Model checking

Software Analysis
Topic 6

Carlo A. Furia

USI – Università della Svizzera Italiana

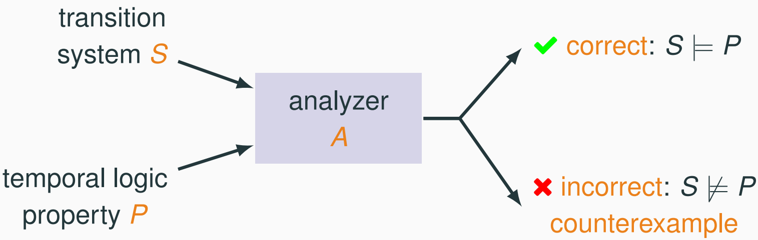
Today's menu

Automata-based model checking

Software model checking

Real-time model checking

Model checking: the very idea



Model checking is the algorithmic verification of finite-state systems:

- analyzes (finite-state) models formalized as transition systems
- verifies ordering and reachability properties expressed in temporal logic
- is completely automatic (“push button”)
- is sound and complete, as it targets fully decidable models
- when model checking fails, it returns a counterexample

Model checking: this lecture

Model checking is a popular verification technique that has been applied in different ways. Due to its popularity, the name "model checking" has been sometimes used to describe techniques that deviate significantly from the original technique.

Model checking: this lecture

Model checking is a popular verification technique that has been applied in different ways. Due to its popularity, the name "model checking" has been sometimes used to describe techniques that deviate significantly from the original technique.

In this lecture we have a look at three variants of model checking:

model checking in the automata-based framework – the most fundamental and conceptually elegant presentation of the ideas of model checking

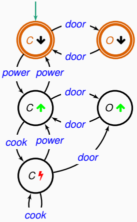
software model checking combines model checking and predicate abstraction techniques to analyze real code using finite-state abstractions

real-time model checking analyzes quantitative time models

Automata-based model checking

Verification of finite-state systems

Model checking denotes a family of techniques for the algorithmic verification of finite-state systems with temporal-logic specifications



model checker A

$$\square\left(\begin{array}{c} cook \\ \Longrightarrow \\ cook \ \mathrm{U} \ power \end{array}\right)$$

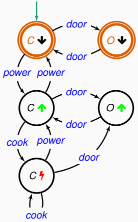
✔ correct

✖ incorrect: counterexample

power cook door door power

Verification of finite-state systems

Model checking denotes a family of techniques for the algorithmic verification of finite-state systems with temporal-logic specifications



(finite-state) transition system, often a kind of automata

some flavor of temporal or modal logic

$$\square\left(\mathit{cook} \implies \mathit{cook}\ \mathrm{U}\ \mathit{power} \right)$$

model checker A

✔ correct

✘ incorrect: counterexample

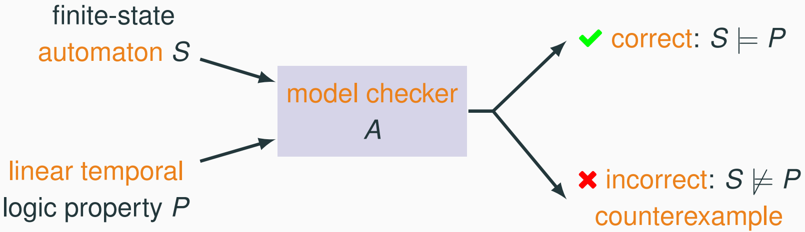
power cook door door power

Linear-time model checking

Linear-time model checking problem: given

- S: a finite-state automaton (FSA)
- P: a linear temporal logic (LTL) property

determine if every run of S satisfies P, or
provide a counterexample: a run of S that violates P

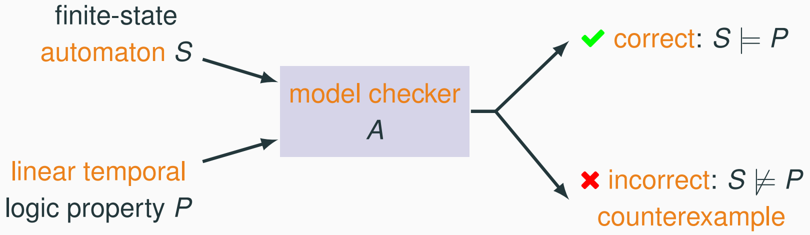


Linear-time model checking

Linear-time model checking problem: given

- S: a finite-state automaton (FSA)
- P: a linear temporal logic (LTL) property

determine if every run of S satisfies P, or
provide a counterexample: a run of S that violates P



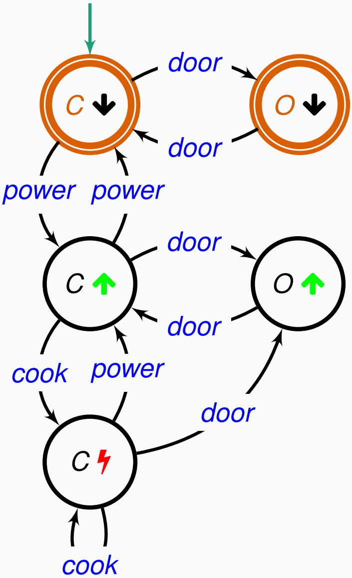
We first describe syntax and semantics of FSAs and LTL, and then describe an algorithm for linear-time model checking.

Automata-based model checking

Finite-state automata

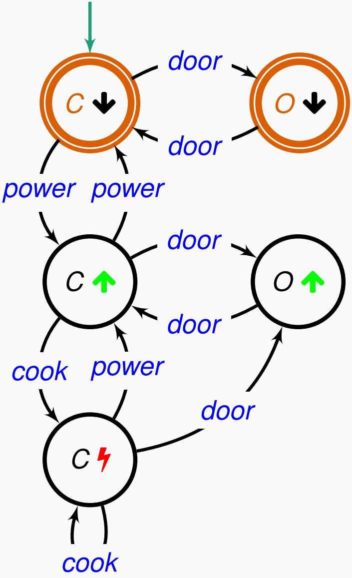
Finite-state automata: example

We model the behavior of a microwave oven using an FSA.



Finite-state automata: example

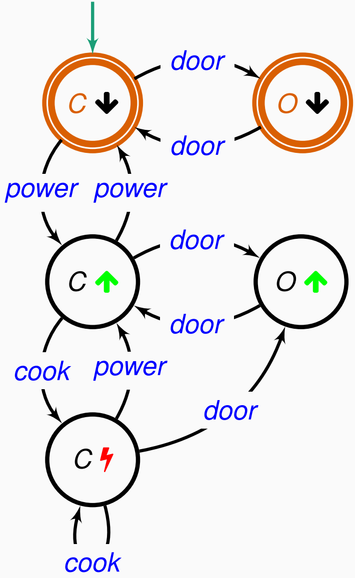
We model the behavior of a microwave oven using an FSA.



The oven's door can be open O or closed C.

Finite-state automata: example

We model the behavior of a microwave oven using an FSA.

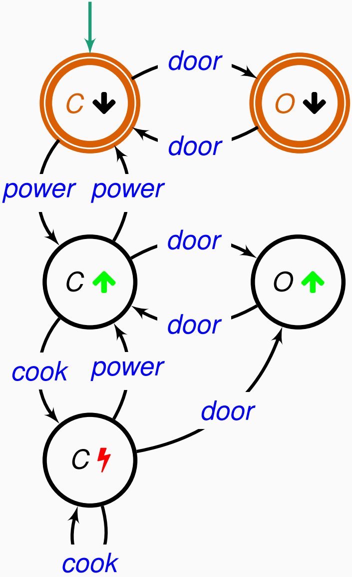


The oven's door can be open O or closed C.

The oven may be powered on ⬆ or off ⬇.

Finite-state automata: example

We model the behavior of a microwave oven using an FSA.



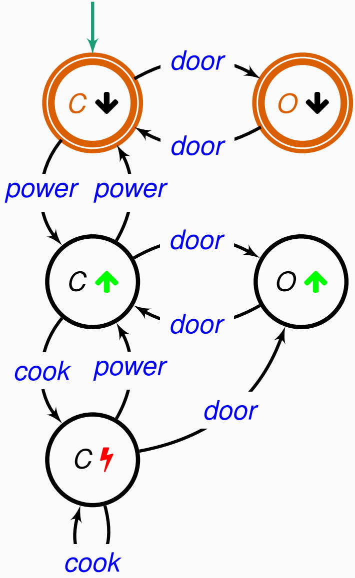
The oven's door can be open O or closed C.

The oven may be powered on ↑ or off ↓.

When the oven is on and closed, pressing the cook button *cook* starts cooking ϟ.

Finite-state automata: example

We model the behavior of a microwave oven using an FSA.



The oven's door can be open *O* or closed *C*.

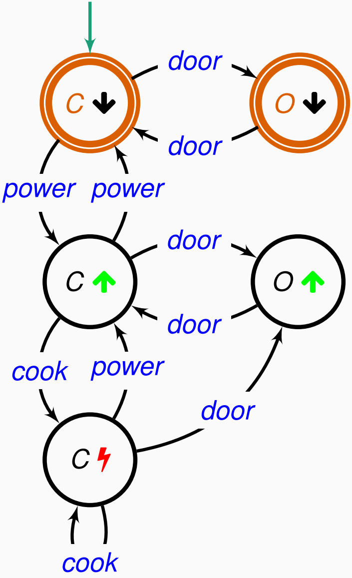
The oven may be powered on ↑ or off ↓.

When the oven is on and closed, pressing the cook button *cook* starts cooking ⚡.

The oven is initially closed and off, and it must be eventually powered off.

Finite-state automata: example

We model the behavior of a microwave oven using an FSA.



The oven's door can be open O or closed C.

The oven may be powered on ↑ or off ↓.

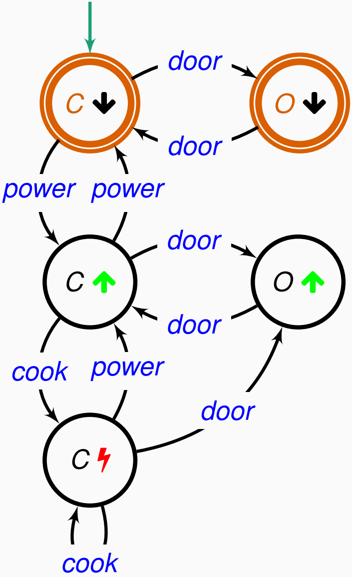
When the oven is on and closed, pressing the cook button *cook* starts cooking ϟ.

The oven is initially closed and off, and it must be eventually powered off.

The *cook* button may only be pressed when the oven is closed and on or cooking.

Finite-state automata: example

We model the behavior of a microwave oven using an FSA.



The oven's door can be open *O* or closed *C*.

The oven may be powered on ↑ or off ↓.

When the oven is on and closed, pressing the cook button *cook* starts cooking ϟ.

The oven is initially closed and off, and it must be eventually powered off.

The *cook* button may only be pressed when the oven is closed and on or cooking.

Pushing the *power* or *door* button while the oven is cooking immediately stops cooking.

Finite-state automata: syntax

A nondeterministic finite-state automaton (FSA) A
is a tuple $\langle \Sigma, S, I, F, \rho \rangle$:

- Σ: finite nonempty input alphabet
- S: finite nonempty set of states
- $I \subseteq S$: set of initial states
- $F \subseteq S$: set of final (accepting) states
- $\rho\colon S \times \Sigma \to \wp(S)$: transition function

An FSA A is deterministic if, for all s, σ, $|\rho(s, \sigma)| \leq 1$; that is, there is at most one outgoing transition from any state s for each input symbol σ.

Finite-state automata: syntax

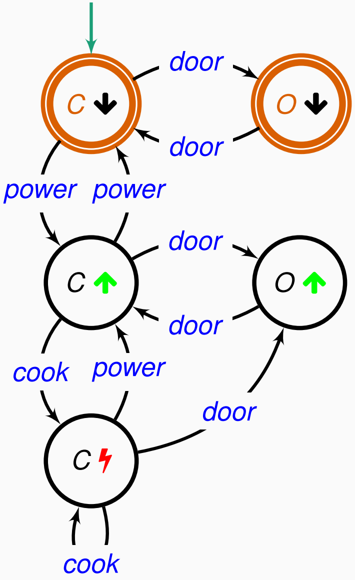
A nondeterministic finite-state automaton (FSA) A
is a tuple $\langle \Sigma, S, I, F, \rho \rangle$:

- Σ: finite nonempty input alphabet
- S: finite nonempty set of states
- $I \subseteq S$: set of initial states
- $F \subseteq S$: set of final (accepting) states
- $\rho \colon S \times \Sigma \to \wp(S)$: transition function

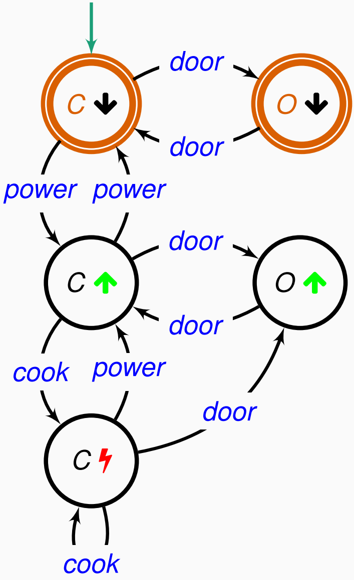
An FSA A is deterministic if, for all s, σ, $|\rho(s, \sigma)| \leq 1$; that is, there is at most one outgoing transition from any state s for each input symbol σ.

We commonly represent FSAs with a graph whose nodes are states, transitions are edges, and input events decorate transitions, and initial and final states are marked.

FSA syntax: example

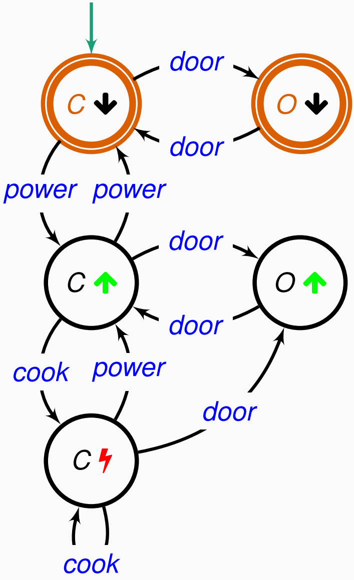


FSA syntax: example



alphabet $\Sigma = \{power, door, cook\}$

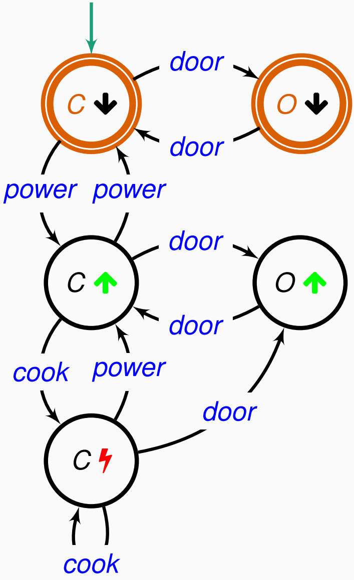
FSA syntax: example



alphabet $\Sigma = \{power, door, cook\}$

states $S = \{C{\downarrow}, C{\uparrow}, O{\downarrow}, O{\uparrow}, C⚡\}$

FSA syntax: example

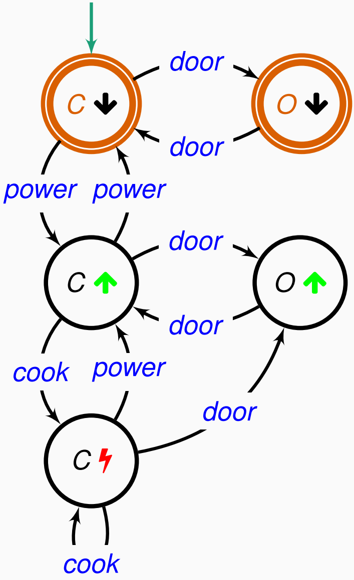


alphabet $\Sigma = \{\mathit{power}, \mathit{door}, \mathit{cook}\}$

states $S = \{C\downarrow, C\uparrow, O\downarrow, O\uparrow, C\lightning\}$

initial states $I = \{C\downarrow\}$

FSA syntax: example



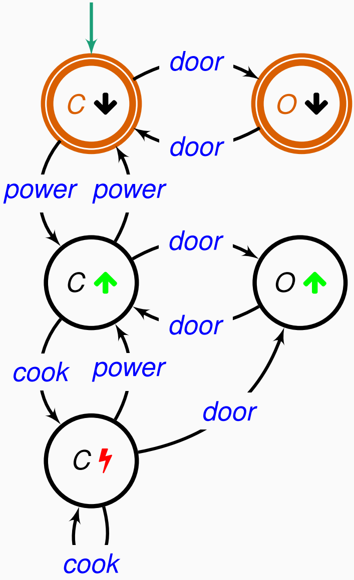
alphabet $\Sigma = \{power, door, cook\}$

states $S = \{C{\downarrow}, C{\uparrow}, O{\downarrow}, O{\uparrow}, C\lightning\}$

initial states $I = \{C{\downarrow}\}$

final states $F = \{C{\downarrow}, O{\downarrow}\}$

FSA syntax: example



alphabet $\Sigma = \{power, door, cook\}$

states $S = \{C{\downarrow}, C{\uparrow}, O{\downarrow}, O{\uparrow}, C\lightning\}$

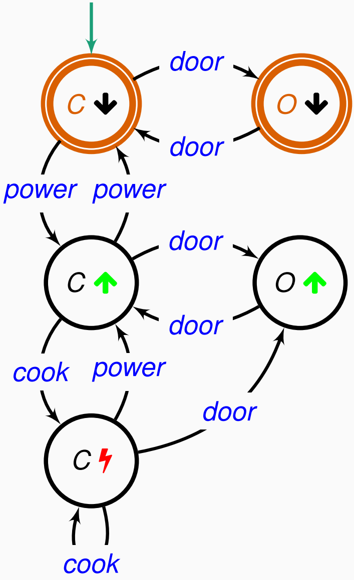
initial states $I = \{C{\downarrow}\}$

final states $F = \{C{\downarrow}, O{\downarrow}\}$

transitions ρ:

- $\rho(C{\downarrow}, power) = \{C{\uparrow}\}$
- $\rho(O{\downarrow}, door) = \{C{\downarrow}\}$
- $\rho(O{\downarrow}, cook) = \{\}$
- $\ldots$

FSA syntax: example



alphabet $\Sigma = \{\textit{power}, \textit{door}, \textit{cook}\}$

states $S = \{C{\downarrow}, C{\uparrow}, O{\downarrow}, O{\uparrow}, C⚡\}$

initial states $I = \{C{\downarrow}\}$

final states $F = \{C{\downarrow}, O{\downarrow}\}$

transitions ρ:

- $\rho(C{\downarrow}, \textit{power}) = \{C{\uparrow}\}$
- $\rho(O{\downarrow}, \textit{door}) = \{C{\downarrow}\}$
- $\rho(O{\downarrow}, \textit{cook}) = \{\}$
- $\ldots$

The automaton is deterministic

Finite-state automata: runs

Let $A = \langle \Sigma, S, I, \rho, F \rangle$ be an FSA.

An input word is an input sequence of any (finite) length:

$$w = w[1]\, w[2]\, \ldots\, w[n] \in \Sigma^*$$

The empty word ϵ has zero length ($n = |\epsilon| = 0$).

Finite-state automata: runs

Let $A = \langle \Sigma, S, I, \rho, F \rangle$ be an FSA.

An input word is an input sequence of any (finite) length:

$$w = w[1]\, w[2] \,\ldots\, w[n] \in \Sigma^*$$

The empty word ϵ has zero length ($n = |\epsilon| = 0$).

A run of A over w is a sequence of states

$$r = r[0]\, r[1] \,\ldots\, r[n] \in S^* \qquad \text{that:}$$

- starts from an initial state: $r[0] \in I$
- follows A's transitions: for all $1 \leq k \leq n$, $r[k] \in \rho(r[k-1], w[k])$

Finite-state automata: runs

Let $A = \langle \Sigma, S, I, \rho, F \rangle$ be an FSA.

An input word is an input sequence of any (finite) length:

$$w = w[1]\, w[2] \ldots w[n] \in \Sigma^*$$

The empty word ϵ has zero length ($n = |\epsilon| = 0$).

