack2);
    hash(s, Table2);
    foreach \( t \in \text{Succ}^\odot(s) \) do {
        if \( t \not\in \text{Table2} \) then
            DFS2(t)
        else if \( t \) is on Stack1 {
            output(“bad cycle:”);
            output(Stack1, Stack2, t);
            exit;
        }
    }
    pop(Stack2);
}

procedure Main() {
    foreach \( s \in \text{Init}^\odot \) {
        if \( s \not\in \text{Table1} \) then DFS1(s);
    }
    output(“no bad cycle”);
    exit;
}

(Slide shamelessly stolen from Kousha Etessami.)
Recall: Input to the LTL model checking problem is a KS, $M$, and $\varphi$. 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.).
Recall: Input to the LTL model checking problem is a KS, $M$, and $\varphi$. 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: \( \mathcal{L}(\varphi) = \emptyset \iff \mathcal{L}(\varphi) \cap \Sigma^\omega = \emptyset \iff \Sigma^\omega \models \neg \varphi. \)

**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

\[ \varphi ::= p \in AP \mid \neg \varphi \mid \varphi \land \varphi \mid AX \varphi \mid EX \varphi \mid A(\varphi U \varphi) \mid E(\varphi U \varphi) \]

Note, there's no arbitrary nesting of path quantifiers (cf. CTL\(^*\)). For example, you can't say \[ XAX \varphi \] in CTL. But \[ EXEG \varphi \] is OK.
Computation Tree Logic (CTL)

**CTL syntax**

ϕ ::= p ∈ AP | ¬ϕ | ϕ ∧ ϕ | AXϕ | EXϕ | A(ϕUϕ) | E(ϕUϕ)

- Note, there’s no arbitrary nesting of path quantifiers (cf. CTL*).
- For example, you can’t say XAFϕ in CTL.
- But EFEGϕ is OK.
CTL—syntactic sugar and equalities

- $AX\varphi = \neg EX(\neg \varphi)$
- $EF\varphi = E(trueU\varphi)$
- $AG\varphi = \neg EF(\neg \varphi)$
- $AF\varphi = \neg EG(\neg \varphi)$
- $A(\varphi U \psi) = \neg E(\neg \psi U(\neg \varphi \land \neg \psi)) \land \neg EG\neg \psi$
- $A(\varphi R \psi) = \neg E(\neg \varphi U \neg \psi)$
- $E(\varphi R \psi) = \neg A(\neg \varphi U \neg \psi)$

Corollary

Any CTL formula can be expressed in terms of $\neg$, $\lor$, $EX$, $EU$, and $EG$ alone.
CTL—syntactic sugar and equalities

- $AX \varphi = \neg EX(\neg \varphi)$
- $EF \varphi = E(trueU \varphi)$
- $AG \varphi = \neg EF(\neg \varphi)$
- $AF \varphi = \neg EG(\neg \varphi)$
- $A(\varphi U \psi) = \neg E(\neg \psi U(\neg \varphi \land \neg \psi)) \land \neg EG \neg \psi$
- $A(\varphi R \psi) = \neg E(\neg \varphi U \neg \psi)$
- $E(\varphi R \psi) = \neg A(\neg \varphi U \neg \psi)$

Corollary

*Any CTL formula can be expressed in terms of $\neg$, $\lor$, $EX$, $EU$ 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
- $M, s \models \varphi \land \psi$ iff $M, s \models \varphi$ and $M, s \models \psi$
- $M, s \models \text{AX} \varphi$ iff for all $s \rightarrow s_1, M, s_1 \models \varphi$
- $M, s \models \text{EX} \varphi$ iff there is a $s \rightarrow s_1$, s.t. $M, s_1 \models \varphi$
- $M, s \models \text{A}(\varphi \text{U} \psi)$ iff for all $s_1 \rightarrow s_2 \rightarrow \ldots$, 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 $0 \leq j < k$
- $M, s \models \text{E}(\varphi \text{U} \psi)$ iff there is a $\ldots$
Some CTL specifications:

- **EF**(Start ∧ ¬Ready): It is possible to reach a state in which Start but not Ready holds.
- **AG**(Req → AFAck): Every req. is eventually answered.
- **AG**(AFDeviceEnabled): The device is enabled infinitely often on all paths.
- **AG**(EFRestart): From any state it is possible to reach a state in which Restart holds.
“Labelling algorithm”—what it does:

- **Input:** A CTL formula, $\varphi$, and a Kripke structure, $M = (S, s_0, R, L)$ over a set $AP$.