A run of A over w is a sequence of states

$$r = r[0]\, r[1] \ldots r[n] \in S^* \qquad \text{that:}$$

- starts from an initial state: $r[0] \in I$
- follows A's transitions: for all $1 \leq k \leq n$, $r[k] \in \rho(r[k-1], w[k])$

A run r is accepting if it ends in a final state: $r[n] \in F$.
In this case we say that A accepts w.

Finite-state automata: runs

Let $A = \langle \Sigma, S, I, \rho, F \rangle$ be an FSA.

An input word is an input sequence of any (finite) length:

$$w = w[1]\, w[2]\, \ldots\, w[n] \in \Sigma^*$$

The empty word ϵ has zero length ($n = |\epsilon| = 0$).

A run of A over w is a sequence of states

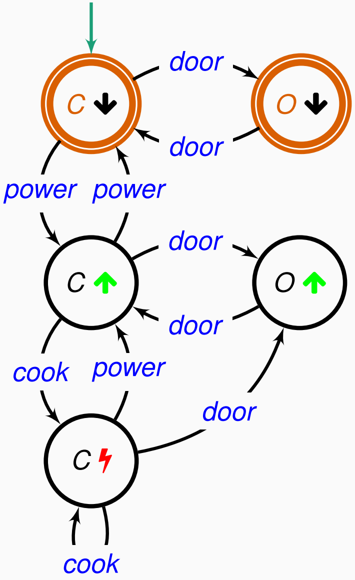
$$r = r[0]\, r[1]\, \ldots\, r[n] \in S^* \qquad \text{that:}$$

- starts from an initial state: $r[0] \in I$
- follows A's transitions: for all $1 \leq k \leq n$, $r[k] \in \rho(r[k-1], w[k])$

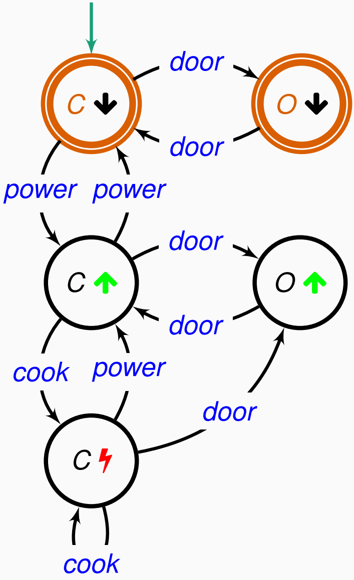
A run r is accepting if it ends in a final state: $r[n] \in F$.
In this case we say that A accepts w.

In practice, an accepting run is any path on A's directed graph that starts in an initial state and ends in a final state.

FSA runs: example

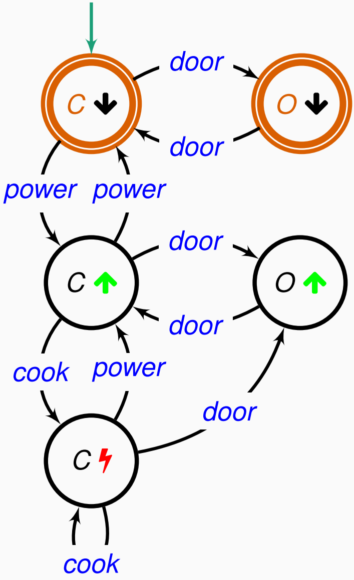


FSA runs: example



Run $r = C{\downarrow}\ C{\uparrow}\ C\text{ϟ}\ O{\uparrow}$

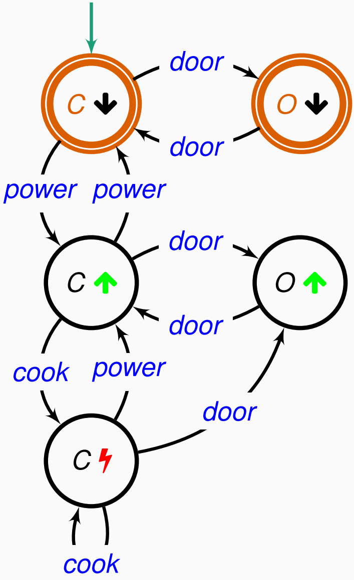
FSA runs: example



Run $r = C$ ⬇ C ⬆ C ⚡ O ⬆

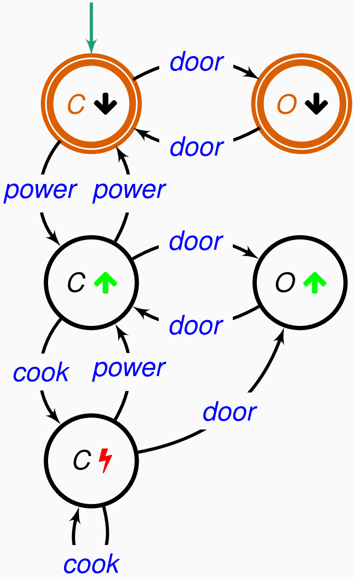
over $w = \mathit{power}\ \mathit{cook}\ \mathit{door}$

FSA runs: example



Run $r = C{\downarrow}\; C{\uparrow}\; C{\lightning}\; O{\uparrow}$
over $w = \textit{power cook door}$
is not accepting.

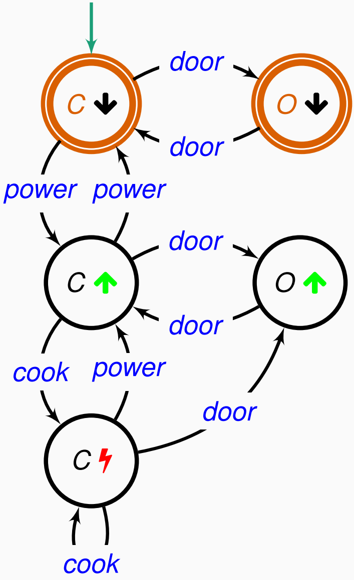
FSA runs: example



Run $r = C{\downarrow}\ C{\uparrow}\ C{\lightning}\ O{\uparrow}$
over $w = \mathit{power}\ \mathit{cook}\ \mathit{door}$
is not accepting.

Run $r = C{\downarrow}\ C{\uparrow}\ C{\lightning}\ C{\uparrow}\ C{\downarrow}$

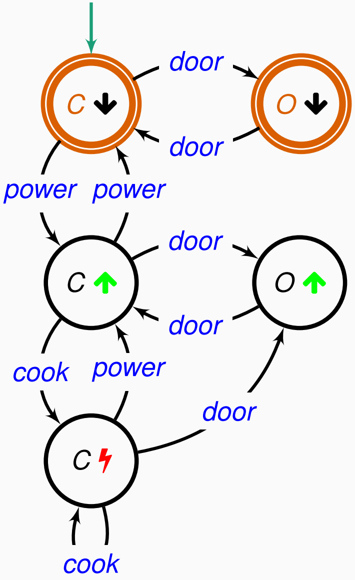
FSA runs: example



Run $r = C$ ↓ C ↑ C ϟ O ↑
over $w =$ *power cook door*
is not accepting.

Run $r = C$ ↓ C ↑ C ϟ C ↑ C ↓
over $w =$ *power cook power power*

FSA runs: example



Run $r = C{\downarrow}\ C{\uparrow}\ C{\lightning}\ O{\uparrow}$
over $w =$ *power cook door*
is not accepting.

Run $r = C{\downarrow}\ C{\uparrow}\ C{\lightning}\ C{\uparrow}\ C{\downarrow}$
over $w =$ *power cook power power*
is accepting.

Finite-state automata: semantics

The language of a finite-state automaton $A = \langle \Sigma, S, I, F, \rho \rangle$ is the set of input words that A accepts:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \text{there is an accepting run of } A \text{ over } w\}$$

Finite-state automata: semantics

The language of a finite-state automaton $A = \langle \Sigma, S, I, F, \rho \rangle$ is the set of input words that A accepts:

$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \text{there is an accepting run of } A \text{ over } w\}$$

The emptiness problem is the fundamental decision problem of automata (and operational formalisms in general):

The emptiness problem: given an automaton A, determine if it accepts any words – that is if A's language is empty.

Finite-state automata: semantics

The language of a finite-state automaton $A = \langle \Sigma, S, I, F, \rho \rangle$ is the set of input words that A accepts:

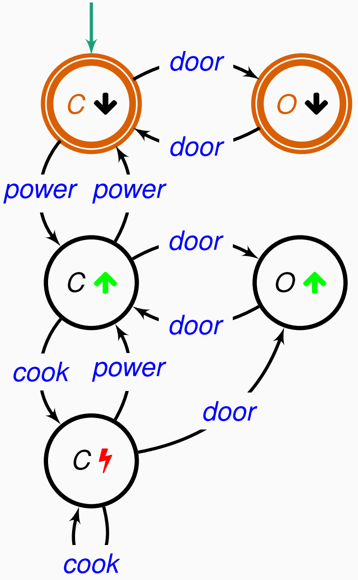
$$\mathcal{L}(A) = \{w \in \Sigma^* \mid \text{there is an accepting run of } A \text{ over } w\}$$

The emptiness problem is the fundamental decision problem of automata (and operational formalisms in general):

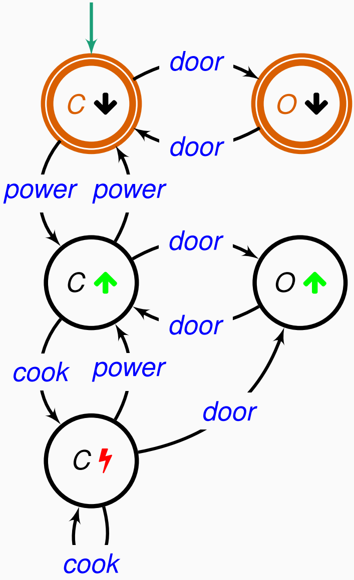
The emptiness problem: given an automaton A, determine if it accepts any words – that is if A's language is empty.

Note that even though FSAs have a finite number of states, their languages may consist of an infinite number of words.

FSA semantics: example

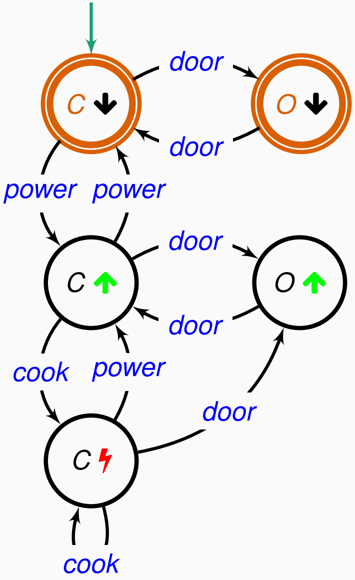


FSA semantics: example



The language of the microwave automaton is not empty.

FSA semantics: example



The language of the microwave automaton is not empty.

Words in the automaton's language include:

- *power power*
- *power cook power door door power*
- ϵ
- ...

Automata-based model checking

Linear temporal logic

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

$$door \implies \mathsf{X}\, door$$

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

$$door \Longrightarrow \mathsf{X}\, door$$

The door will be closed eventually after it is opened:

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

$$\mathit{door} \Longrightarrow \mathsf{X}\, \mathit{door}$$

The door will be closed eventually after it is opened:

$$\mathit{door} \Longrightarrow \Diamond\, \mathit{door}$$

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

$$door \Longrightarrow \mathsf{X}\, door$$

The door will be closed <u>eventually</u> after it is opened:

$$door \Longrightarrow \Diamond\, door$$

The oven cooks until it is turned off:

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

$$door \implies \mathsf{X}\, door$$

The door will be closed eventually after it is opened:

$$door \implies \Diamond\, door$$

The oven cooks until it is turned off:

$$cook\ \mathsf{U}\ power$$

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

$$door \implies \mathsf{X}\, door$$

The door will be closed eventually after it is opened:

$$door \implies \Diamond\, door$$

The oven cooks until it is turned off:

$$cook \;\mathsf{U}\; power$$

The power button is never pressed:

Linear temporal logic: example

We model properties of the microwave oven using LTL formulas:

The door is closed right after it is opened:

$$door \Longrightarrow \mathsf{X}\, door$$

The door will be closed <u>eventually</u> after it is opened:

$$door \Longrightarrow \Diamond\, door$$

The oven cooks until it is turned off:

$$cook\ \mathsf{U}\ power$$

The power button is never pressed:

$$\Box(\neg power)$$

Linear temporal logic: syntax

Formulas of propositional linear temporal logic (LTL) are defined as:

$$
\begin{aligned}
F ::= \; & p \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 && \text{(propositional connectives)} \\
\mid \; & \mathsf{X}\, F \mid \Box\, F \mid \Diamond\, F \mid F_1 \,\mathsf{U}\, F_2 && \text{(temporal connectives)}
\end{aligned}
$$

where $p \in \Pi$ is any proposition from a set Π.

Linear temporal logic: syntax

box/always/
globally

next/in the
next step

Formulas of propositional linear temporal logic (LTL) are defined as:

$$F ::= p \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 \quad \text{(propositional connectives)}$$
$$\mid \; \mathrm{X}\,F \mid \Box\,F \mid \Diamond\,F \mid F_1 \,\mathrm{U}\, F_2 \quad \text{(temporal connectives)}$$

where $p \in \Pi$ is any proposition from a set Π.

diamond/eventually/
sometimes

until

Linear temporal logic: syntax

next/in the next step

box/always/ globally

Formulas of propositional linear temporal logic (LTL) are defined as:

$$F ::= p \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 \quad \text{(propositional connectives)}$$

$$\mid \; \mathrm{X}\, F \mid \Box F \mid \Diamond F \mid F_1 \,\mathrm{U}\, F_2 \quad \text{(temporal connectives)}$$

where $p \in \Pi$ is any proposition from a set Π.

diamond/eventually/ sometimes

until

Temporal connectives, also called temporal operators or modal operators, describe when their arguments are true.

LTL syntax: examples

The examples we have seen before are LTL formulas over propositions in $\Pi = \{\mathit{door}, \mathit{power}, \mathit{cook}\}$.

LTL syntax: examples

The examples we have seen before are LTL formulas over propositions in $\Pi = \{\mathit{door}, \mathit{power}, \mathit{cook}\}$.

The door is closed right after it is opened:

$$\mathit{door} \Longrightarrow \mathsf{X}\, \mathit{door}$$

LTL syntax: examples

The examples we have seen before are LTL formulas over propositions in $\Pi = \{door, power, cook\}$.

The door is closed right after it is opened:

$$door \implies \mathsf{X}\, door$$

The door will be closed eventually after it is opened:

$$door \implies \Diamond\, door$$

LTL syntax: examples

The examples we have seen before are LTL formulas over propositions in $\Pi = \{door, power, cook\}$.

The door is closed right after it is opened:

$$door \Longrightarrow \mathsf{X}\, door$$

The door will be closed eventually after it is opened:

$$door \Longrightarrow \Diamond\, door$$

The oven cooks until it it turned off:

$$cook \ \mathsf{U}\ power$$

LTL syntax: examples

The examples we have seen before are LTL formulas over propositions in $\Pi = \{door, power, cook\}$.

The door is closed right after it is opened:

$$door \Longrightarrow \mathsf{X}\, door$$

The door will be closed eventually after it is opened:

$$door \Longrightarrow \Diamond\, door$$

The oven cooks until it it turned off:

$$cook \;\mathsf{U}\; power$$

The power button is never pressed:

$$\Box(\neg power)$$

Linear temporal logic: satisfaction relation

A word $w = w[1]\, w[2] \ldots w[n] \in \Pi^*$ satisfies LTL formula F at position (step) $1 \leq k \leq n$, written $w, k \models F$, iff:

$$
\begin{array}{rcl}
w, k \models p & \text{iff} & p = w[k] \\
w, k \models \neg F & \text{iff} & w, k \not\models F \\
w, k \models F_1 \wedge F_2 & \text{iff} & w, k \models F_1 \text{ and } w, k \models F_2 \\
w, k \models F_1 \vee F_2 & \text{iff} & w, k \models F_1 \text{ or } w, k \models F_2 \\
w, k \models F_1 \Longrightarrow F_2 & \text{iff} & w, k \models \neg F_1 \vee F_2 \\
w, k \models \mathsf{X}\, F & \text{iff} & k < n \text{ and } w, k+1 \models F \\
w, k \models \square\, F & \text{iff} & \text{for all } k \leq h \leq n\colon\ w, h \models F \\
w, k \models \lozenge\, F & \text{iff} & \text{for some } k \leq h \leq n\colon\ w, h \models F \\
w, k \models F_1 \,\mathsf{U}\, F_2 & \text{iff} & \text{for some } k \leq h \leq n\colon\ w, h \models F_2 \\
 & & \text{and, for all } k \leq j < h\colon\ w, j \models F_1
\end{array}
$$

LTL satisfaction: example

$$\begin{array}{ccccccc} w & = & \mathit{door} & \mathit{door} & \mathit{door} & \mathit{power} \\ & & w[1] & w[2] & w[3] & w[4] \end{array}$$

$w, 1 \models \mathit{door}$	$w, 2 \models \mathit{door}$	$w, 3 \models \mathit{door}$	$w, 4 \models \mathit{power}$
$w, 1 \models \neg \mathit{power}$	$w, 2 \models \neg \mathit{cook}$	$w, 4 \models \begin{array}{l} \mathit{door} \\ \vee \mathit{power} \end{array}$	$w, 4 \models \begin{array}{l} \mathit{power} \\ \wedge \neg \mathit{door} \end{array}$
$w, 1 \models \mathsf{X}\, \mathit{door}$	$w, 2 \models \mathsf{X}(\neg \mathit{cook})$	$w, 3 \models \mathsf{X}\, \mathit{power}$	$w, 4 \models \neg(\mathsf{X} \top)$
$w, 1 \models \Box(\neg \mathit{cook})$	$w, 2 \models \Box \begin{pmatrix} \mathit{door} \\ \vee \top \end{pmatrix}$	$w, 3 \models \neg \Box\, \mathit{door}$	$w, 4 \models \Box\, \mathit{power}$
$w, 1 \models \Diamond\, \mathit{door}$	$w, 2 \models \Diamond(\neg \mathit{door})$	$w, 3 \models \Diamond\, \mathsf{X}\, \mathit{power}$	$w, 4 \models \Diamond\, \mathit{power}$
$w, 1 \models \mathit{door}\ \mathsf{U}\ \mathit{power}$	$w, 2 \models \top\ \mathsf{U}\ \mathit{power}$	$w, 3 \models \bot\ \mathsf{U}\ \top$	$w, 4 \models \mathit{power}\ \mathsf{U}\ \mathit{power}$

Linear temporal logic: semantics

A word w satisfies an LTL formula F if it satisfies it initially:

$$w \models F \quad \text{iff} \quad w, 1 \models F$$

Linear temporal logic: semantics

A word w satisfies an LTL formula F if it satisfies it initially:

$$w \models F \quad \text{iff} \quad w, 1 \models F$$

The language of a linear temporal logic formula F is the set of words that satisfy F:

$$\mathcal{L}(F) = \{w \in \Pi^* \mid w \models F\}$$

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\Box\, p$				
$\mathsf{X}\, q$				
$\mathsf{X}(\Box\, p)$				
$\Box(\mathsf{X}\, p)$				
$\Diamond\, p$				
$\Diamond(\mathsf{X}\, p)$				
$p\,\mathsf{U}\,q$				

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\Box p$	p	$p\,p\,p$	ϵ	$p\,q\,p$
$\mathsf{X}\,q$				
$\mathsf{X}(\Box p)$				
$\Box(\mathsf{X}\,p)$				
$\Diamond p$				
$\Diamond(\mathsf{X}\,p)$				
$p\,\mathsf{U}\,q$				

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\square\, p$	p	$p\,p\,p$	ϵ	$p\,q\,p$
$\mathsf{X}\, q$	$p\,q$	$q\,q\,q$	$p\,q\,p\,p\,q$	$q\,p\,q\,q\,q$
$\mathsf{X}(\square\, p)$				
$\square(\mathsf{X}\, p)$				
$\lozenge\, p$				
$\lozenge(\mathsf{X}\, p)$				
$p\,\mathsf{U}\,q$				

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\Box\, p$	p	$p\,p\,p$	ϵ	$p\,q\,p$
$\mathsf{X}\, q$	$p\,q$	$q\,q\,q$	$p\,q\,p\,p\,q$	$q\,p\,q\,q\,q$
$\mathsf{X}(\Box\, p)$	$q\,p$	$p\,p\,p$	p	$p\,q\,p\,p$
$\Box(\mathsf{X}\, p)$				
$\Diamond\, p$				
$\Diamond(\mathsf{X}\, p)$				
$p\,\mathsf{U}\,q$				

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\square\, p$	p	$p\,p\,p$	ϵ	$p\,q\,p$
$\mathsf{X}\, q$	$p\,q$	$q\,q\,q$	$p\,q\,p\,p\,q$	$q\,p\,q\,q\,q$
$\mathsf{X}(\square\, p)$	$q\,p$	$p\,p\,p$	p	$p\,q\,p\,p$
$\square(\mathsf{X}\, p)$	ϵ			$p\,p\,p$
$\diamond\, p$				
$\diamond(\mathsf{X}\, p)$				
$p\,\mathsf{U}\,q$				

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\square\, p$	p	$p\,p\,p$	ϵ	$p\,q\,p$
$\mathsf{X}\, q$	$p\,q$	$q\,q\,q$	$p\,q\,p\,p\,q$	$q\,p\,q\,q\,q$
$\mathsf{X}(\square\, p)$	$q\,p$	$p\,p\,p$	p	$p\,q\,p\,p$
$\square(\mathsf{X}\, p)$	ϵ			$p\,p\,p$
$\lozenge\, p$	p	$q\,q\,q\,q\,p\,q$	$q\,p\,p$	ϵ
$\lozenge(\mathsf{X}\, p)$				
$p\, \mathsf{U}\, q$				

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\Box\, p$	p	$p\,p\,p$	ϵ	$p\,q\,p$
$\mathsf{X}\, q$	$p\,q$	$q\,q\,q$	$p\,q\,p\,p\,q$	$q\,p\,q\,q\,q$
$\mathsf{X}(\Box\, p)$	$q\,p$	$p\,p\,p$	p	$p\,q\,p\,p$
$\Box(\mathsf{X}\, p)$	ϵ			$p\,p\,p$
$\Diamond\, p$	p	$q\,q\,q\,q\,p\,q$	$q\,p\,p$	ϵ
$\Diamond(\mathsf{X}\, p)$	$p\,p$	$q\,q\,q\,p\,q$	$q\,p$	$p\,q\,q$
$p\,\mathsf{U}\,q$				

LTL semantics: examples

Examples of words that satisfy:

F	$w_1 \models F$	$w_2 \models F$	$w_3 \models F$	$w_4 \not\models F$
$\Box\, p$	p	$p\,p\,p$	ϵ	$p\,q\,p$
$\mathsf{X}\, q$	$p\,q$	$q\,q\,q$	$p\,q\,p\,p\,q$	$q\,p\,q\,q\,q$
$\mathsf{X}(\Box\, p)$	$q\,p$	$p\,p\,p$	p	$p\,q\,p\,p$
$\Box(\mathsf{X}\, p)$	ϵ			$p\,p\,p$
$\Diamond\, p$	p	$q\,q\,q\,q\,p\,q$	$q\,p\,p$	ϵ
$\Diamond(\mathsf{X}\, p)$	$p\,p$	$q\,q\,q\,p\,q$	$q\,p$	$p\,q\,q$
$p\,\mathsf{U}\,q$	$p\,p\,q$	q	$p\,p\,p\,p\,q$	$p\,p\,p$

Automata-based model checking

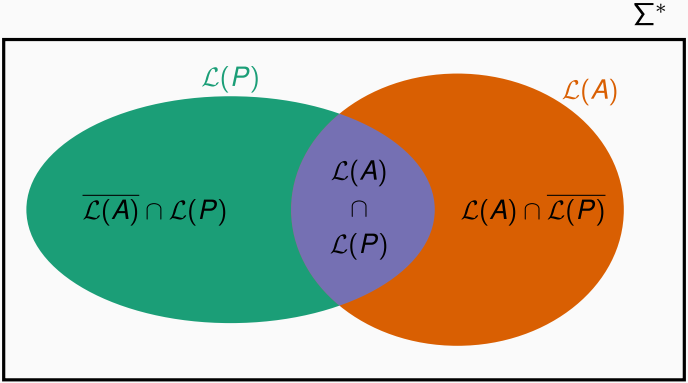
Model-checking algorithm

Linear-time model checking, semantically

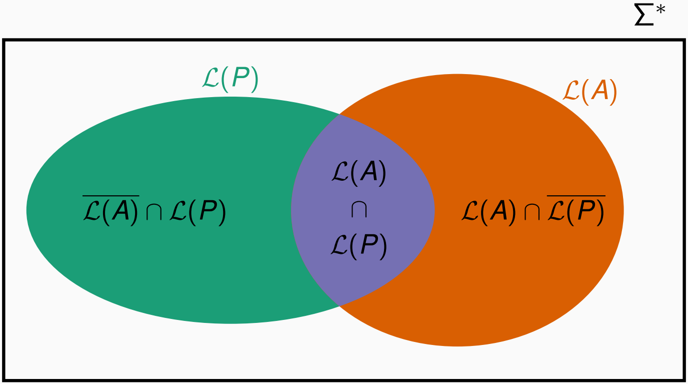
Linear-time model checking problem: given

- A: a finite-state automaton with alphabet Σ
- P: a linear temporal logic property over propositions in Σ

determine if $A \models P$: every word accepted by A satisfies P

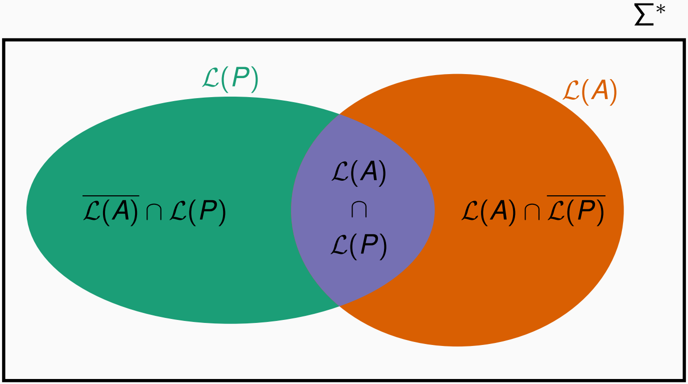


Linear-time model checking, semantically



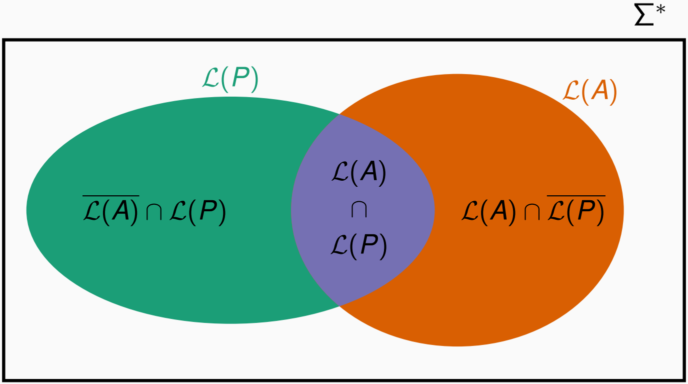
$A \models P$

Linear-time model checking, semantically



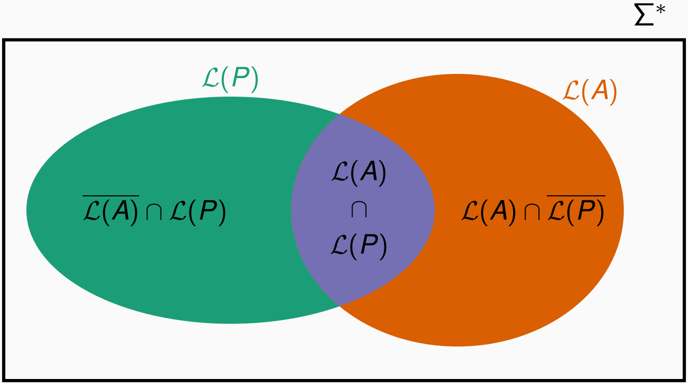
$A \models P$ iff $w \models A$ implies $w \models P$

Linear-time model checking, semantically



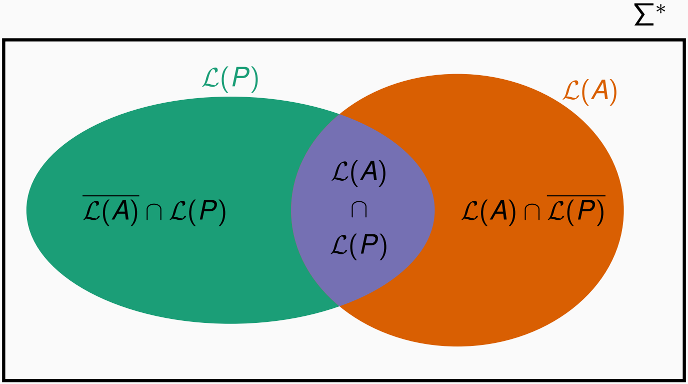
$$
\begin{aligned}
A \models P \quad & \text{iff} \quad w \models A \text{ implies } w \models P \\
& \text{iff} \quad w \in \mathcal{L}(A) \text{ implies } w \in \mathcal{L}(P)
\end{aligned}
$$

Linear-time model checking, semantically



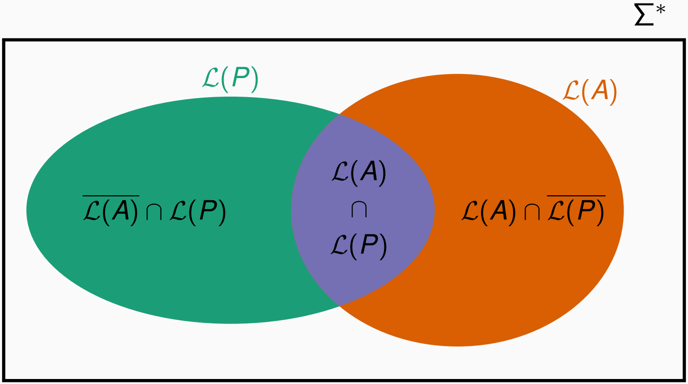
$$
\begin{aligned}
A \models P \quad & \text{iff} \quad w \models A \text{ implies } w \models P \\
& \text{iff} \quad w \in \mathcal{L}(A) \text{ implies } w \in \mathcal{L}(P) \\
& \text{iff} \quad \mathcal{L}(A) \subseteq \mathcal{L}(P)
\end{aligned}
$$

Linear-time model checking, semantically



$$
\begin{aligned}
A \models P \quad & \text{iff} \quad w \models A \text{ implies } w \models P \\
& \text{iff} \quad w \in \mathcal{L}(A) \text{ implies } w \in \mathcal{L}(P) \\
& \text{iff} \quad \mathcal{L}(A) \subseteq \mathcal{L}(P) \\
& \text{iff} \quad \mathcal{L}(A) \cap \overline{\mathcal{L}(P)} = \emptyset
\end{aligned}
$$

Linear-time model checking, semantically



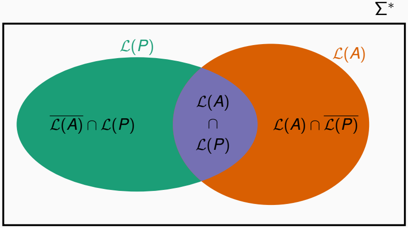
$$
\begin{aligned}
A \models P \quad & \text{iff} \quad w \models A \text{ implies } w \models P \\
& \text{iff} \quad w \in \mathcal{L}(A) \text{ implies } w \in \mathcal{L}(P) \\
& \text{iff} \quad \mathcal{L}(A) \subseteq \mathcal{L}(P) \\
& \text{iff} \quad \mathcal{L}(A) \cap \overline{\mathcal{L}(P)} = \emptyset \\
& \text{iff} \quad \mathcal{L}(A) \cap \mathcal{L}(\neg P) = \emptyset
\end{aligned}
$$

Model checking as emptiness checking

Linear-time model checking: given

- A: a finite-state automaton with alphabet Σ
- P: a linear temporal logic property over propositions in Σ

$\mathcal{L}(A) \cap \mathcal{L}(\neg P)$ is empty	$\mathcal{L}(A) \cap \mathcal{L}(\neg P)$ is not empty
every word accepted by A satisfies P	some word accepted by A does not satisfy P
✔ $A \models P$	✖ $A \not\models P$
	every word in $\mathcal{L}(A) \cap \mathcal{L}(\neg P)$ is a counterexample



Model-checking algorithm

Expressing the model-checking problem as emptiness checking suggests a technique for model checking which combines three algorithms:

MONITOR: given a temporal logic formula P build an automaton $\mathcal{A}(P)$ such that $\mathcal{L}(\mathcal{A}(P)) = \mathcal{L}(P)$

INTERSECTION: given automata A and B, build an automaton $A \times B$ such that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$

EMPTINESS: given an automaton A determine whether $\mathcal{L}(A) = \emptyset$

Model-checking algorithm

Expressing the model-checking problem as emptiness checking suggests a technique for model checking which combines three algorithms:

MONITOR: given a temporal logic formula P build an automaton $\mathcal{A}(P)$ such that $\mathcal{L}(\mathcal{A}(P)) = \mathcal{L}(P)$

INTERSECTION: given automata A and B, build an automaton $A \times B$ such that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$

EMPTINESS: given an automaton A determine whether $\mathcal{L}(A) = \emptyset$

Model-checking algorithm: given a finite-state automaton A and a linear temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

$$\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset \quad \text{iff} \quad A \models P$$

Monitors: from temporal logic to automata

Given an LTL formula P it is always possible to build a monitor of P: an FSA $\mathcal{A}(P)$ that accepts precisely the words that satisfy P.

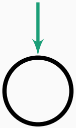
There are various algorithms to build monitors. We won't describe any of them in detail, but simply show their general ideas on some examples.

Monitors: always

$P_1 = \Box\, p$ $\quad$ p holds always

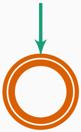
Monitors: always

$P_1 = \Box\, p$ $\quad$ p holds always



Monitors: always

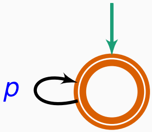
$P_1 = \Box\, p$ p holds always



- the empty word satisfies P_1

Monitors: always

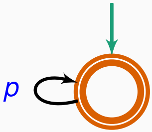
$P_1 = \square p$ $\quad$ p holds always



- the empty word satisfies P_1
- as long as p occurs, we accept

Monitors: always

$P_1 = \Box\, p$ $\quad$ p holds always



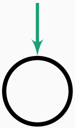
- the empty word satisfies P_1
- as long as p occurs, we accept
- no other transition is possible

Monitors: implication and next

$P_2 = \Box(p \Longrightarrow \mathsf{X}\, q)$ whenever p holds, q holds next

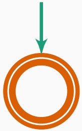
Monitors: implication and next

$P_2 = \square(p \Longrightarrow \mathsf{X}\, q)$ whenever p holds, q holds next



Monitors: implication and next

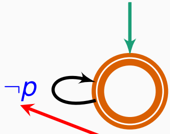
$P_2 = \square(p \Longrightarrow \mathsf{X}\, q)$ whenever p holds, q holds next



- the empty word satisfies P_2

Monitors: implication and next

$P_2 = \Box(p \Longrightarrow \mathsf{X}\, q)$ whenever p holds, q holds next

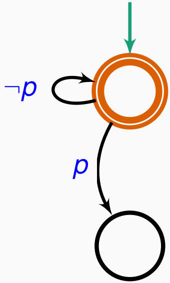


a transition for every proposition
$q \in \Pi$ such that $q \neq p$

- the empty word satisfies P_2
- as long as any proposition other than p occurs, the implication is trivially true

Monitors: implication and next

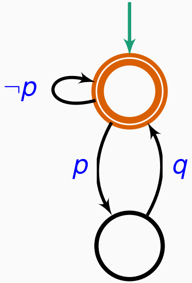
$$P_2 = \Box(p \Longrightarrow \mathsf{X}\, q) \qquad \text{whenever } p \text{ holds, } q \text{ holds next}$$



- the empty word satisfies P_2
- as long as any proposition other than p occurs, the implication is trivially true
- when p occurs, move to a different state ("next") which is not accepting

Monitors: implication and next

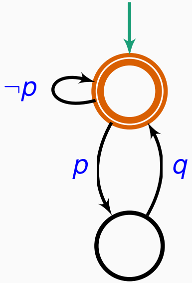
$P_2 = \square(p \Longrightarrow \mathsf{X}\, q)$ whenever p holds, q holds next



- the empty word satisfies P_2
- as long as any proposition other than p occurs, the implication is trivially true
- when p occurs, move to a different state ("next") which is not accepting
- now q must occur right away, which leads back to an accepting state

Monitors: implication and next

$P_2 = \square(p \Longrightarrow \mathsf{X}\, q)$ whenever p holds, q holds next



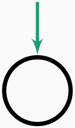
- the empty word satisfies P_2
- as long as any proposition other than p occurs, the implication is trivially true
- when p occurs, move to a different state ("next") which is not accepting
- now q must occur right away, which leads back to an accepting state
- no other transition is possible

Monitors: until

$P_3 = \square(p \Longrightarrow \mathsf{X}(o \mathsf{U} q))$ whenever p holds, o holds until q holds later

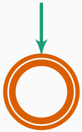
Monitors: until

$P_3 = \square(p \Longrightarrow \mathsf{X}(o \mathsf{U} q))$ whenever p holds, o holds until q holds later



Monitors: until

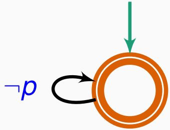
$P_3 = \Box(p \Longrightarrow \mathsf{X}(o\mathsf{U}q))$ whenever p holds, o holds until q holds later



- the empty word satisfies P_3

Monitors: until

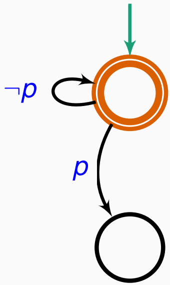
$P_3 = \Box(p \Longrightarrow \mathsf{X}(o \mathsf{U} q))$ whenever p holds, o holds until q holds later



- the empty word satisfies P_3
- as long as any proposition other than p occurs, the implication is trivially true

Monitors: until

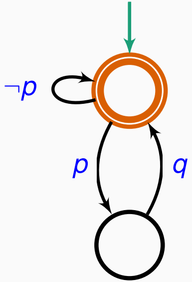
$P_3 = \square(p \Longrightarrow \mathsf{X}(o \mathsf{U} q))$ whenever p holds, o holds until q holds later



- the empty word satisfies P_3
- as long as any proposition other than p occurs, the implication is trivially true
- when p occurs, move to a different state ("next") which is not accepting

Monitors: until

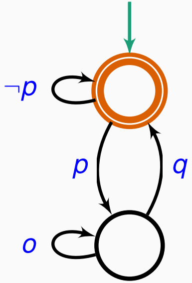
$P_3 = \Box(p \Longrightarrow \mathsf{X}(o \mathsf{U} q))$ whenever p holds, o holds until q holds later



- the empty word satisfies P_3
- as long as any proposition other than p occurs, the implication is trivially true
- when p occurs, move to a different state ("next") which is not accepting
- to go back to an accepting state, q must occur eventually

Monitors: until

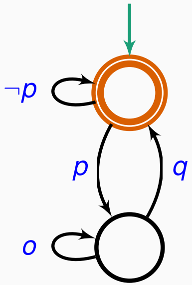
$P_3 = \Box(p \Longrightarrow X(o U q))$ whenever p holds, o holds until q holds later



- the empty word satisfies P_3
- as long as any proposition other than p occurs, the implication is trivially true
- when p occurs, move to a different state ("next") which is not accepting
- to go back to an accepting state, q must occur eventually
- until q occurs, only o may occur

Monitors: until

$P_3 = \square(p \Longrightarrow \mathsf{X}(o \mathsf{U} q))$ whenever p holds, o holds until q holds later



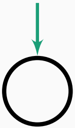
- the empty word satisfies P_3
- as long as any proposition other than p occurs, the implication is trivially true
- when p occurs, move to a different state ("next") which is not accepting
- to go back to an accepting state, q must occur <u>eventually</u>
- until q occurs, only o may occur
- no other transition is possible

Monitors: eventually and until

$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds

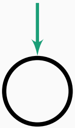
Monitors: eventually and until

$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds



Monitors: eventually and until

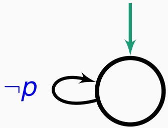
$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds



- the empty word does not satisfy P_4

Monitors: eventually and until

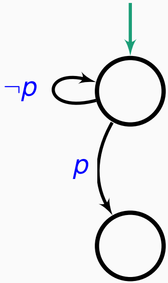
$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds



- the empty word does not satisfy P_4
- as long as any proposition other than p occurs, P_4 remains false

Monitors: eventually and until

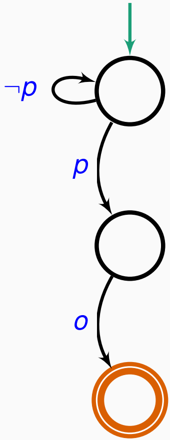
$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds



- the empty word does not satisfy P_4
- as long as any proposition other than p occurs, P_4 remains false
- p must eventually occur; then move to a different state ("next") which is also not accepting

Monitors: eventually and until

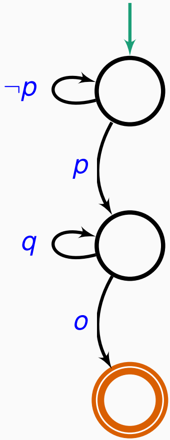
$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds



- the empty word does not satisfy P_4
- as long as any proposition other than p occurs, P_4 remains false
- p must eventually occur; then move to a different state ("next") which is also not accepting
- from in the new state, o must occur eventually, which finally leads to an accepting state

Monitors: eventually and until

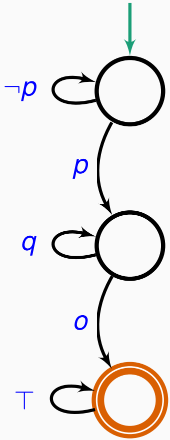
$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds



- the empty word does not satisfy P_4
- as long as any proposition other than p occurs, P_4 remains false
- p must eventually occur; then move to a different state ("next") which is also not accepting
- from in the new state, o must occur eventually, which finally leads to an accepting state
- until o occurs, only q may occur

Monitors: eventually and until

$P_4 = \Diamond(p \wedge \mathsf{X}(q \mathsf{U} o))$ eventually p holds, and then q holds until o holds



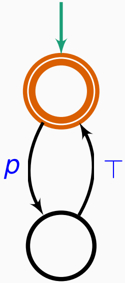
- the empty word does not satisfy P_4
- as long as any proposition other than p occurs, P_4 remains false
- p must eventually occur; then move to a different state ("next") which is also not accepting
- from in the new state, o must occur eventually, which finally leads to an accepting state
- until o occurs, only q may occur
- after we reach the accepting state, we never leave it – P_4 remains true forever

From automata to LTL?