- **Output:** A set of formulae, $\text{label}(s_0)$, that are true in $s_0$ (i.e., $M, s_0 \models \varphi$ iff $\varphi \in \text{label}(s)$).
**CTL model checking—labelling algorithm**

“Labelling algorithm”—what it does:

- **Input**: A CTL formula, $\varphi$, 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 structural induction\(^1\) (that is, algorithm starts with innermost formulae and works its way “outwards”):

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

\(^1\)Only few cases due to earlier corollary!
CTL model checking—labelling algorithm

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

Runs in $O(|S| + |R|)$. 
**CTL model checking—labelling algorithm**

- \( \Phi = \text{EG} \varphi \) 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 \( \varphi \) does not hold)
    - \( R' = R|_{S' \times S'} \)
    - \( L' = L|_{S'} \)
**CTL model checking—labelling algorithm**

- $\Phi = \text{EG} \varphi$ 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 $\varphi$ does not hold)
    - $R' = R|_{S' \times S'}$
    - $L' = L|_{S'}$

**Lemma**

$M, s \models \text{EG} \varphi$ 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.

(⇒) As for 1.: Clearly, $s \in S'$.
Now we need to show 2. Let $w' = uw$ be a path in $M$ such that $\varphi$ 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 $\varphi$ is true along the path, we have for $u$ and $w$ that they're both contained in $S'$ by the construction of $M'$.

(⇐) 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 $\varphi$ holds, then it is a model for $\text{EG}\varphi$. Since the initial state of that path, $s \in S'$ is clearly also in $S$, the lemma follows.
ctl model checking—labelling algorithm

```
procedure CheckEG(f)
    S′ := { s | f ∈ label(s) };
    SCC := { C | C is a nontrivial SCC of S′ };
    T := ∪C∈SCC{ s | s ∈ C };
    for all s ∈ T do label(s) := label(s) ∪ { EG f };
    while T ≠ ∅ do
        choose s ∈ T;
        T := T \ {s};
        for all t such that t ∈ S′ and R(t, s) do
            if EG f ∉ label(t) then
                label(t) := label(t) ∪ { EG f };
                T := T ∪ {t};
            end if;
        end for all;
    end while;
end procedure
```

Runs in \( O(|S'| + |R'|) \).
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. Introduction
2. LTL model checking
3. CTL model checking
4. Binary decision diagrams
5. Symbolic model checking

Andreas Bauer
NICTA & ANU
COMP4600 Advanced algorithms: Algorithms for verification

\[ 1 \models AG(Start \rightarrow AF(Heat)) \]
CTL model checking—example

Same Kripke structure we used earlier:

```
1

Start, Error

2

"start oven"

3

"open door"

"reset"

4

Close, Heat

5

"start oven"

6

"start oven"

"start cooking"

7

Start, Close, Heat

Start, Close, Error

Start

Error

Close

Start

Close

Heat
```

Observe:

- $\text{AG}(\text{Start} \rightarrow \text{AFHeat})$ equiv. to $\neg \text{EF}(\text{Start} \land \text{EG}\neg\text{Heat})$
- We use $\text{EF}\varphi$ as shorthand for $\text{E}(\text{trueU}\varphi)$. 
CTL model checking—example

How the algorithm proceeds:

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

\[\text{not 6, because you can reach 7 from 6, where Heat is true}\]
CTL model checking—example

How the algorithm proceeds:

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

\[2\text{not 6, because you can reach 7 from 6, where Heat is true}\]
CTL model checking—example

How the algorithm proceeds:

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

---

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

\[\text{not 6, because you can reach 7 from 6, where Heat is true}\]
CTL model checking—example

How the algorithm proceeds:

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

\[\text{not 6, because you can reach 7 from 6, where } Heat \text{ is true}\]
How the algorithm proceeds:

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

---

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

\[^2\text{not 6, because you can reach 7 from 6, where Heat is true}\]
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 $x \rightarrow y_0, y_1$ be the if-then-else operator defined by

$$x \rightarrow y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1)$$

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

- $\neg x =$
Let \( x \rightarrow y_0, y_1 \) be the if-then-else operator defined by

\[
x \rightarrow y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1)
\]