While it is always possible to go from LTL to FSA, automata are strictly more expressive.

ODD: "p occurs in every odd time step $1, 3, 5, \ldots$"

Automaton accepting language *ODD*:



There is no LTL formula that is satisfied precisely by words in *ODD*.

In particular, formula

$$p \wedge \Box(p \Longrightarrow \mathrm{X}\,\mathrm{X}\,p)$$

is too strong because, once p occurs in an even time step, it will always occur afterwards.

Automata of negated properties

The model-checking algorithm computes the monitor of a negated property $\neg P$. There are two ways of doing this, either of which may be easier depending on the specific P:

1. build the monitor $\mathcal{A}(\neg P)$ of $\neg P$ directly
2. build the monitor $\mathcal{A}(P)$ of P, and then complement it, getting $\overline{\mathcal{A}(P)} = \mathcal{A}(\neg P)$

Automata of negated properties

The model-checking algorithm computes the monitor of a negated property $\neg P$. There are two ways of doing this, either of which may be easier depending on the specific P:

1. build the monitor $\mathcal{A}(\neg P)$ of $\neg P$ directly
2. build the monitor $\mathcal{A}(P)$ of P, and then complement it, getting $\overline{\mathcal{A}(P)} = \mathcal{A}(\neg P)$

To build the monitor of the negated property directly, the following equivalences may be useful:

$$\neg \Diamond p \equiv \Box \neg p \qquad \neg \Box p \equiv \Diamond \neg p$$

$$\neg \mathsf{X}\, p \equiv \mathsf{X}(\neg p) \;\vee\; \neg \mathsf{X} \top \qquad \Diamond p \equiv \top \,\mathsf{U}\, p$$

holds in the last position of a word, or on the empty word

Complementing automata

The complement automaton $\overline{A}$ accepts
the language $\overline{\mathcal{L}(A)}$ of all words that A rejects

To build the complement automaton from A:

1. build A_1: an automaton equivalent to A that is deterministic and with total transition function
2. build A_2 by switching accepting and non-accepting states of A_1
3. then $A_2 = \overline{A}$

Complementing automata

The complement automaton $\overline{A}$ accepts
the language $\overline{\mathcal{L}(A)}$ of all words that A rejects

To build the complement automaton from A:

1. build A_1: an automaton equivalent to A that is deterministic and with total transition function
2. build A_2 by switching accepting and non-accepting states of A_1
3. then $A_2 = \overline{A}$

To make a transition function ρ total:

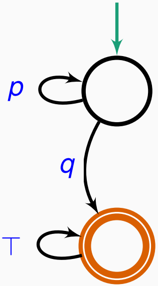
1. add a special error state $e \notin S$
2. add a loop $\rho(e, \sigma) = \{e\}$ for every $\sigma \in \Sigma$
3. for every s, σ such that $\rho(s, \sigma) = \{\}$ (or undefined), define the transition $\rho(s, \sigma) = \{e\}$

Complementing automata: example

Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.

Complementing automata: example

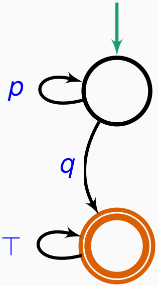
Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$

Complementing automata: example

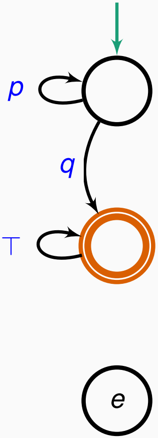
Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$
2. make the transition function total

Complementing automata: example

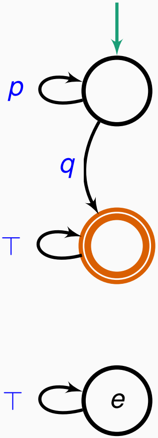
Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$
2. make the transition function total

Complementing automata: example

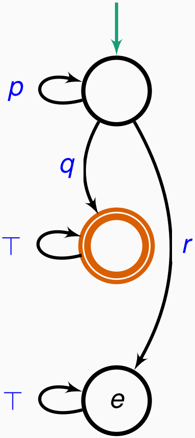
Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$
2. make the transition function total

Complementing automata: example

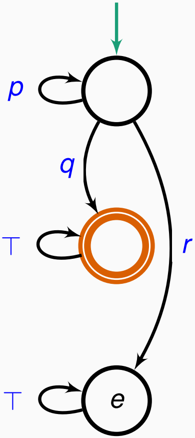
Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$
2. make the transition function total

Complementing automata: example

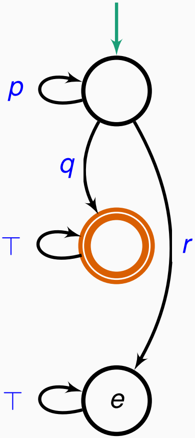
Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$
2. make the transition function total
3. check that it's deterministic

Complementing automata: example

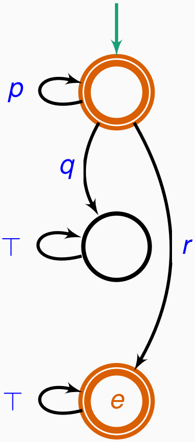
Build the monitor of $P_6 = \neg P_5 = \neg\big(p \,\mathsf{U}\, q\big)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$
2. make the transition function total
3. check that it's deterministic
4. complement its accepting and non-accepting states

Complementing automata: example

Build the monitor of $P_6 = \neg P_5 = \neg(p \,\mathsf{U}\, q)$ by complementing $\mathcal{A}(P_5)$.
The input alphabet Σ is $\{p, q, r\}$.



1. build the monitor $\mathcal{A}(P_5)$ of $P_5 = p \,\mathsf{U}\, q$
2. make the transition function total
3. check that it's deterministic
4. complement its accepting and non-accepting states

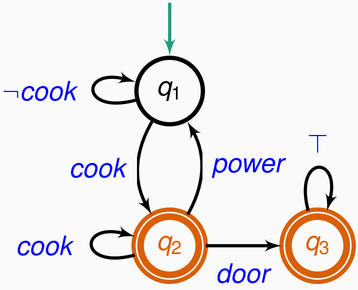
Monitor: running example

Let's build the monitors for two variants of properties that we would like to model check in the microwave running example.

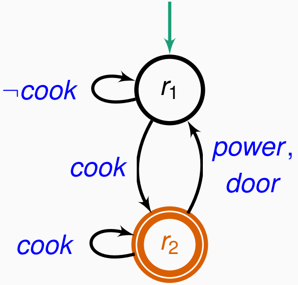
$$Q = \Box(\mathit{cook} \Longrightarrow \mathit{cook} \,\mathsf{U}\, \mathit{power})$$
$$\neg Q = \Diamond(\mathit{cook} \wedge \neg(\mathit{cook} \,\mathsf{U}\, \mathit{power}))$$

$$R = \Box(\mathit{cook} \Longrightarrow \mathit{cook} \,\mathsf{U}\, \mathit{power} \vee \mathit{door})$$
$$\neg R = \Diamond(\mathit{cook} \wedge \neg(\mathit{cook} \,\mathsf{U}\, \mathit{power} \vee \mathit{door}))$$

Automaton $\mathcal{A}(\neg Q)$ built by complementing $\mathcal{A}(Q)$:



Automaton $\mathcal{A}(\neg R)$ built by complementing $\mathcal{A}(R)$:



Intersection: running automata in parallel

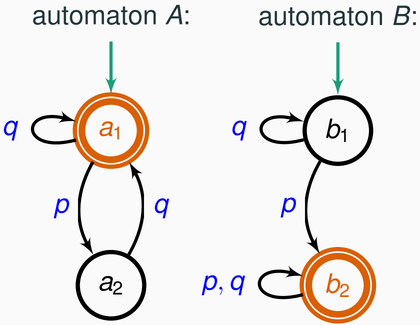
An automaton C that accepts the intersection of two automata A and B's languages runs A and B in parallel:

- starts from any combination of initial states of A and B
- transitions only when both A and B have a transition for the current input
- accepts when both A and B accept

Intersection: running automata in parallel

An automaton C that accepts the intersection of two automata A and B's languages runs A and B in parallel:

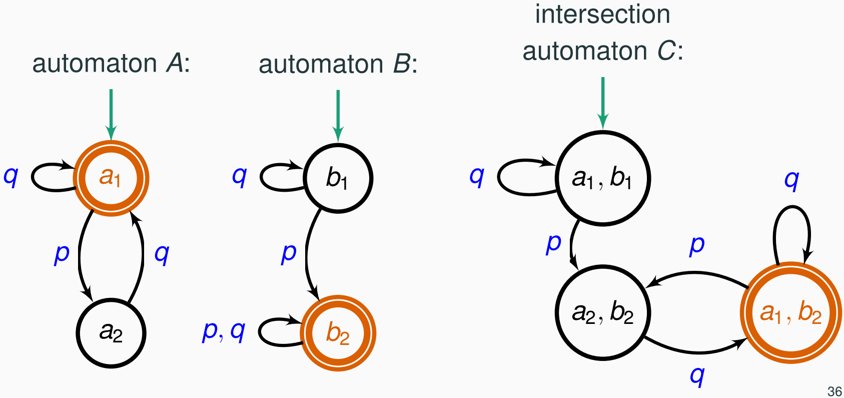
- starts from any combination of initial states of A and B
- transitions only when both A and B have a transition for the current input
- accepts when both A and B accept



Intersection: running automata in parallel

An automaton C that accepts the intersection of two automata A and B's languages runs A and B in parallel:

- starts from any combination of initial states of A and B
- transitions only when both A and B have a transition for the current input
- accepts when both A and B accept



Product automaton construction

Given FSAs $A = \langle \Sigma, S_A, I_A, F_A, \rho_A \rangle$ and $B = \langle \Sigma, S_B, I_B, F_B, \rho_B \rangle$, the product automaton $A \times B = \langle \Sigma, S, I, F, \rho, \rangle$ is defined as:

$$
\begin{aligned}
S &= S_A \times S_B \\
I &= \{(a, b) \mid a \in I_A \text{ and } b \in I_B\} \\
F &= \{(a, b) \mid a \in F_A \text{ and } b \in F_B\} \\
\rho((a, b), \sigma) &= \{(a_2, b_2) \mid a_2 \in \rho_A(a, \sigma) \text{ and } b_2 \in \rho_B(b, \sigma)\}
\end{aligned}
$$

Product automaton construction

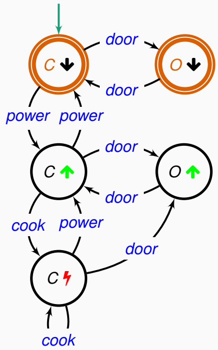
Given FSAs $A = \langle \Sigma, S_A, I_A, F_A, \rho_A \rangle$ and $B = \langle \Sigma, S_B, I_B, F_B, \rho_B \rangle$, the product automaton $A \times B = \langle \Sigma, S, I, F, \rho, \rangle$ is defined as:

$$\begin{aligned}
S &= S_A \times S_B \\
I &= \{(a, b) \mid a \in I_A \text{ and } b \in I_B\} \\
F &= \{(a, b) \mid a \in F_A \text{ and } b \in F_B\} \\
\rho((a, b), \sigma) &= \{(a_2, b_2) \mid a_2 \in \rho_A(a, \sigma) \text{ and } b_2 \in \rho_B(b, \sigma)\}
\end{aligned}$$

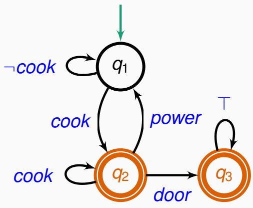
The language of the product automaton is the intersection of the intersected automata's languages:

$$\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$$

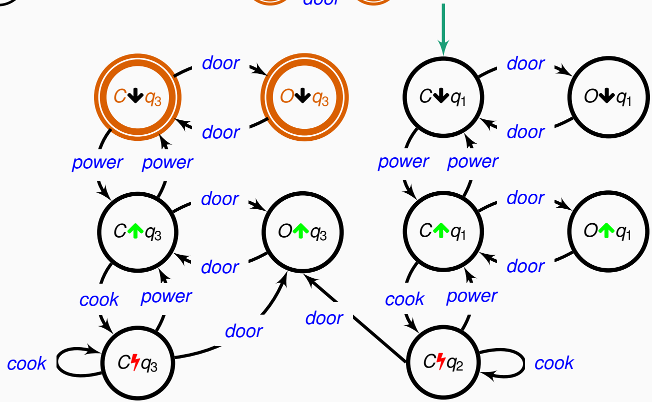
Product automaton of microwave and $\mathcal{A}(\neg Q)$



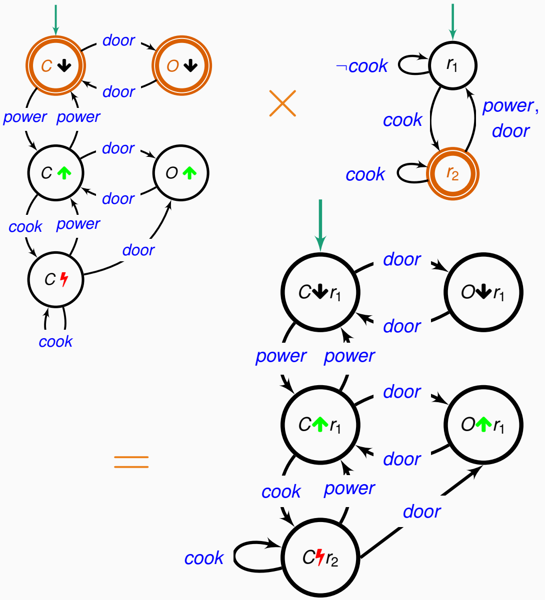
×



=



Product automaton of microwave and $\mathcal{A}(\neg R)$



Emptiness: reachability on graph

An automaton $A = \langle \Sigma, S, I, F, \rho \rangle$ accepts the empty language iff there is no final state $f \in F$ that is reachable from any initial state $i \in I$ on the directed graph representing A's transitions

If we find a directed path from some $i \in I$ to some $f \in F$, following it gives a word that is accepted by A.

Emptiness: reachability on graph

An automaton $A = \langle \Sigma, S, I, F, \rho \rangle$ accepts the empty language iff there is no final state $f \in F$ that is reachable from any initial state $i \in I$ on the directed graph representing A's transitions

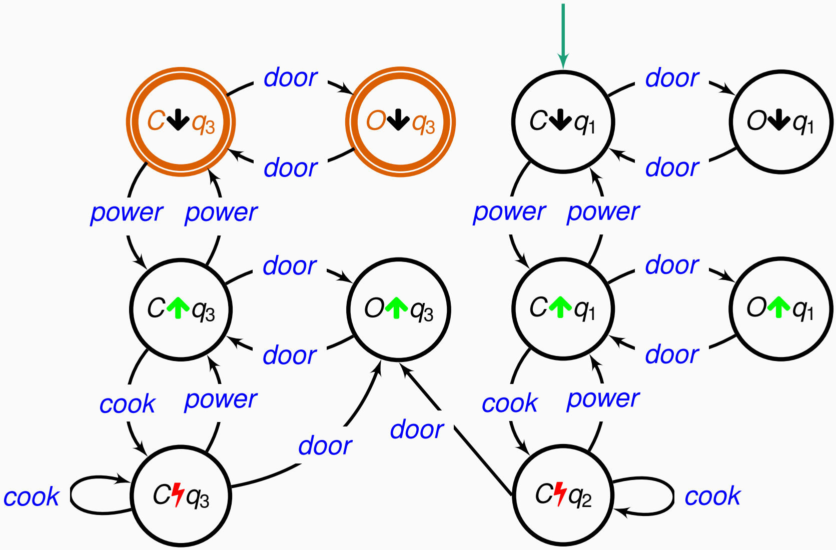
If we find a directed path from some $i \in I$ to some $f \in F$, following it gives a word that is accepted by A.

In the overall model-checking algorithm, we check emptiness of

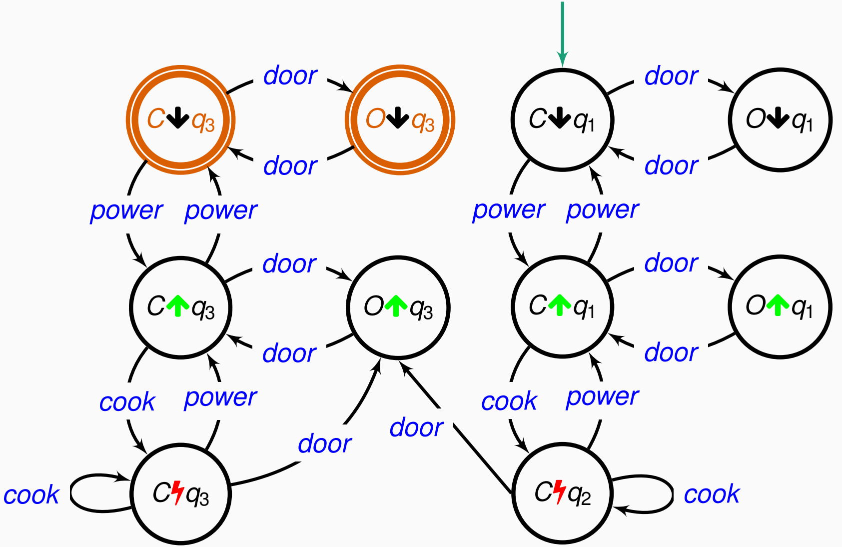
$$C = A \times \mathcal{A}(\neg P)$$

- if $\mathcal{L}(C)$ is empty, we conclude $A \models P$
- if $\mathcal{L}(C)$ is not empty, any accepting path in C gives a counterexample word w such that $w \models A$ and $w \not\models P$

Emptiness: microwave example with property Q

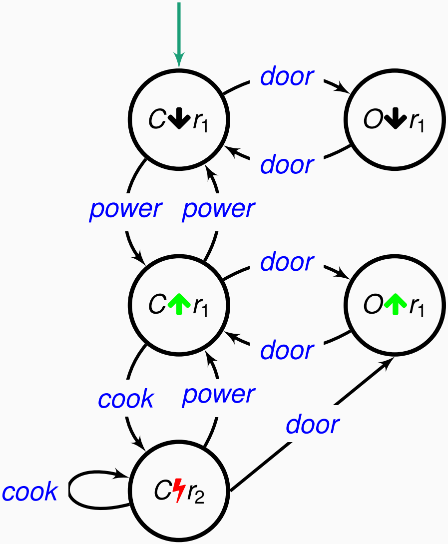


Emptiness: microwave example with property Q

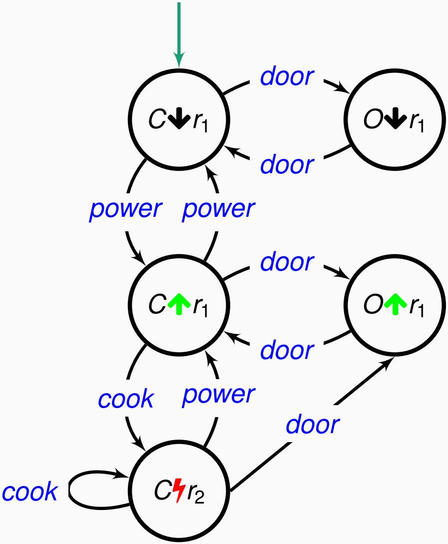


$A \not\models Q$ Counterexample: *power cook door door power*

Emptiness: microwave example with property R



Emptiness: microwave example with property R



$A \models R$

Model checking: static or dynamic?

Static:

- without executing the software
- based on symbolic constraints on states
- typically sound

Dynamic:

- while executing the software
- based on enumerating concrete states
- typically complete

Model checking: static or dynamic?

Static:

- without executing the software
- based on symbolic constraints on states
- typically sound

Dynamic:

- while executing the software
- based on enumerating concrete states
- typically complete

Model checking combines elements of static and dynamic analysis:

- it performs an exhaustive analysis
- the analysis is based on enumerating concrete states
- the concrete states abstract some aspect of the software
- the model-checking algorithm executes all possible system runs

Model checking: static or dynamic?

Static:

- without executing the software
- based on symbolic constraints on states
- typically sound

Dynamic:

- while executing the software
- based on enumerating concrete states
- typically complete

Model checking combines elements of static and dynamic analysis:

- it performs an exhaustive analysis
- the analysis is based on enumerating concrete states
- the concrete states abstract some aspect of the software
- the model-checking algorithm executes all possible system runs

model checking

STATIC → DYNAMIC

static analysis	deductive verification	software model checking	symbolic execution	dynamic analysis

Soundness and completeness

Model checking is sound and complete:

sound: if $\mathcal{L}(A \times \mathcal{A}(\neg P))$ is empty, $A \models P$

complete: if $\mathcal{L}(A \times \mathcal{A}(\neg P))$ is not empty, $A \not\models P$ and we get a counterexample

It's possible to have both soundness and completeness because analysis of finite-state models is decidable: model checking can be seen as exhaustive testing, which is possible for finite-state models.

Soundness and completeness

Model checking is sound and complete:

sound: if $\mathcal{L}(A \times \mathcal{A}(\neg P))$ is empty, $A \models P$

complete: if $\mathcal{L}(A \times \mathcal{A}(\neg P))$ is not empty, $A \not\models P$ and we get a counterexample

It's possible to have both soundness and completeness because analysis of finite-state models is decidable: model checking can be seen as exhaustive testing, which is possible for finite-state models.

If the finite-state model A that we model check is an abstraction of a more complex infinite-state program, then the analysis of model-checking may be not sound or complete for the program. We will analyze this in detail when presenting software model checking.

Complexity of the model-checking algorithm

Model-checking algorithm: given a finite-state automaton A and a linear temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

Complexity of the model-checking algorithm

Model-checking algorithm: given a finite-state automaton A and a linear temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

EMPTINESS is equivalent to reachability, which can be done time linear in the size $|A|$ of the automaton

Complexity of the model-checking algorithm

Model-checking algorithm: given a finite-state automaton A and a linear temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

EMPTINESS is equivalent to reachability, which can be done time linear in the size $|A|$ of the automaton

INTERSECTION of two automata $|A|$ and $|B|$ creates an automaton of size $\mathrm{O}(|A| \cdot |B|)$

Complexity of the model-checking algorithm

Model-checking algorithm: given a finite-state automaton A and a linear temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

EMPTINESS is equivalent to reachability, which can be done time linear in the size $|A|$ of the automaton

INTERSECTION of two automata $|A|$ and $|B|$ creates an automaton of size $\mathrm{O}(|A| \cdot |B|)$

MONITOR construction for an LTL formula F produces an automaton $\mathcal{A}(F)$ of size not larger than $2^{\mathrm{O}(|F|)}$

Complexity of the model-checking algorithm

Model-checking algorithm: given a finite-state automaton A and a linear temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

EMPTINESS is equivalent to reachability, which can be done time linear in the size $|A|$ of the automaton

INTERSECTION of two automata $|A|$ and $|B|$ creates an automaton of size $\mathrm{O}(|A| \cdot |B|)$

MONITOR construction for an LTL formula F produces an automaton $\mathcal{A}(F)$ of size not larger than $2^{\mathrm{O}(|F|)}$

The automata-based model checking algorithm to decide whether $A \models F$ runs in time $\mathrm{O}\big(|A| \cdot 2^{\mathrm{O}(|F|)}\big)$

Complexity of the model-checking problem

What is the worst-case complexity of the model-checking problem?

The model-checking problem for finite-state automata
and LTL formulas is PSPACE-complete

Complexity of the model-checking problem

What is the worst-case complexity of the model-checking problem?

The model-checking problem for finite-state automata
and LTL formulas is PSPACE-complete

EMPTINESS is reachability, which can be done in linear time (and space) with depth-first or breadth-first search, and is NLOGSPACE-complete

Complexity of the model-checking problem

What is the worst-case complexity of the model-checking problem?

The model-checking problem for finite-state automata
and LTL formulas is PSPACE-complete

EMPTINESS is reachability, which can be done in linear time (and space) with depth-first or breadth-first search, and is NLOGSPACE-complete

INTERSECTION has to keep track of all possible combinations of states, which is proportional to the product of the intersected automata's sizes in the worst case

Complexity of the model-checking problem

What is the worst-case complexity of the model-checking problem?

> The model-checking problem for finite-state automata and LTL formulas is PSPACE-complete

EMPTINESS is reachability, which can be done in linear time (and space) with depth-first or breadth-first search, and is NLOGSPACE-complete

INTERSECTION has to keep track of all possible combinations of states, which is proportional to the product of the intersected automata's sizes in the worst case

MONITOR construction introduces an exponential blow-up because conjunction of LTL formulas requires to monitor both conjoined formulas in parallel, which has multiplicative complexity

Complexity of the model-checking problem

What is the worst-case complexity of the model-checking problem?

The model-checking problem for finite-state automata and LTL formulas is PSPACE-complete

EMPTINESS is reachability, which can be done in linear time (and space) with depth-first or breadth-first search, and is NLOGSPACE-complete

INTERSECTION has to keep track of all possible combinations of states, which is proportional to the product of the intersected automata's sizes in the worst case

MONITOR construction introduces an exponential blow-up because conjunction of LTL formulas requires to monitor both conjoined formulas in parallel, which has multiplicative complexity

Thus, the model-checking algorithm we have presented is worst-case optimal.

Model-checking complexity in practice

The automata-based model checking algorithm
to decide whether $A \models F$ runs in time $\mathrm{O}\big(|A| \cdot 2^{\mathrm{O}(|F|)}\big)$

Model-checking complexity in practice

The automata-based model checking algorithm
to decide whether $A \models F$ runs in time $\mathrm{O}\left(|A| \cdot 2^{\mathrm{O}(|F|)}\right)$

In most cases, the LTL property F is much smaller than the system model A. Therefore, the exponential dependency is not a major problem in practice.

Model-checking complexity in practice

On the contrary, the combinatorial growth of the product construction often leads to a huge graph to check that does not fit memory.

To ameliorate this, the model-checking algorithm can be performed on the fly by constructing the graph incrementally while checking emptiness:

- construct monitor in depth-first order
- while constructing the monitor, construct product in depth-first order
- while constructing the product, check reachability in depth-first order

This way, only the necessary part of the graph is built.

The on-the-fly algorithm does not change the worst-case complexity but it likely to make model-checking feasible in practice – especially when a counterexample exists and can be found effectively.

Automata-based model checking

From programs to automata

Model checking for software analysis?

Finite-state models capture programs that use a bounded (finite and independent of input size) amount of memory.

How limiting is the restriction to finite-state models?

- Even when considering programs that are infinite state, a finite-state model can be useful to capture important behavioral features such as concurrency
- The small scope hypothesis suggests that model checking can be a very effective bug finding tool that provides high levels of assurance

Model checking for software analysis?

Finite-state models capture programs that use a bounded (finite and independent of input size) amount of memory.

How limiting is the restriction to finite-state models?

- Even when considering programs that are infinite state, a finite-state model can be useful to capture important behavioral features such as concurrency
- The small scope hypothesis suggests that model checking can be a very effective bug finding tool that provides high levels of assurance

Small scope hypothesis:
most bugs have small counterexamples
Daniel Jackson: Software Abstractions

Shared memory concurrency

Let's see how model checking can be applied to analyze properties of concurrent programs.

Shared memory concurrency

Let's see how model checking can be applied to analyze properties of concurrent programs.

The following concurrent program includes two processes that increment a `counter` variable in shared memory.

shared memory

```
var counter: Integer // initially 0
```

process t	process u
`var cnt: Integer`	`var cnt: Integer` ← local memory
1 `cnt := counter`	`cnt := counter` 3
2 `counter := cnt + 1;`	`counter := cnt + 1` 4

code

Shared memory concurrency

Let's see how model checking can be applied to analyze properties of concurrent programs.

The following concurrent program includes two processes that increment a `counter` variable in shared memory.

shared memory

```
var counter: Integer // initially 0
```

---

process t

```
  var cnt: Integer
1 cnt := counter
2 counter := cnt + 1;
```

code

process u

local memory

```
var cnt: Integer
cnt := counter            3
counter := cnt + 1        4
```

Each numbered line of code includes exactly one statement that can execute atomically.

Statements in different processes can be executed in any relative order.

Finite-state models of concurrent processes

Analyzing the behavior of concurrent programs require to reason about a finite but very large number of different execution orders.

For example, consider the analysis question:

> What is the value of `counter` after processes t and u terminate?

Finite-state models of concurrent processes

Analyzing the behavior of concurrent programs require to reason about a finite but very large number of different execution orders.

For example, consider the analysis question:

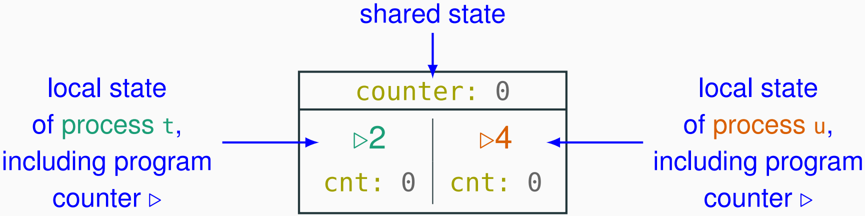
> What is the value of `counter` after processes *t* and *u* terminate?

To analyze properties of concurrent programs, we formalize essential elements of their behavior using a state/transition model similar to finite-state automata.

- states in a model correspond to program states
- transitions connect states according to execution order

Concurrent states

A state captures the shared and local states of a concurrent program:

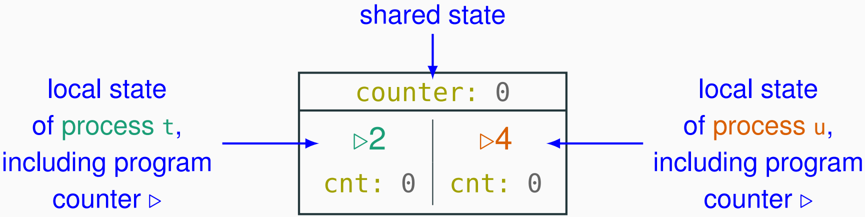



```
var counter: Integer // initially 0
```

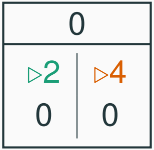
process t	process u
`var cnt: Integer`	`var cnt: Integer`
1 `cnt := counter`	`cnt := counter` 3
2 `counter := cnt + 1`	`counter := cnt + 1` 4

Concurrent states

A state captures the shared and local states of a concurrent program:

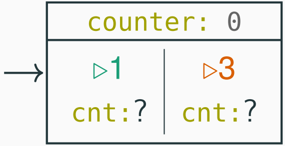


For simplicity, we may only keep a state's essential information:



Initial state

As with automata, we mark the initial state with an incoming arrow:




```
var counter: Integer // initially 0
```

process t

```
  var cnt: Integer
1 cnt := counter
2 counter := cnt + 1
```

process u

```
var cnt: Integer
cnt := counter          3
counter := cnt + 1      4
```

Final states

As with automata, the final states of a computation – when the program terminates – are marked with double-line edges:

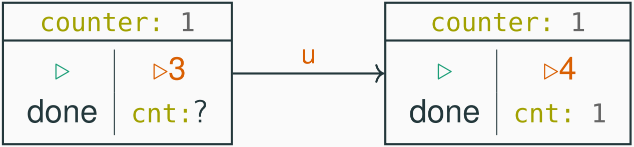
counter: 2	
▷ done	▷ done

```
var counter: Integer // initially 0
```

process t	process u
`var cnt: Integer`	`var cnt: Integer`
1 `cnt := counter`	`cnt := counter` 3
2 `counter := cnt + 1`	`counter := cnt + 1` 4

Transitions

A transition corresponds to the execution of one atomic instruction, and it is an arrow connecting two states (or a state to itself):

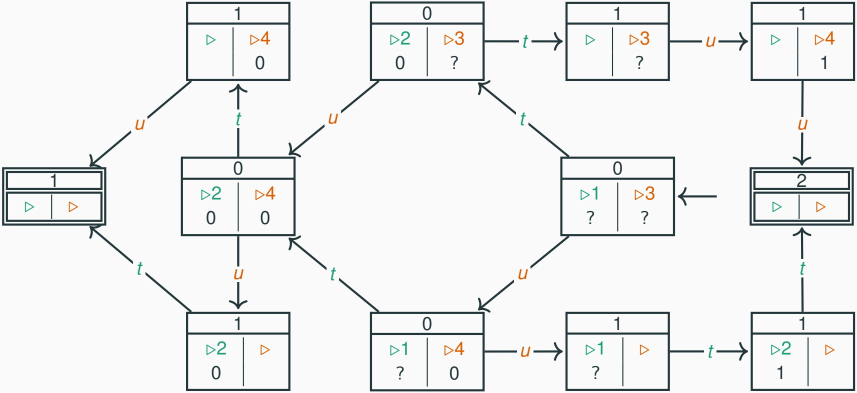



```
var counter: Integer // initially 0
```

process t	process u
`var cnt: Integer`	`var cnt: Integer`
1 `cnt := counter`	`cnt := counter` 3
2 `counter := cnt + 1`	`counter := cnt + 1` 4

A complete state/transition model

The complete state/transition model for the concurrent counter example explicitly shows all possible interleavings:



The labels on transitions indicate which process executes, but that information is subsumed by the transition's pre- and post-state.

Kripke structure

The state/transition model we have built is a Kripke structure (also know as state/transition diagram, or finite-state machine):

A Kripke structure K is a tuple $\langle S, I, R, P, L \rangle$:

- S: finite nonempty set of states
- $I \subseteq S$: set of initial states
- $R \subseteq S \times S$: transition relation
- P: a set of propositions
- $L\colon S \rightarrow \wp(P)$: labeling function

Kripke structure

The state/transition model we have built is a Kripke structure (also know as state/transition diagram, or finite-state machine):

A Kripke structure K is a tuple $\langle S, I, R, P, L \rangle$:

- S: finite nonempty set of states
- $I \subseteq S$: set of initial states
- $R \subseteq S \times S$: transition relation
- P: a set of propositions
- $L\colon S \to \wp(P)$: labeling function

There are no explicit final states – non-error states without outgoing transitions correspond to final computational states.

Kripke structure

The state/transition model we have built is a Kripke structure (also know as state/transition diagram, or finite-state machine):

A Kripke structure K is a tuple $\langle S, I, R, P, L \rangle$:

- S: finite nonempty set of states
- $I \subseteq S$: set of initial states
- $R \subseteq S \times S$: transition relation
- P: a set of propositions
- $L\colon S \to \wp(P)$: labeling function

There are no explicit final states – non-error states without outgoing transitions correspond to final computational states.

The labeling function assigns to each state $s \in S$ the set $L(s) \subseteq P$ of propositions that hold in s.

From Kripke structures to finite-state automata

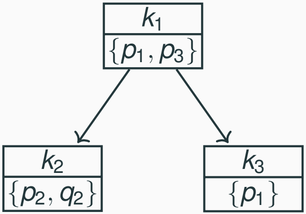
We can represent the computations of a Kripke structure $K = \langle S_K, I_K, R_K, P_K, L_K \rangle$ with an automaton $A = \langle \Sigma, S, I, F, \rho \rangle$.

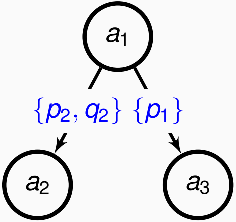
The basic idea is that we label each automata transition with a set of propositions corresponding to the propositions that label the state the transition enters.

From Kripke structures to finite-state automata

We can represent the computations of a Kripke structure $K = \langle S_K, I_K, R_K, P_K, L_K \rangle$ with an automaton $A = \langle \Sigma, S, I, F, \rho \rangle$.

The basic idea is that we label each automata transition with a set of propositions corresponding to the propositions that label the state the transition enters.





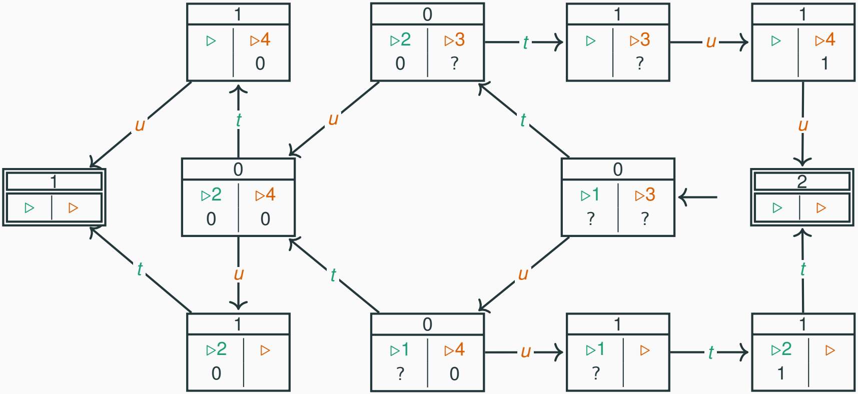
From Kripke structures to finite-state automata

We can represent the computations of a Kripke structure $K = \langle S_K, I_K, R_K, P_K, L_K \rangle$ with an automaton $A = \langle \Sigma, S, I, F, \rho \rangle$.

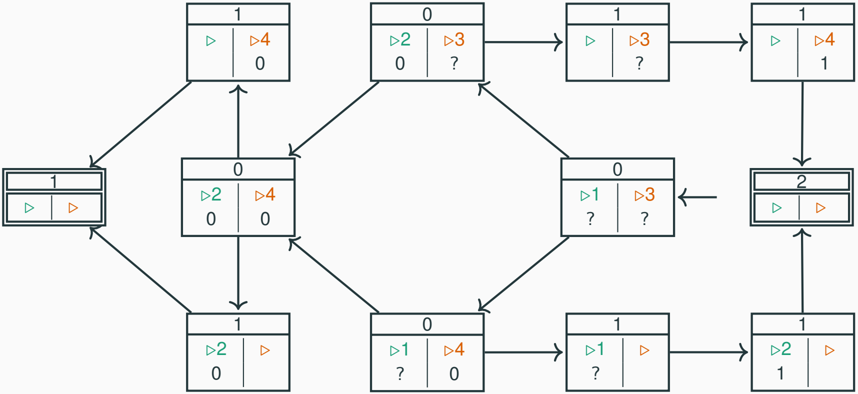
The basic idea is that we label each automata transition with a set of propositions corresponding to the propositions that label the state the transition enters.