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

- \( \neg x = (x \rightarrow 0, 1) \)
- \( x \iff y = \)
Binary decision diagrams

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

$$x \rightarrow y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1)$$

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

- $\neg x = (x \rightarrow 0, 1)$
- $x \iff y = x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 1)$
- etc.
Let $x \rightarrow y_0, y_1$ be the if-then-else operator defined by

$$x \rightarrow y_0, y_1 = (x \land y_0) \lor (\neg x \land y_1)$$

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

- $\neg x = (x \rightarrow 0, 1)$
- $x \iff y = x \rightarrow (y \rightarrow 1, 0), (y \rightarrow 0, 1)$
- 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 $t$,

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

- 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 $x \rightarrow t_1, t_0$. 
Binary decision diagrams—how to obtain INF?

**Theorem**

*Any Boolean expression is equivalent to an expression in INF.*

**Proof.**

See inductive INF construction.
Example: $t = (x_1 \leftrightarrow y_1) \land (x_2 \leftrightarrow y_2)$

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

$t = x_1 \rightarrow t_1, t_0$

$t_0 = y_1 \rightarrow 0, t_{00}$

$t_1 = y_1 \rightarrow t_{11}, 0$

$t_{00} = x_2 \rightarrow t_{001}, t_{000}$

$t_{11} = x_2 \rightarrow t_{111}, t_{110}$

$t_{000} = y_2 \rightarrow 0, 1$

$t_{001} = y_2 \rightarrow 1, 0$

$t_{110} = y_2 \rightarrow 0, 1$

$t_{111} = y_2 \rightarrow 1, 0$
Example: \( t = (x_1 \iff y_1) \land (x_2 \iff y_2) \)

Corresponding binary decision tree:

(Source: Henrik Reif Andersen’s lecture notes.)
Binary decision diagrams—how to obtain BDD?

Consider again:

\[
\begin{align*}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{111}, t_{110} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0 \\
t_{110} &= y_2 \rightarrow 0, 1 \\
t_{111} &= y_2 \rightarrow 1, 0 
\end{align*}
\]

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?

\[
\begin{align*}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{111}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0 \\
t_{111} &= y_2 \rightarrow 1, 0
\end{align*}
\]
Binary decision diagrams—how to obtain BDD?

\[
\begin{align*}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{11}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{11} &= x_2 \rightarrow t_{001/111}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0 \\
t_{111} &= y_2 \rightarrow 1, 0
\end{align*}
\]
Binary decision diagrams—how to obtain BDD?

\[
\begin{align*}
  t &= x_1 \rightarrow t_1, t_0 \\
  t_0 &= y_1 \rightarrow 0, t_{00} \\
  t_1 &= y_1 \rightarrow t_{00/11}, 0 \\
  t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
  t_{11} &= x_2 \rightarrow t_{001}, t_{000} \\
  t_{000} &= y_2 \rightarrow 0, 1 \\
  t_{001} &= y_2 \rightarrow 1, 0
\end{align*}
\]
Binary decision diagrams—how to obtain BDD?

\[ t = x_1 \rightarrow t_1, t_0 \]
\[ t_0 = y_1 \rightarrow 0, t_{00} \]
\[ t_1 = y_1 \rightarrow t_{00}, 0 \]
\[ t_{00} = x_2 \rightarrow t_{001}, t_{000} \]
\[ t_{000} = y_2 \rightarrow 0, 1 \]
\[ t_{001} = y_2 \rightarrow 1, 0 \]
Binary decision diagrams—how to obtain BDD?

\[

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
\]

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?

\[
\begin{align*}
t &= x_1 \rightarrow t_1, t_0 \\
t_0 &= y_1 \rightarrow 0, t_{00} \\
t_1 &= y_1 \rightarrow t_{00}, 0 \\
t_{00} &= x_2 \rightarrow t_{001}, t_{000} \\
t_{000} &= y_2 \rightarrow 0, 1 \\
t_{001} &= y_2 \rightarrow 1, 0
\end{align*}
\]