- $\Sigma = \wp(P_K)$
- $S = S_k \cup \{s_0\}$
  s_0 is an initialization state that leads to initial states of K
- $I = \{s_0\}$
- $F = \{s \in S_k \mid \neg(s \; R \; s_2)$ for all s_2 and s is not an error state$\}$
  the final states are those without outgoing transitions
- $s \in \rho(s_0, L(s))$ iff $s \in I_K$
  the initialization state leads to K's initial states with
  an input corresponding to the proposition that hold there
- $s_2 \in \rho(s_1, L(s_2))$ iff $s_1 \; R \; s_2$: each transition takes an input corresponding to the proposition that hold in the state it reaches

Kripke structure to finite-state automaton: example



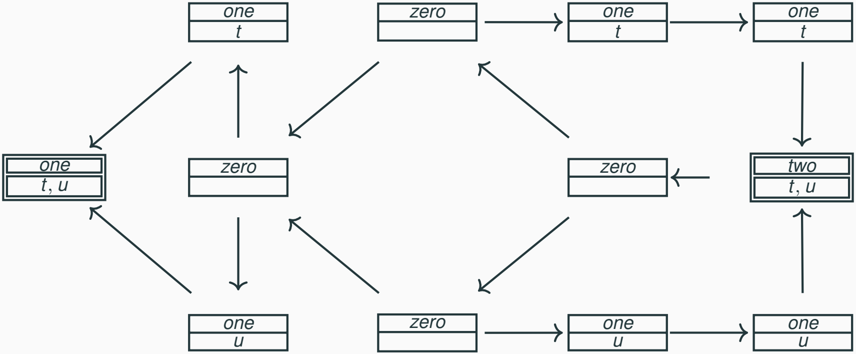
Kripke structure to finite-state automaton: example



Kripke structure to finite-state automaton: example

We keep track of predicates:

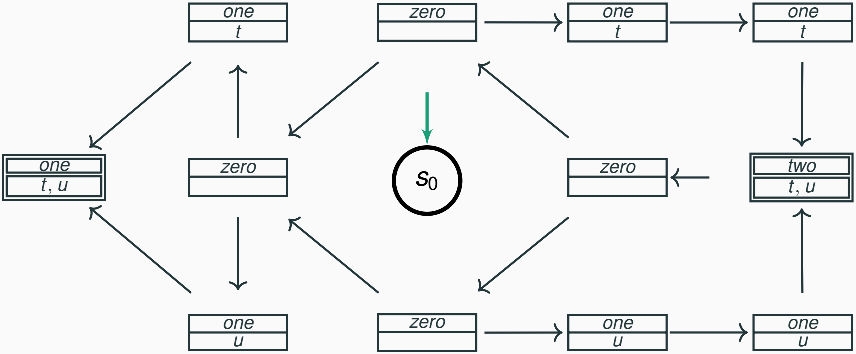
- *zero*, *one*, *two*: corresponding to the value of the counter
- t, u: process t, u has terminated



Kripke structure to finite-state automaton: example

We keep track of predicates:

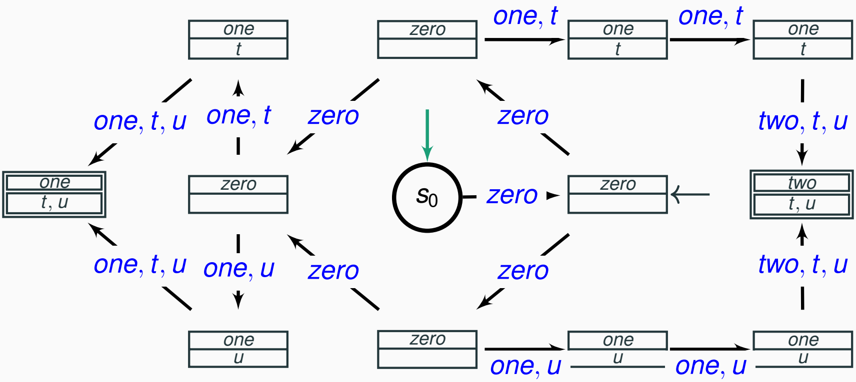
- *zero*, *one*, *two*: corresponding to the value of the counter
- *t*, *u*: process *t*, *u* has terminated



Kripke structure to finite-state automaton: example

We keep track of predicates:

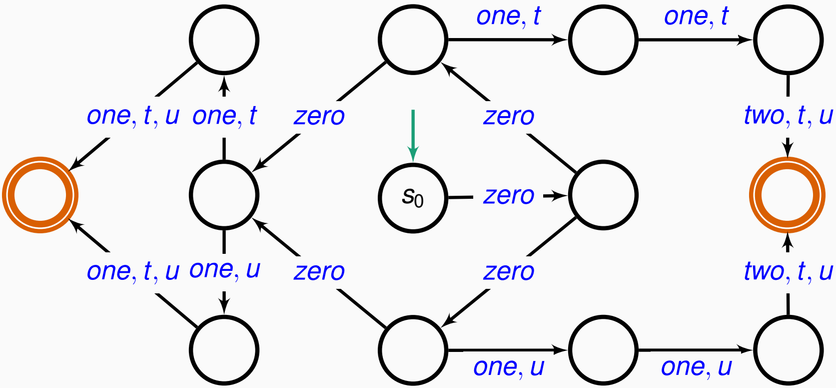
- *zero*, *one*, *two*: corresponding to the value of the counter
- t, u: process t, u has terminated



Kripke structure to finite-state automaton: example

We keep track of predicates:

- *zero*, *one*, *two*: corresponding to the value of the counter
- t, u: process t, u has terminated



Model checking concurrent behavior

Now we can express analysis questions as instances of the model-checking problem.

Is `counter` always 2 after processes t and u terminate?

Model checking concurrent behavior

Now we can express analysis questions as instances of the model-checking problem.

Is `counter` always 2 after processes t and u terminate?

$$Count \stackrel{?}{\models} Two$$

Count: the automaton built from the Kripke structure modeling the concurrent program

Two: the LTL formula $\Box(t \wedge u \Longrightarrow two)$

Model checking concurrent behavior

Now we can express analysis questions as instances of the model-checking problem.

Is `counter` always 2 after processes t and u terminate?

$$Count \overset{?}{\models} Two$$

Count: the automaton built from the Kripke structure modeling the concurrent program

Two: the LTL formula $\square(t \wedge u \Longrightarrow two)$

Now LTL formulas are over $\Pi = power(P)$; correspondingly, we interpret the satisfaction of a proposition at one time step as:

$$w, k \models p \qquad \text{iff} \qquad p \in w[k]$$

since each word symbol is a set of propositions in P.

Automata-based model checking

Model checking tools

Model checking with Spin

Model checking tools provide specialized languages to conveniently and concisely express transition systems conveniently as the composition of parallel processes.

Spin is widely used explicit-state model checking: it implements the automata-theoretic model checking algorithm we have seen, in the version on the fly and with many optimizations to support scalability.

Model checking with Spin

Model checking tools provide specialized languages to conveniently and concisely express transition systems conveniently as the composition of parallel processes.

Spin is widely used explicit-state model checking: it implements the automata-theoretic model checking algorithm we have seen, in the version on the fly and with many optimizations to support scalability.

Spin inputs transition system models expressed in the ProMeLa (Process Meta Language) language, which describes transition systems and LTL properties.

Shared counters in ProMeLa

Here is the shared counter increment example encoded as a transition system in ProMeLa.

```
// shared memory
int count = 0;

// boolean array
// to keep track of termination
bit done[3] = 0;

proctype inc_process() {
  int tmp;          // process-local variable
  tmp = count;
  count = tmp + 1;
  done[_pid] = 1;   // set termination bit
  // '_pid' gives the process id of the running process
}

// spawn two processes running the same code
init { run inc_process(); run inc_process(); }
// the spawned processes will have _pid == 1 and 2
// process 0 is the main entry point of the system
```

LTL properties

We can embed the LTL properties we want to verify in a ProMeLa file.

We can encode predicates directly with C-style Boolean expression, or use pre-processor macros to define named predicates.

```
#define t (done[1] == 1)
#define u (done[2] == 1)

ltl Two {
  [] (t && u -> count == 2)
}

ltl OneOrTwo {
  [] (t && u -> (count == 1 || count == 2))
}
```

Running Spin on the shared counter example

```
> spin -a unitCounter.pml      # create analyzer pan.c
> gcc -o mc_unitCounter pan.c  # compile analyzer
> ./mc_unitCounter -a -N Two   # run analyzer with property `Two'

  pan:1: assertion violated
  pan: wrote unitCounter.pml.trail
# this means: not (unitCounter |= Two)
> spin -k unitCounter.pml.trail unitCounter.pml # replay error trace

  Process 2 reads 0
  Process 1 reads 0
  Process 2 writes 1
  Process 2 terminates
  Process 1 writes 1
  Process 1 terminates
```

Print statements

To make error traces easier to follow, we can embed **printf** statement that will be used when we replay an error trace.

To ensure a **printf** statement is printed when a process takes a certain step, we use **atomic** blocks, which are guaranteed to be not interruptible.

```
proctype inc_process() {
  int tmp;
  atomic {
   tmp = count;
   printf("Process␣%d␣reads␣%d\n", _pid, tmp);
  }
  atomic{
   count = tmp + 1;
   printf("Process␣%d␣writes␣%d\n", _pid, tmp + 1);
  }
  atomic {
   done[_pid] = 1;
   printf("Process␣%d␣terminates\n", _pid);
  }
}
```

LTL properties on the command line

We can also provide LTL properties directly on the command line of Spin. In this case, the given formula is directly intersected with the automaton; thus we have to pass the negated property $\neg P$ to verify P

```
> spin -a -f '![](t && u -> (count >= 1))' unitCounter.pml
> gcc -o mc_unitCounter pan.c
> ./mc_unitCounter -a     # verifies command-line property by default

  [no errors]
# this means: unitCounter |= [](t && u -> (count >= 1))
```

LTL properties on the command line

We can also provide LTL properties directly on the command line of Spin. In this case, the given formula is directly intersected with the automaton; thus we have to pass the negated property $\neg P$ to verify P

```
> spin -a -f '![](t && u -> (count >= 1))' unitCounter.pml
> gcc -o mc_unitCounter pan.c
> ./mc_unitCounter -a     # verifies command-line property by default

  [no errors]
# this means: unitCounter |= [](t && u -> (count >= 1))
```

If you want a more efficient encoding of LTL properties, you can use a specialized tool that outputs ProMeLa code for $\mathcal{A}(P)$:

Oddoux and Gastin's `ltl2tgba`
Web interface to LTL2BA

A more interesting example

Here is a more challenging example, which is not easy to analyze without tools.

If each process increments the shared `counter` 10 times, what is the minimum value of `count` after a complete run?

```
var counter: Integer // initially 0
```

process t

```
  var cnt, k: Integer
1 k := 0
2 while (k < 10)
3   cnt := counter
4   counter := cnt + 1
5   k := k + 1
```

process u

```
var cnt, k: Integer
k := 0                    3
while (k < 10)            4
  cnt := counter          5
  counter := cnt + 1      6
  k := k + 1              7
```

Running Spin on the multiple increment example

```
# compile one version of the analyzer for each bound k = 10, 9, ..., 2
> for k in $(seq 10 -1 2); do \
      spin -a -f '!'"[](t && u -> (count >= $k))" multiCounter.pml; \
      gcc -o "bound_$k" pan.c; done
> ./bound_10 -a | grep 'assertion violated'
# ...
> ./bound 2 -a | grep 'assertion violated'
> ./bound 1 -a | grep 'assertion violated'
```

Model checking concurrent Java

Let us analyze using spin a more complex example of concurrent code using Java threads.

The following analysis is based on Hillel Wayne's, who analyzed, using the formal language TLA^{+} and model checking, an extreme programming challenge posed by Tom Cargill.

```java
class BoundedBuffer {
  synchronized void put(Object x)
          throws InterruptedException {
    while( occupied == buffer.length )
      wait();
    notify();
    ++occupied;
    putAt %= buffer.length;
    buffer[putAt++] = x;
  }

  synchronized Object take()
             throws InterruptedException {
    while( occupied == 0 )
      wait();
    notify();
    --occupied;
    takeAt %= buffer.length;
    return buffer[takeAt++];
  }

  private Object[] buffer = new Object[4];
  private int putAt=0, takeAt=0, occupied=0;
}
```

```java
class BoundedBuffer {
  synchronized void put(Object x)
          throws InterruptedException {
    while( occupied == buffer.length )
      wait();
    notify();
    ++occupied;
    putAt %= buffer.length;
    buffer[putAt++] = x;
  }

  synchronized Object take()
             throws InterruptedException {
    while( occupied == 0 )
      wait();
    notify();
    --occupied;
    takeAt %= buffer.length;
    return buffer[takeAt++];
  }

  private Object[] buffer = new Object[4];
  private int putAt=0, takeAt=0, occupied=0;
}
```

block running thread
(waiting for condition)

unblock any one blocked thread
(nondeterministically chosen)

monitor: threads run on shared object
in mutual exclusion

```java
class BoundedBuffer {
  synchronized void put(Object x)
          throws InterruptedException {
    while( occupied == buffer.length )
      wait();
    notify();
    ++occupied;
    putAt %= buffer.length;
    buffer[putAt++] = x;
  }

  synchronized Object take()
             throws InterruptedException {
    while( occupied == 0 )
      wait();
    notify();
    --occupied;
    takeAt %= buffer.length;
    return buffer[takeAt++];
  }

  private Object[] buffer = new Object[4];
  private int putAt=0, takeAt=0, occupied=0;
}
```

Perhaps someone can show me how to modify [this] code to make it more testable, and then how to write a test that exposes its bug.

Tom Cargill: Extreme Programming Challenge 14

```java
class BoundedBuffer {
  synchronized void put(Object x)
          throws InterruptedException {
    while( occupied == buffer.length )
      wait();
    notify();
    ++occupied;
    putAt %= buffer.length;
    buffer[putAt++] = x;
  }

  synchronized Object take()
             throws InterruptedException {
    while( occupied == 0 )
      wait();
    notify();
    --occupied;
    takeAt %= buffer.length;
    return buffer[takeAt++];
  }

  private Object[] buffer = new Object[4];
  private int putAt=0, takeAt=0, occupied=0;
}
```

Now that the unit tests have been written I see that there are no bugs. Sometimes unit tests surprise you by telling you that your code actually does work.

Don Wells

A ProMeLa model of Java monitors

```
// number of producer processes
#define NPROD 1
// number of consumer processes
#define NCONS 2
```

A ProMeLa model of Java monitors

```
// number of producer processes
#define NPROD 1
// number of consumer processes
#define NCONS 2

int lock = 0;       // pid of process which has the monitor's lock

bool blocked[P];    // set of blocked processes  (blocked[pid] iff pid is blocked)
int  nblocked = 0;  // number of blocked processes

int contains = 0;   // number of items in buffer
```

A ProMeLa model of Java monitors

```
// number of producer processes
#define NPROD 1
// number of consumer processes
#define NCONS 2

int lock = 0;       // pid of process which has the monitor's lock

bool blocked[P];    // set of blocked processes  (blocked[pid] iff pid is blocked)
int  nblocked = 0;  // number of blocked processes

int contains = 0;   // number of items in buffer

proctype scheduler()
{
  int p;
  do
   // wait until no process is active on the monitor
   :: lock == 0 ->
     printf("Scheduler␣running\n");
     assert (nblocked < P);   // all blocked means deadlock!
     // nondeterministically select an unblocked process

  ...
```

Spin verification of Java monitors

```
pan:1: assertion violated (nblocked<(1+2)) (at depth 339)
```

Simplified error trace (Spin processes correspond to Java threads):

1. Initially all processes P (producer) and C_1, C_2 (consumers) are unblocked and the buffer (capacity 1) is empty
2. C_1 finds empty buffer, blocks
3. C_2 finds empty buffer, blocks
4. P writes to buffer, wakes up C_2
5. P finds full buffer, blocks
6. C_2 reads from buffer, wakes up C_1
7. C_1 finds empty buffer, blocks
8. C_2 finds empty buffer, blocks
9. All processes are blocked!

Detailed analysis using model checking

By modifying the model and running model checking again, we can evaluate different design choices and reason about properties that are nearly impossible to get right by testing and debugging.

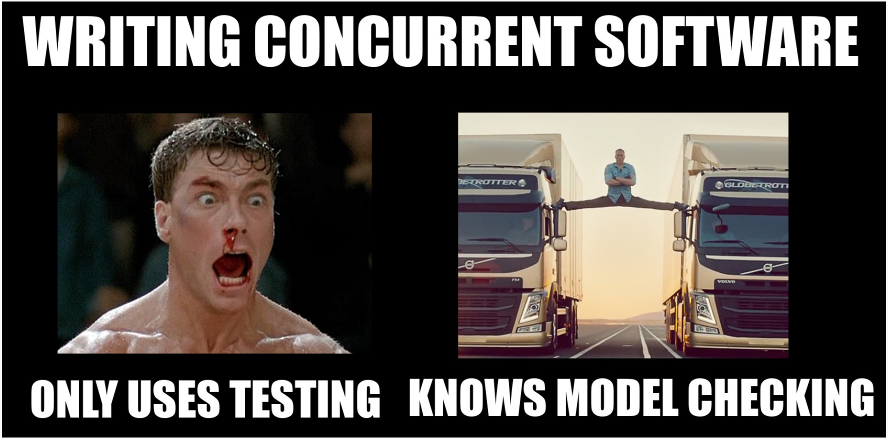
- If we remove the violated assertion, we still get an invalid end state error, which means exactly that the system can deadlock.
- A deadlock seems to occur only when the number of process (producers and consumers) is more than twice the buffer's size.
- To avoid deadlocks it is enough to require add an entry queue to the ProMeLa model that processes have to go through every time they enter the monitor.

Detailed analysis using model checking

By modifying the model and running model checking again, we can evaluate different design choices and reason about properties that are nearly impossible to get right by testing and debugging.

- If we remove the violated assertion, we still get an invalid end state error, which means exactly that the system can deadlock.
- A deadlock seems to occur only when the number of process (producers and consumers) is more than twice the buffer's size.
- To avoid deadlocks it is enough to require add an entry queue to the ProMeLa model that processes have to go through every time they enter the monitor.

Indeed, more recent versions of Java also offer a different, library-based implementation of monitors, which has nicer fairness properties (such as that `notify` wakes up the thread that has been waiting the longest).


WRITING CONCURRENT SOFTWARE
ONLY USES TESTING
KNOWS MODEL CHECKING

Automata-based model checking

Variants of model checking

Flexible automata-based model checking

finite-state model S

temporal logic property P

model checking algorithm A:

MONITOR: $\mathcal{A}(\neg P)$

INTERSECTION: $S \times \mathcal{A}(\neg P)$

EMPTINESS: $\mathcal{L}(S \times \mathcal{A}(\neg P)) = \emptyset$

✔ correct

✖ incorrect

The framework of automata-based model checking is quite flexible, as it can accommodate several variants in:

- the kind of automaton model used to describe the system
- the flavor of temporal logic used to express properties
- the techniques used to implement the checking algorithm

Let's outline some interesting variants in each category.

Variants of the model checking algorithm

There are two main families of algorithms to implement the monitor construction, intersection, and emptiness steps of model checking:

explicit-state algorithms perform the steps as algorithms on graphs, very similarly to how we have presented them (with practical optimizations such as applying them on the fly)

symbolic algorithms encode system model S and property P as symbolic constraints, and then rely on constraint solvers to perform the analysis

Variants of the model checking algorithm

There are two main families of algorithms to implement the monitor construction, intersection, and emptiness steps of model checking:

explicit-state algorithms perform the steps as algorithms on graphs, very similarly to how we have presented them (with practical optimizations such as applying them on the fly)

symbolic algorithms encode system model S and property P as symbolic constraints, and then rely on constraint solvers to perform the analysis

The first symbolic model checking algorithms represented constraints using binary decision diagrams (BDDs), a directed acyclic data structure that can concisely represent Boolean functions and operations such as conjunction and emptiness check.

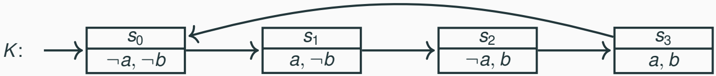
Taking advantage of the spectacular advances in SAT-solving technology, bounded model checking is a different kind of symbolic algorithm that uses propositional logic to encode the model checking problem.

Bounded model checking

Bounded model checking encodes automata and temporal logic as propositional formulas, where proposition represents the state a computation is in at each step in a run.

Bounded model checking

Bounded model checking encodes automata and temporal logic as propositional formulas, where proposition represents the state a computation is in at each step in a run.