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 $\text{low}(u)$ and $\text{high}(u)$. (In pictures, these are shown as dotted and solid lines, respectively). A variable $\text{var}(u)$ is associated with each variable node.
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.,
  \[ \text{var}(u) = \text{var}(v), \text{low}(u) = \text{low}(v), \text{high}(u) = \text{high}(v) \Rightarrow u = v \]

- (no redundancy) no variable node $u$ has identical low- and high-successor, i.e., $\text{low}(u) \neq \text{high}(u)$. 
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 $f : \mathbb{B}^n \rightarrow \mathbb{B}$. Nodes $u$ of ROBDD for $f$ inductively define Boolean expressions $t^u$:

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

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 \( f : \mathbb{B}^n \rightarrow \mathbb{B} \). Nodes \( u \) of ROBDD for \( f \) inductively define Boolean expressions \( t^u \):

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

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 \( f : \mathbb{B}^n \rightarrow \mathbb{B} \) 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.).
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 $f, g : \mathbb{B}^n \to \mathbb{B}$

- How do you check validity of $f$ if given as ROBDD? (compare to non-terminal node; $O(1)$ vs coNP for formulae)
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)
- How do you check equivalence of $f$ and $g$ if given as ROBDDs?
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)
- How do you check equivalence of $f$ and $g$ if given as ROBDDs? (compare nodes; $O(n)$ vs coNP for formulae)
Consider ROBDD for \((x_1 \Leftrightarrow y_1) \land (x_2 \Leftrightarrow y_2)\)

... but different var. ordering of \(x_1 < x_2 < y_1 < y_2\):
Consider ROBDD for \((x_1 \Leftrightarrow y_1) \land (x_2 \Leftrightarrow y_2)\)

...but different var. ordering of \(x_1 < x_2 < y_1 < y_2\):
ROBDDs—construction

We saw how to construct OBDD, but how to construct ROBBD?
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 $H(i, l, h) = u$)
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

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

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

($\text{Mk}[T, H]$ means that $\text{Mk}$ 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:

\[
\text{Mk}[T, H](i, l, h) =
\begin{cases}
    \text{return } l & \text{if } l = h \\
    \text{else if } \text{member}(H, i, l, h) \text{ then return } \text{lookup}(H, i, l, h) \\
    \text{else } u \leftarrow \text{add}(T, i, l, h) \\
    \text{insert}(H, i, l, h, u) \\
    \text{return } u
\end{cases}
\]

($MK[T, H]$ 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:

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

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

What is the running time of $\text{MK}$?

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 $t$.

$$\text{BUILD}[T, H](t)$$
1: function BUILD'(t, i) =
2: if $i > n$ then
3: if $t$ is false then return 0 else return 1
4: else $v_0 \leftarrow$ BUILD'(t[0/$x_i$], $i + 1$)
5: $v_1 \leftarrow$ BUILD'(t[1/$x_i$], $i + 1$)
6: return MK($i, v_0, v_1$)
7: end BUILD'
8:
9: return BUILD'(t, 1)
ROBDDs—construction

**Input:** $t$ be Boolean expression of $n$ var (with fixed var. ordering).

**Output:** ROBBD of $t$.

\[
\text{\textbf{BUILD}}[T, H](t)
\]

1: function \textbf{BUILD}'$(t, i)$ =
2: \hspace{1em} if $i > n$ then Evaluation
3: \hspace{2em} if $t$ is false then return 0 else return 1
4: \hspace{2em} else $v_0 \leftarrow \text{\textbf{BUILD}}'(t[0/x_i], i + 1)$
5: \hspace{2em} $v_1 \leftarrow \text{\textbf{BUILD}}'(t[1/x_i], i + 1)$
6: \hspace{1em} return \text{MK}$(i, v_0, v_1)$ Shannon expansion
7: end \textbf{BUILD}'
8:  
9: return \text{\textbf{BUILD}}'(t, 1)
ROBDDs—construction