We represent a run $r[0]\, r[1] \, \dots \, r[n]$ using $2 \cdot (n+1)$ propositions $a[k], b[k]$ for $k = 0, \dots, n$, where $a[k]$ denotes that $a \in r[k]$ holds at k.

The runs of length up to n are encoded as:

$$runs(K, n) = init(0) \wedge N(n) \wedge \bigwedge_{0 \leq k < n} \big(\neg N(k) \Longrightarrow T(k, k+1)\big)$$

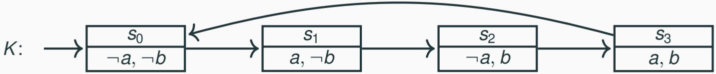
$init(k)$ is the indicator function of the set of initial states

$N(k)$ denotes that the run has length up to k

$T(x, y)$ is the indicator function of the transition relation

Bounded model checking

Bounded model checking encodes automata and temporal logic as propositional formulas, where proposition represents the state a computation is in at each step in a run.



We represent a run $r[0]\,r[1]\,\ldots\,r[n]$ using $2 \cdot (n+1)$ propositions $a[k], b[k]$ for $k = 0, \ldots, n$, where $a[k]$ denotes that $a \in r[k]$ holds at k.

The runs of length up to n are encoded as:

$$runs(K, n) = init(0) \wedge N(n) \wedge \bigwedge_{0 \leq k < n} \big(\neg N(k) \implies T(k, k+1)\big)$$

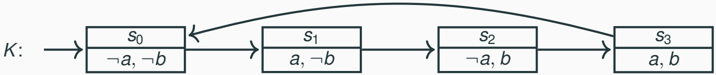
step k is taken unless the run has ended before k steps

$init(k)$ is the indicator function of the set of initial states

$N(k)$ denotes that the run has length up to k

$T(x, y)$ is the indicator function of the transition relation

Encoding runs as propositional formulas: example



Let's encode all runs of length up to bound 2.

a transition is taken unless the run has ended before

$$runs(K, 2) = init(0) \wedge N(2) \wedge (\neg N(0) \implies T(0,1)) \wedge (\neg N(1) \implies T(1,2))$$

$$init(0) = \neg a[0] \wedge \neg b[0]$$

$$T(0,1) = \big(a[1] \iff \neg a[0]\big) \wedge \big(b[1] \iff (a[0] \oplus b[0])\big)$$

$$T(1,2) = \big(a[2] \iff \neg a[1]\big) \wedge \big(b[2] \iff (a[1] \oplus b[1])\big)$$

a switches value in every transition

b becomes true after *a* XOR *b*

$$N(k) = \bigvee_{0 \le t \le k} length(t)$$

the run's length is encoded in binary using ℓ_1, ℓ_0

$$length(0) = \neg \ell_1 \wedge \neg \ell_0 \qquad length(1) = \neg \ell_1 \wedge \ell_0 \qquad length(2) = \ell_1 \wedge \neg \ell_0$$

Encoding monitors as propositional formulas

Using a similar formalization, we can also express monitors of LTL formulas as propositional formulas.

Encoding monitors as propositional formulas

Using a similar formalization, we can also express monitors of LTL formulas as propositional formulas.

For example, for property $P = \Diamond(a \land b)$:

$$runs(P, n) = \big(a[0] \land b[0]\big) \lor \bigvee_{0 \leq t < n} \big(\neg N(t) \land a[t+1] \land b[t+1]\big)$$

Bounded model checking

In more general cases, the encoding has to be a bit more complex because $runs(K, n)$ should encode runs of length up to n or with loops of length up to n.

In our example, however, there are no loops of length up to 2.

Bounded model checking

In more general cases, the encoding has to be a bit more complex because $runs(K, n)$ should encode runs of length up to n or with loops of length up to n.

In our example, however, there are no loops of length up to 2.

Once we have defined the encoding, the model checking algorithm is straightforward to implement:

monitor constructs $runs(\neg P, n) = \neg runs(P, n)$
complementing the property is negation

intersection is conjunction: $runs(K, n) \land \neg runs(P, n)$

emptiness is satisfiability: $K \models P$ iff $runs(K, n) \land \neg runs(P, n)$ is unsatisfiable

Bounded model checking: completeness

Using a finite bound on run length does not affect completeness of bounded model checking.

Every finite-state model K has a computable diameter D_K:
all runs of K have length up to D_K or a loop of length up to D_K.

Bounded model checking: completeness

Using a finite bound on run length does not affect completeness of bounded model checking.

Every finite-state model K has a computable diameter D_K:
all runs of K have length up to D_K or a loop of length up to D_K.

Therefore, bounded model checking works as follows:

```
k = 0
while k <= D_K:
    if SAT(runs(K, k) ∧ ¬runs(P, k)):
        return (K ⊭ P, counterexample)
    else:
        k = k + 1
return K ⊨ P  # no counterexample beyond diameter
```

This algorithm has the additional advantage that it always returns counterexamples of minimal length.

Different kinds of automata

Among the kinds of finite-state models used for model checking, we have seen Kripke structures and their correspondence to finite-state automata.

We will also see timed automata as an example of infinite-state model that can still be analyzed using the model checking approach since its emptiness problem is still decidable.

Different kinds of automata

Among the kinds of finite-state models used for model checking, we have seen Kripke structures and their correspondence to finite-state automata.

We will also see timed automata as an example of infinite-state model that can still be analyzed using the model checking approach since its emptiness problem is still decidable.

A popular variant of finite-state automata are Büchi automata: finite-state automata that run over infinite words.

Infinite words are useful abstraction when termination is not an interesting event (such as in reactive systems) or is a potentially error (such as in deadlocks).

Since these are classic applications, most presentation of model checking use Büchi automata instead of finite-state automata.

Büchi automata: semantics

Büchi automata have the same syntax $B = \langle \Sigma, S, I, F, \rho \rangle$ as finite-state automata but are interpreted over infinite words:

An infinite word is an input sequence of unbounded length:

$$w = w[1]\, w[2]\, \ldots\, w[n] \in \Sigma^\omega$$

infinite sequences of elements in Σ

Büchi automata: semantics

Büchi automata have the same syntax $B = \langle \Sigma, S, I, F, \rho \rangle$ as finite-state automata but are interpreted over infinite words:

An infinite word is an input sequence of unbounded length:

$$w = w[1]\, w[2]\, \ldots\, w[n] \in \Sigma^\omega$$

infinite sequences of elements in Σ

A run of B over w is a sequence of states

$$r = r[0]\, r[1]\, \ldots\, \in S^\omega$$

that starts from an initial state and follows B's transitions.

Büchi automata: semantics

Büchi automata have the same syntax $B = \langle \Sigma, S, I, F, \rho \rangle$ as finite-state automata but are interpreted over infinite words:

An infinite word is an input sequence of unbounded length:

$$w = w[1]\, w[2]\, \ldots\, w[n] \in \Sigma^\omega$$

infinite sequences of elements in Σ

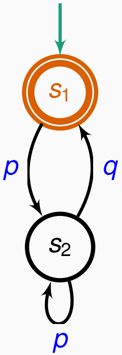
A run of B over w is a sequence of states

$$r = r[0]\, r[1]\, \ldots \in S^\omega$$

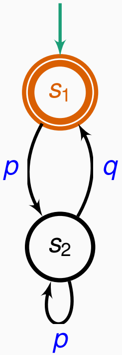
that starts from an initial state and follows B's transitions.

A run r of B is accepting if it traverses infinitely often a final state: $r[k] \in F$ for infinitely many value of k.

Büchi runs: example

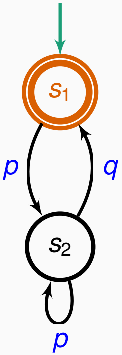


Büchi runs: example



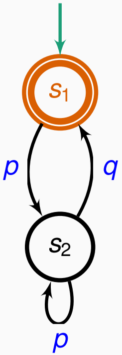
Run $r = s_1\ s_2\ s_2\ s_1\ s_2\ s_2\ s_1\ \cdots$

Büchi runs: example



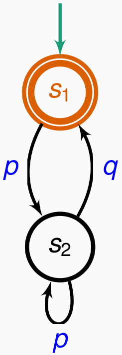
Run $r = s_1\, s_2\, s_2\, s_1\, s_2\, s_2\, s_1\ \cdots$
over $w = p\, p\, q\, p\, p\, q \cdots$

Büchi runs: example



Run $r = s_1\, s_2\, s_2\, s_1\, s_2\, s_2\, s_1\, \cdots$
over $w = p\, p\, q\, p\, p\, q \cdots$
is accepting.

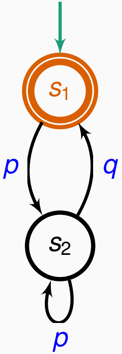
Büchi runs: example



Run $r = s_1\, s_2\, s_2\, s_1\, s_2\, s_2\, s_1\, \cdots$
over $w = p\, p\, q\, p\, p\, q \cdots$
is accepting.

Run $r = s_1\, s_2\, s_2\, s_s\, s_2\, \cdots$

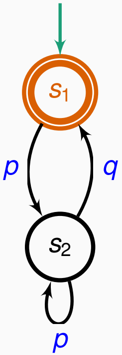
Büchi runs: example



Run $r = s_1\ s_2\ s_2\ s_1\ s_2\ s_2\ s_1\ \cdots$
over $w = p\,p\,q\,p\,p\,q \cdots$
is accepting.

Run $r = s_1\ s_2\ s_2\ s_s\ s_2\ \cdots$
over $w = p\,p\,p\,p\ \cdots$

Büchi runs: example



Run $r = s_1\, s_2\, s_2\, s_1\, s_2\, s_2\, s_1\, \cdots$
over $w = p\,p\,q\,p\,p\,q \cdots$
is accepting.

Run $r = s_1\, s_2\, s_2\, s_s\, s_2\, \cdots$
over $w = p\,p\,p\,p\, \cdots$
is not accepting.

Büchi vs. finite state automata

There are a few technical differences in the kinds of properties Büchi rather than finite-state automata have:

FINITE-STATE AUTOMATA	BÜCHI AUTOMATA
closed under determinization	nondeterministic more expressive than deterministic
complementing is easy through determinization	complementing is hard (cannot always determinize)
intersection is easy	intersection is a bit trickier because of the acceptance condition
emptiness is reachability	emptiness is loop detection (reachability of a state from itself)

Büchi vs. finite state automata

There are a few technical differences in the kinds of properties Büchi rather than finite-state automata have:

FINITE-STATE AUTOMATA	BÜCHI AUTOMATA
closed under determinization	nondeterministic more expressive than deterministic
complementing is easy through determinization	complementing is hard (cannot always determinize)
intersection is easy	intersection is a bit trickier because of the acceptance condition
emptiness is reachability	emptiness is loop detection (reachability of a state from itself)

Kripke structures can also be interpreted over infinite words. Then, all states of a Büchi automaton representing the infinite-length runs of a Kripke structure are accepting to ensure progress.

Model checking Büchi automata

The automata-based model checking algorithm needs a few adjustments to work with Büchi automata.

MONITORS follow the semantics of LTL over infinite words, which is actually simpler because there is no word length to worry about. It is easier to complement the property rather than the monitor, due to the difficulties of complementing Büchi automata.

INTERSECTION uses a product construction that keeps track of the acceptance condition of A and of B simultaneously.

EMPTINESS: B is not empty iff there exists a path to a cycle containing an accepting state. This can be checked efficiently by finding the maximal strongly connected components of B and checking whether they contain an accepting state.

A strongly connected component is a subgraph of a directed graph where every node is reachable from any other node.

Model checking Büchi automata

The automata-based model checking algorithm needs a few adjustments to work with Büchi automata.

Despite the technical differences, the complexity of model checking Büchi automata is essentially the same as the complexity of finite-state automata:

Model checking LTL properties
of Büchi automata is PSPACE-complete.

Algorithms have complexity linear in the size of the automaton and exponential in the size of the property.

Different kinds of temporal logic

LTL (linear-time temporal logic) is the most intuitive variant of temporal logic.

We will see an LTL extension called MTL, which adds quantitative time constraints.

The other main kind of temporal logic is branching-time temporal logic. Let's have a look at CTL (Computation Tree Logic) – a popular version of branching-time logic.

CTL: syntax

Formulas of propositional CTL are defined as:

$$
\begin{array}{rll}
F ::= & p \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \implies F_2 & \text{(propositional connectives)} \\
\mid & \exists \mathrm{X}\, F \mid \exists \Box\, F \mid \exists \Diamond\, F \mid F_1\, \exists \mathrm{U}\, F_2 & \text{(existential connectives)} \\
\mid & \forall \mathrm{X}\, F \mid \forall \Box\, F \mid \forall \Diamond\, F \mid F_1\, \forall \mathrm{U}\, F_2 & \text{(universal connectives)}
\end{array}
$$

CTL: syntax

Formulas of propositional CTL are defined as:

$$
\begin{aligned}
F ::=\ & p \mid \neg F \mid F_1 \land F_2 \mid F_1 \lor F_2 \mid F_1 \Longrightarrow F_2 && \text{(propositional connectives)}\\
\mid\ & \exists \mathrm{X}\, F \mid \exists\Box\, F \mid \exists\Diamond\, F \mid F_1\ \exists\mathrm{U}\ F_2 && \text{(existential connectives)}\\
\mid\ & \forall \mathrm{X}\, F \mid \forall\Box\, F \mid \forall\Diamond\, F \mid F_1\ \forall\mathrm{U}\ F_2 && \text{(universal connectives)}
\end{aligned}
$$

Each temporal connective has two variants:

existential: its arguments holds on some run from the current step

universal: its arguments holds on all runs from the current step

For this reason, $\forall$ and $\exists$ in branching-time formulas are called path quantifiers, since they quantify over paths (that is, runs).

CTL: satisfaction relation

CTL formulas are interpreted directly on some form of transition system – typically a Kripke structure $K = \langle S, I, R, P, L \rangle$ in the infinite-word interpretation.

CTL: satisfaction relation

CTL formulas are interpreted directly on some form of transition system – typically a Kripke structure $K = \langle S, I, R, P, L \rangle$ in the infinite-word interpretation.

A structure K generates runs that satisfy CTL formula F at state s, written $K, s \models F$, iff:

$$
\begin{array}{rcl}
K, s \models p & \text{iff} & p \in L(s) \\
K, s \models \neg F & \text{iff} & K, s \not\models F \\
K, s \models F_1 \wedge F_2 & \text{iff} & K, s \models F_1 \text{ and } K, s \models F_2 \\
K, s \models F_1 \vee F_2 & \text{iff} & K, s \models F_1 \text{ or } K, s \models F_2
\end{array}
$$

CTL: satisfaction relation

CTL formulas are interpreted directly on some form of transition system – typically a Kripke structure $K = \langle S, I, R, P, L \rangle$ in the infinite-word interpretation.

A structure K generates runs that satisfy CTL formula F at state s, written $K, s \models F$, iff:

$K, s \models \exists \mathrm{X}\, F$	iff	for some s_1 : $s\, R\, s_1$ and $K, s_1 \models F$
$K, s \models \forall \mathrm{X}\, F$	iff	for all s_1 such that $s\, R\, s_1$: $K, s_1 \models F$

Intuitively:

$K, s \models \exists \mathrm{X}\, F$	a state where F holds is reachable from s in one step
$K, s \models \forall \mathrm{X}\, F$	F holds in all states reachable from s in one step

CTL: satisfaction relation

CTL formulas are interpreted directly on some form of transition system – typically a Kripke structure $K = \langle S, I, R, P, L \rangle$ in the infinite-word interpretation.

A structure K generates runs that satisfy CTL formula F at state s, written $K, s \models F$, iff:

$K, s \models \exists\Box\, F$ iff for some run $r = s\, s_1\, s_2\, s_3\, \ldots$ following R:
for all states $s_k \in r$: $K, s_k \models F$

$K, s \models \forall\Box\, F$ iff for all runs $r = s\, s_1\, s_2\, s_3\, \ldots$ following R:
for all states $s_k \in r$: $K, s_k \models F$

Intuitively:

$K, s \models \exists\Box\, F$ there is a run from s where F holds always
$K, s \models \forall\Box\, F$ F holds always in all runs from s

CTL: satisfaction relation

CTL formulas are interpreted directly on some form of transition system – typically a Kripke structure $K = \langle S, I, R, P, L \rangle$ in the infinite-word interpretation.

A structure K generates runs that satisfy CTL formula F at state s, written $K, s \models F$, iff:

$K, s \models \exists\Diamond F$ iff for some run $r = s\, s_1\, s_2\, s_3\, \ldots$ following R: for some state $s_k \in r$: $K, s_k \models F$

$K, s \models \forall\Diamond F$ iff for all runs $r = s\, s_1\, s_2\, s_3\, \ldots$ following R: for some state $s_k \in r$: $K, s_k \models F$

Intuitively:

$K, s \models \exists\Diamond F$	there is a run from s where F holds eventually
$K, s \models \forall\Diamond F$	F holds eventually in all runs from s

CTL: satisfaction relation

CTL formulas are interpreted directly on some form of transition system – typically a Kripke structure $K = \langle S, I, R, P, L \rangle$ in the infinite-word interpretation.

A structure K generates runs that satisfy CTL formula F at state s, written $K, s \models F$, iff:

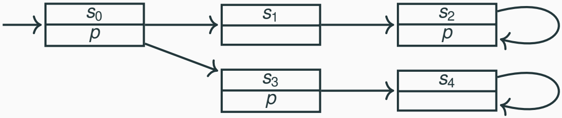
$K, s \models F_1 \ \exists\mathrm{U}\ F_2$ iff for some run $r = s\, s_1\, s_2\, s_3 \, \ldots$ following R:
for some state $s_k \in r$: $K, s_k \models F$, and,
for all $s_h \in s\, s_1\, s_{k-1}$: $K, s_h \models F_1$

$K, s \models F_1 \ \forall\mathrm{U}\ F_2$ iff for all runs $r = s\, s_1\, s_2\, s_3 \, \ldots$ following R:
for some state $s_k \in r$: $K, s_k \models F$, and,
for all $s_h \in s\, s_1\, s_{k-1}$: $K, s_h \models F_1$

Intuitively:

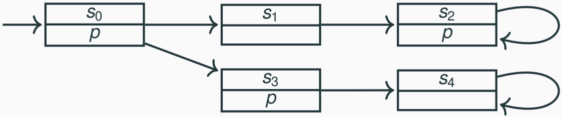
$K, s \models F_1 \ \exists\mathrm{U}\ F_2$ there is a run from s where $F_1 \ \mathrm{U}\ F_2$ holds at s
$K, s \models F_1 \ \forall\mathrm{U}\ F_2$ $F_1 \ \mathrm{U}\ F_2$ holds at s in all runs from s

CTL semantics: examples



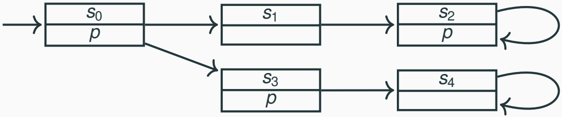
$$K, s_0 \quad \models \quad \exists \mathrm{X}(\neg p)$$

CTL semantics: examples



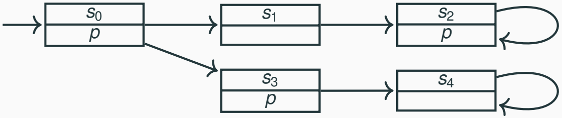
$$
\begin{array}{lcll}
K, s_0 & \models & \exists \mathrm{X}(\neg p) & ✔ \\
K, s_0 & \models & \forall \mathrm{X}(\neg p) &
\end{array}
$$

CTL semantics: examples



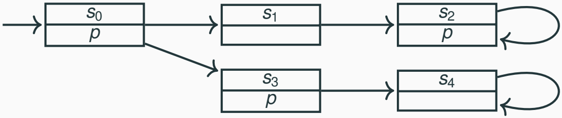
$$\begin{array}{llll} K, s_0 & \models & \exists X(\neg p) & ✔ \\ K, s_0 & \models & \forall X(\neg p) & ✖ \\ K, s_2 & \models & \forall X\, p & \end{array}$$

CTL semantics: examples



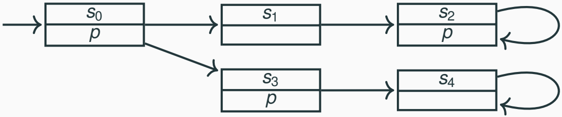
K, s_0	$\models$	$\exists \mathrm{X}(\neg p)$	✓
K, s_0	$\models$	$\forall \mathrm{X}(\neg p)$	✗
K, s_2	$\models$	$\forall \mathrm{X}\, p$	✓
K, s_0	$\models$	$\exists \Diamond(\neg p)$	