**Input:** $t$ be Boolean expression of $n$ var (with fixed var. ordering).

**Output:** ROBBD of $t$.

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

1: \hspace{1em} \textbf{function} \ BUILD'(t, i) =  
2: \hspace{1em} \textbf{if} \ i > n \ \textbf{then} \ \textbf{Evaluation}  
3: \hspace{2em} \textbf{if} \ t \ \textbf{is false} \ \textbf{then} \ \textbf{return} \ 0 \ \textbf{else} \ \textbf{return} \ 1  
4: \hspace{2em} \textbf{else} \ \textbf{return} \ \textbf{BUILD}'(t[0/x_i], i + 1)  
5: \hspace{2em} v_1 \ \leftarrow \ \textbf{BUILD}'(t[1/x_i], i + 1)  
6: \hspace{2em} \textbf{return} \ \text{MK}(i, v_0, v_1) \ \} \ \text{Shannon expansion}  
7: \hspace{1em} \textbf{end BUILD}'  
8: \hspace{1em} \textbf{return} \ \text{BUILD}'(t, 1)  

What is the running time of \text{BUILD}?
**ROBDDs—construction**

**Input:** $t$ be Boolean expression of $n$ var (with fixed var. ordering).

**Output:** ROBBD of $t$.

```
BUILD[T, H](t)
1: function BUILD'(t, i) =
2:   if $i > n$ then Evaluation
3:     if $t$ is false then return 0 else return 1
4:   else $v_0 \leftarrow$ BUILD'(t[0/$x_i$], $i + 1$)
5:     $v_1 \leftarrow$ BUILD'(t[1/$x_i$], $i + 1$)
6:     return MK($i, v_0, v_1$) Shannon expansion
7: end BUILD'
8:
9: 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 \((x_1 \Leftrightarrow x_2) \lor x_3:\)

Can we do better?

---

Andreas Bauer

NICTA & ANU

COMP4600 Advanced algorithms: Algorithms for verification
Intuitive explanation for bad running time:

*BUILD* callgraph on \((x_1 \leftrightarrow x_2) \lor x_3:\)

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

\textbf{APPLY}[T,H](op,u_1,u_2)

1: \textit{init}(G)

2:

3: \textbf{function APP}(u_1,u_2) =

4: \quad \textbf{if } G(u_1,u_2) \neq \text{ empty \ then return } G(u_1,u_2)

5: \quad \textbf{else if } u_1 \in \{0,1\} \ \text{ and } u_2 \in \{0,1\} \ \text{ then } u \leftarrow \text{op}(u_1,u_2)

6: \quad \textbf{else if } \text{var}(u_1) = \text{var}(u_2) \ \text{ then}

7: \quad \quad u \leftarrow \text{MK}(\text{var}(u_1), \text{APP}(\text{low}(u_1),\text{low}(u_2)), \text{APP}(\text{high}(u_1),\text{high}(u_2)))

8: \quad \textbf{else if } \text{var}(u_1) < \text{var}(u_2) \ \text{ then}

9: \quad \quad u \leftarrow \text{MK}(\text{var}(u_1), \text{APP}(\text{low}(u_1),u_2), \text{APP}(\text{high}(u_1),u_2))

10: \quad \quad \textbf{else } (*) \text{var}(u_1) > \text{var}(u_2) (*)

11: \quad \quad u \leftarrow \text{MK}(\text{var}(u_2), \text{APP}(u_1,\text{low}(u_2)), \text{APP}(u_1,\text{high}(u_2)))

12: \quad G(u_1,u_2) \leftarrow u

13: \quad \textbf{return } u

14: \textbf{end APP}

15:

16: \textbf{return} APP(u_1,u_2)

\textbf{Uses Shannon expansion:}

- \( t = x \rightarrow t[1/x], t[0/x] \)
- \((x \rightarrow t_1, t_2) \text{ op } (x \rightarrow t'_1, t'_2) = x \rightarrow t_1 \text{ op } t'_1, t_2 \text{ op } t'_2 \)
- \((x \rightarrow t_1, t_2) \text{ op } t_3 = x \rightarrow t_1 \text{ op } t_3, t_2 \text{ op } t_3 \)
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 arbitrarily, 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 arbitrarily, but add $2^n$ more assignments in total, respectively.

$$\text{SatCount}[T](u)$$

1: function $count(u)$
2: if $u = 0$ then $res \leftarrow 0$
3: else if $u = 1$ then $res \leftarrow 1$
4: else $res \leftarrow 2^{var(low(u))-var(u)-1} \times count(low(u))$
5: \hspace{1cm} + $2^{var(high(u))-var(u)-1} \times count(high(u))$
6: return $res$
7: end $count$
ROBDDs—AnySat & AllSat

\textbf{AnySat}(u)

1: \textbf{if } u = 0 \textbf{ then Error}
2: \textbf{else if } u = 1 \textbf{ then return } []
3: \textbf{else if } \text{low}(u) = 0 \textbf{ then return } [x_{\text{var}(u)} \mapsto 1, \text{ AnySat}(\text{high}(u))] \newline
4: \textbf{else return } [x_{\text{var}(u)} \mapsto 0, \text{ AnySat}(\text{low}(u))]

\textbf{AllSat}(u)

1: \textbf{if } u = 0 \textbf{ then return } \langle \rangle \\
2: \textbf{else if } u = 1 \textbf{ then return } \langle [\ ] \rangle \\
3: \textbf{else return}
4: \langle \text{add } [x_{\text{var}(u)} \mapsto 0] \text{ in front of all truth-assignments in AllSat(low(u))}, \rangle \\
5: \text{add } [x_{\text{var}(u)} \mapsto 1] \text{ in front of all truth-assignments in AllSat(high(u))} \rangle
### ROBDDs—algorithm running times

<table>
<thead>
<tr>
<th>Function</th>
<th>Complexity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mk($i, u_0, u_1$)</td>
<td>$O(1)$</td>
</tr>
<tr>
<td>Build($t$)</td>
<td>$O(2^n)$</td>
</tr>
<tr>
<td>Apply($op, u_1, u_2$)</td>
<td>$O(</td>
</tr>
<tr>
<td>Restrict($u, j, b$)</td>
<td>$O(</td>
</tr>
<tr>
<td>SatCount($u$)</td>
<td>$O(</td>
</tr>
<tr>
<td>AnySat($u$)</td>
<td>$O(</td>
</tr>
<tr>
<td>AllSat($u$)</td>
<td>$O(</td>
</tr>
<tr>
<td>Simplify($d, u$)</td>
<td>$O(</td>
</tr>
</tbody>
</table>
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 $\mathcal{M} \rightarrow M$ (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 $\mathcal{M} \rightarrow M$ (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 $\mathcal{M} \rightarrow M$ (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:

![Transition diagram](image)
Symbolic model checking—basic idea

- For example:

- Transition $s_1 \rightarrow s_2$ is $a \land b \land a' \land \neg b'$
Symbolic model checking—basic idea

- For example:

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

- Whole TS:
  $$(a \land b \land a' \land \neg b') \lor (a \land \neg b \land a' \land \neg b') \lor (a \land \neg b \land a' \land b')$$
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 $c_i = 1 \land t_i = 0$ then $t_i, c_i, h_i := 1, 0, 1$
  - if $h_i = 1$ then $c_{(i \mod N)+1}, h_i := 1, 0$

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

$$\text{unchanged}_S = \bigwedge_{x \in S} x = x'$$

(Or, $\text{assigned}_{S'} = \text{unchanged}_{\overline{x \setminus S'}}$, 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:

$$P_i = (c_i \land \neg t_i \land t'_i \land \neg c'_i \land h'_i \land \text{assigned}\{c_i, t_i, h_i\})$$

$$\lor (h_i \land c'_i \mod N + 1 \land \neg h'_i \land \text{assigned}\{(c_i \mod N + 1, h_i)\})$$

Termination of task:

$$E_i = t_i \land \neg t'_i \land \text{assigned}\{t_i\}$$

All possible transitions:

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

Initial state (only $c_1$ has token):

$$I = \neg \vec{t} \land \neg \vec{h} \land c_1 \land \neg c_2 \land \ldots \land \neg c_N$$
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?
- \( \ldots \)

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 satisfy $I$ or within finite number of transitions can be reached from $I$. 

Suggests iterative process:

Let $R_0 = 0$ and compute $R_{k+1}$ as disjunction of $I$ and the set of states reachable from $R_k$. 

Andreas Bauer NICTA & ANU

COMP4600 Advanced algorithms: Algorithms for verification
Some observations:

- $R$ needs to satisfy $I$ 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 satisfy $I$ or within finite number of transitions can be reached from $I$.
- Suggests iterative process: $R^0$, $R^1$, $R^2$, \ldots
- 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 satisfy $I$ 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, x, x')
1:   R ← 0
2:   repeat
3:     R' ← R
4:     R ← I ∨ (∃x. T ∧ R)[x/x']
5:   until R' = R
6:   return R
```