CTL semantics: examples



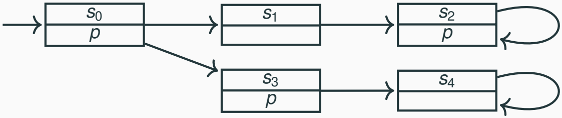
K, s_0	$\models$	$\exists \mathrm{X}(\neg p)$	✔
K, s_0	$\models$	$\forall \mathrm{X}(\neg p)$	✖
K, s_2	$\models$	$\forall \mathrm{X}\, p$	✔
K, s_0	$\models$	$\exists \Diamond(\neg p)$	✔
K, s_0	$\models$	$\forall \Diamond(\neg p)$	

CTL semantics: examples



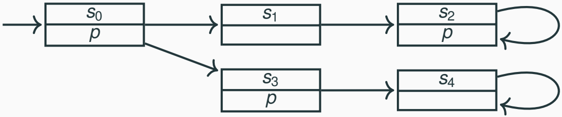
$$
\begin{array}{llll}
K, s_0 & \models & \exists \mathrm{X}(\neg p) & ✔ \\
K, s_0 & \models & \forall \mathrm{X}(\neg p) & ✖ \\
K, s_2 & \models & \forall \mathrm{X}\, p & ✔ \\
K, s_0 & \models & \exists \Diamond(\neg p) & ✔ \\
K, s_0 & \models & \forall \Diamond(\neg p) & ✔ \\
K, s_0 & \models & \forall \Diamond(p \wedge \forall \mathrm{X}\, p) &
\end{array}
$$

CTL semantics: examples



$$
\begin{array}{lcll}
K, s_0 & \models & \exists \mathsf{X}(\neg p) & ✔ \\
K, s_0 & \models & \forall \mathsf{X}(\neg p) & ✖ \\
K, s_2 & \models & \forall \mathsf{X}\, p & ✔ \\
K, s_0 & \models & \exists \Diamond(\neg p) & ✔ \\
K, s_0 & \models & \forall \Diamond(\neg p) & ✔ \\
K, s_0 & \models & \forall \Diamond(p \wedge \forall \mathsf{X}\, p) & ✖ \\
K, s_0 & \models & \Diamond(p \wedge \mathsf{X}\, p) &
\end{array}
$$

CTL semantics: examples



K, s_0	$\models$	$\exists X(\neg p)$	✓
K, s_0	$\models$	$\forall X(\neg p)$	✗
K, s_2	$\models$	$\forall X\, p$	✓
K, s_0	$\models$	$\exists \Diamond(\neg p)$	✓
K, s_0	$\models$	$\forall \Diamond(\neg p)$	✓
K, s_0	$\models$	$\forall \Diamond(p \land \forall X\, p)$	✗
K, s_0	$\models$	$\Diamond(p \land X\, p)$	✓

LTL property

LTL vs. CTL

There has been a long-standing debate about whether linear-time or branching-time logic is "better":

intuitiveness: LTL formulas are generally easier to understand

expressiveness: LTL and CTL have incomparable expressive power

- $\forall\Diamond(p \wedge \forall \mathsf{X}\, p)$ is inexpressible in LTL
- $\Diamond(p \wedge \mathsf{X}\, p)$ is inexpressible in CTL

complexity: CTL model checking is P-complete
LTL model checking is PSPACE-complete

algorithms: CTL model checking can be solved in $\mathrm{O}(|A| \cdot |F|)$
LTL model checking can be solved in $\mathrm{O}(|A| \cdot 2^{|F|})$

LTL vs. CTL

There has been a long-standing debate about whether linear-time or branching-time logic is "better":

intuitiveness: LTL formulas are generally easier to understand

expressiveness: LTL and CTL have incomparable expressive power

- $\forall\Diamond(p \wedge \forall \mathsf{X}\, p)$ is inexpressible in LTL
- $\Diamond(p \wedge \mathsf{X}\, p)$ is inexpressible in CTL

complexity: CTL model checking is P-complete
LTL model checking is PSPACE-complete

algorithms: CTL model checking can be solved in $\mathrm{O}(|A| \cdot |F|)$
LTL model checking can be solved in $\mathrm{O}(|A| \cdot 2^{|F|})$

The complexity gap is not really an issue in practice:

- Formulas F are normally small
- CTL and LTL model checking tends to have similar performance for formulas that are expressible in both logics
- CTL's performance advantage vanishes when model checking open systems

Linear temporal logic with past operators

LTL with past (PLTL) adds temporal connectives that refer to the past:

$$F ::= \; \mathsf{Y}\, F \mid \overleftarrow{\Box}\, F \mid \overleftarrow{\Diamond}\, F \mid F_1 \,\mathsf{S}\, F_2 \qquad \text{(past temporal connectives)}$$

Linear temporal logic with past operators

yesterday/in the previous step

always in the past

LTL with past (PLTL) adds temporal connectives that refer to the past:

$$F ::= \mathsf{Y}\, F \mid \overleftarrow{\Box}\, F \mid \overleftarrow{\Diamond}\, F \mid F_1\, \mathsf{S}\, F_2 \qquad \text{(past temporal connectives)}$$

eventually/sometimes in the past

since

PLTL: complexity and expressiveness

Some properties are easier to express using past operators.

For example: “every alarm is due to a fault”

LTL	PLTL
$\neg\Big(\neg \mathit{fault}\ \mathsf{U}\ (\mathit{alarm} \wedge \neg \mathit{fault})\Big)$	$\square\,(\mathit{alarm} \Longrightarrow \overleftarrow{\Diamond}\, \mathit{fault})$

PLTL: complexity and expressiveness

Some properties are easier to express using past operators.

For example: "every alarm is due to a fault"

LTL	PLTL
$\neg\Big(\neg fault \; \mathsf{U} \; (alarm \wedge \neg fault)\Big)$	$\Box \, (alarm \Longrightarrow \overleftarrow{\Diamond} \, fault)$

However, PLTL has the same expressiveness as LTL: every PLTL formula has an equivalent LTL formula

PLTL: complexity and expressiveness

Some properties are easier to express using past operators.

For example: "every alarm is due to a fault"

LTL	PLTL
$\neg\Big(\neg \mathit{fault}\ \mathsf{U}\ (\mathit{alarm} \land \neg \mathit{fault})\Big)$	$\Box\,(\mathit{alarm} \Longrightarrow \overleftarrow{\Diamond}\ \mathit{fault})$

However, PLTL has the same expressiveness as LTL: every PLTL formula has an equivalent LTL formula

In some pathological cases, the LTL formula equivalent to some PLTL formula P has size exponential in $|P|$. That is, PLTL is exponentially more succinct than LTL.

Automata-based model checking

History and tools

Model checking tools

Some notable model checking tools:

Spin is a versatile open-source explicit-state model checker. It was originally developed at Bell Labs in the 1980s, and has later been taken over to NASA's JPL by its inventor Gerard Holzmann.

NuSMV is a state-of-the-art symbolic model-checker based on SAT solving. It is a complete reimplementation of SMV, which was the first symbolic model checkers (based on BDDs).

TLA$^+$ is a specification language for systems and temporal properties based on logic, which extends Lamport's TLA (Temporal Logic of Actions). TLA$^+$'s toolset includes a model checker as well as interactive provers.

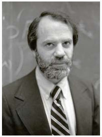
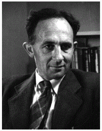
The Alloy Analyzer is sometimes referred to as a model checker, but it does not directly apply the automata-based approach of the other tools.

A brief history of model checking

Most of the ingredients of model checking were introduced by logicians decades before they were used in computer science.

A brief history of model checking

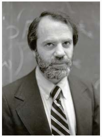
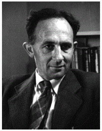
Most of the ingredients of model checking were introduced by logicians decades before they were used in computer science.

Kripke structures	1963	Saul Kripke	
Büchi automata	1960	Julius Büchi	
Temporal (tense) logic	1957	Arthur Prior	
Temporal logic	1968	Hans Kamp	

A brief history of model checking

Most of the ingredients of model checking were introduced by logicians decades before they were used in computer science.

Swiss, ETH graduate

Kripke structures	1963	Saul Kripke	
Büchi automata	1960	Julius Büchi	
Temporal (tense) logic	1957	Arthur Prior	
Temporal logic	1968	Hans Kamp	

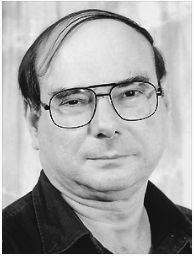
A brief history of model checking

Model checking brought those abstract ideas to practical fruition.

A brief history of model checking

Model checking brought those abstract ideas to practical fruition.

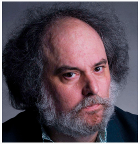
Around 1977, Amir Pnueli proposed to use temporal logic to express program properties.



A brief history of model checking

Model checking brought those abstract ideas to practical fruition.

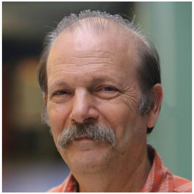
In 1981, the first model checking techniques were invented by Clarke and Emerson, and, independently, Queille and Sifakis.



A brief history of model checking

Model checking brought those abstract ideas to practical fruition.

The automata-theoretic framework was introduced by Vardi and Wolper around 1986.



A brief history of model checking

Model checking brought those abstract ideas to practical fruition.

Scalable model checking tools first appeared to the general public at the beginning of the 1990s.

For example, Holzmann's Spin and McMillan's SMV.

Software model checking

Model checking software?

In its most general meaning, software model checking denotes techniques that apply model checking to real software (actual code).

Model checking denotes a family of techniques for the algorithmic verification of finite-state systems with temporal-logic specifications.

Model checking software?

In its most general meaning, software model checking denotes techniques that apply model checking to real software (actual code).

Model checking denotes a family of techniques for the algorithmic verification of finite-state systems with temporal-logic specifications.

Most software is not finite state:

- arbitrary size inputs
- dynamic memory management
- unbounded recursion

Automatic software model checking with CEGAR

We have seen that finite-state models can capture important properties of real software – such as concurrent behavior – and can explore bounded behavior to systematically find bugs.

Automatic software model checking with CEGAR

We have seen that finite-state models can capture important properties of real software – such as concurrent behavior – and can explore bounded behavior to systematically find bugs.

In this lecture, however, we present a specific approach to software model checking that is:

- fully automatic – in particular, it builds finite-state models of software automatically
- analyzes arbitrary (executable) assertions of generic software behavior – it is not limited to concurrency or other aspects

CEGAR (CounterExample-Guided Abstraction Refinement) is a framework to automatically model check real software.

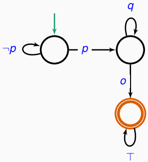
High level view of CEGAR model checking

CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[%])
```

PREDICATE ABSTRACTION



concrete

abstract

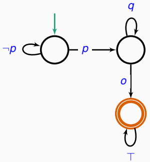
CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                    (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                    (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[%])
```

can execute (test)

PREDICATE ABSTRACTION



can model check

concrete

abstract

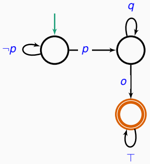
High level view of CEGAR model checking

CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                        (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                        (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[5])
```

PREDICATE ABSTRACTION



1. model check abstraction

concrete

abstract

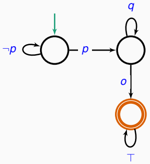
High level view of CEGAR model checking

CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                        (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                        (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[%])
```

PREDICATE ABSTRACTION



✖

counterexample

1. model check abstraction

concrete

abstract

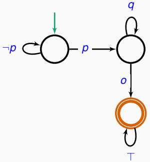
High level view of CEGAR model checking

CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[5])
```

PREDICATE ABSTRACTION



2. execute counterexample

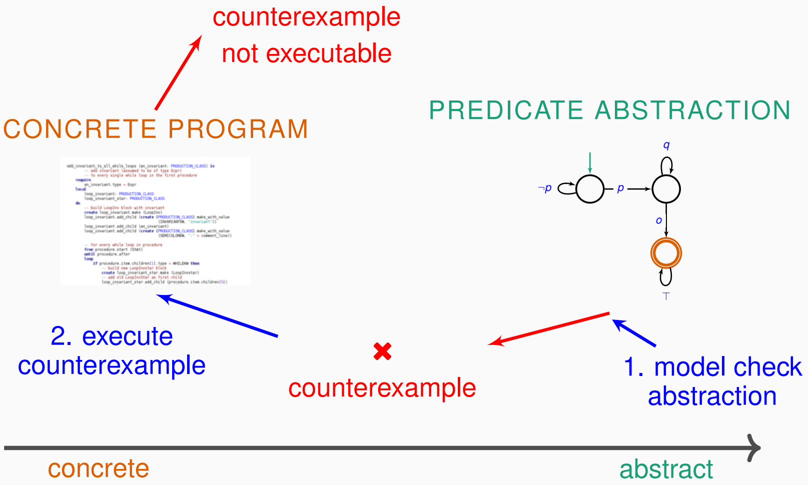
✖ counterexample

1. model check abstraction

concrete

abstract

High level view of CEGAR model checking



High level view of CEGAR model checking

counterexample
not executable

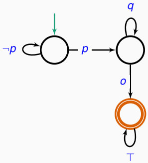
3. refine
abstraction

PREDICATE ABSTRACTION

CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[5])
```



2. execute
counterexample

✖
counterexample

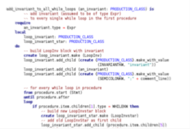
1. model check
abstraction

concrete

abstract

High level view of CEGAR model checking

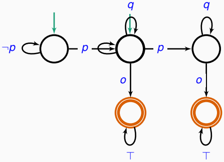
CONCRETE PROGRAM



```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                    (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                    (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[%])
```

PREDICATE ABSTRACTION



concrete

abstract

High level view of CEGAR model checking

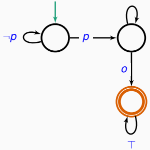
The loop continues until:

CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[%])
```

PREDICATE ABSTRACTION



start over
on new abstraction

concrete

abstract

High level view of CEGAR model checking

The loop continues until:

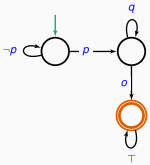
- the program is verified correct

PREDICATE ABSTRACTION

CONCRETE PROGRAM

```
add_invariant_to_all_while_loops (an_invariant: PRODUCTION_CLASS) is
        -- add invariant (assumed to be of type Expr)
        -- to every single while loop in the first procedure
    require
        an_invariant.type = Expr
    local
        loop_invariant: PRODUCTION_CLASS
        loop_invariant_star: PRODUCTION_CLASS
    do
        -- build LoopInv block with invariant
        create loop_invariant.make (LoopInv)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (INVARIANTKW, "invariant"))
        loop_invariant.add_child (an_invariant)
        loop_invariant.add_child (create {PRODUCTION_CLASS}.make_with_value
                                  (SEMICOLONKW, ";" + comment_line))

        -- for every while loop in procedure
        from procedure.start (Stmt)
        until procedure.after
        loop
            if procedure.item.children[1].type = WHILEKW then
                -- build new LoopInvstar block
                create loop_invariant_star.make (LoopInvstar)
                -- add old LoopInvStar as first child
                loop_invariant_star.add_child (procedure.item.children[%])
```



✔ program is correct

1. model check abstraction

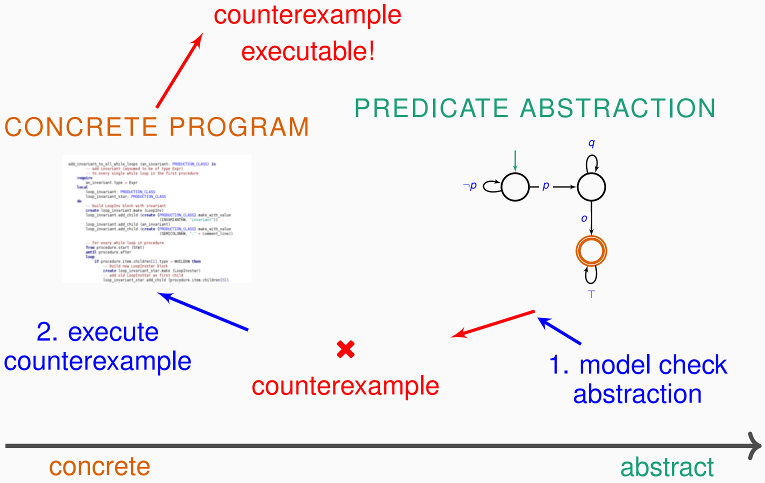
concrete

abstract

High level view of CEGAR model checking

The loop continues until:

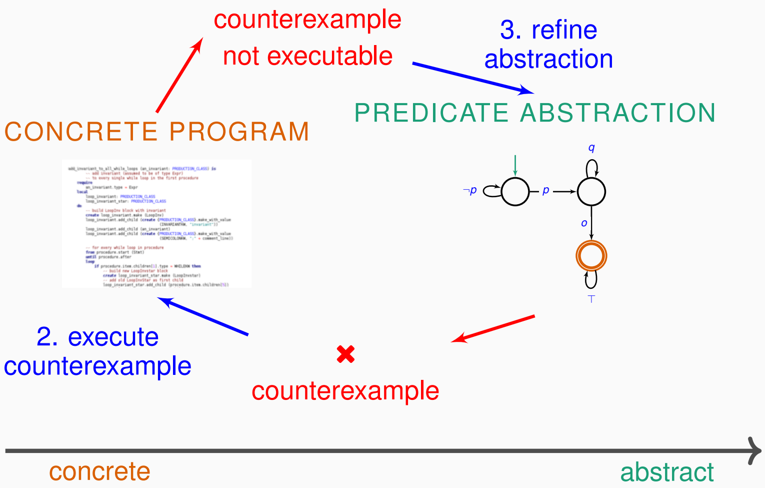
- a real bug is found



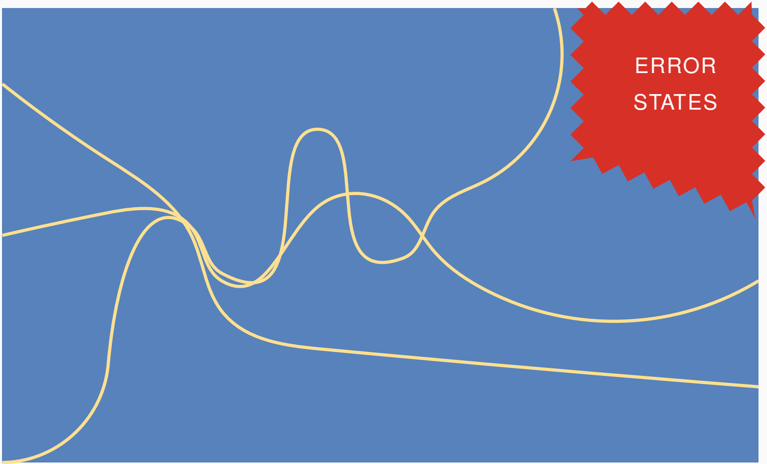
High level view of CEGAR model checking

The loop continues until:

- the abstraction is too big and we run out of memory

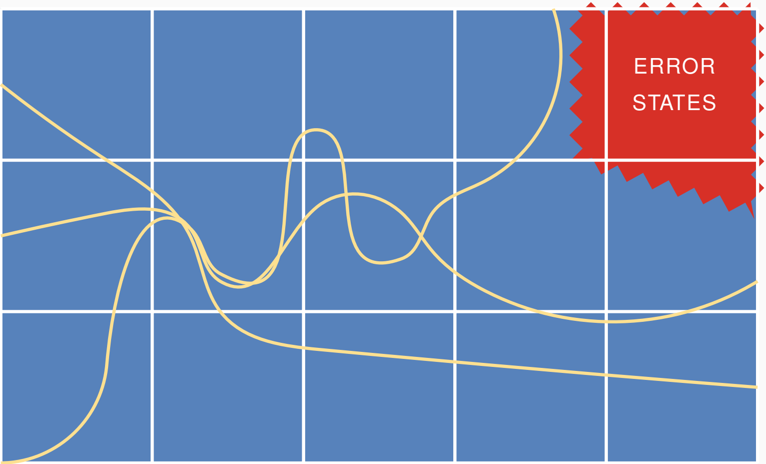


CEGAR model checking: overview



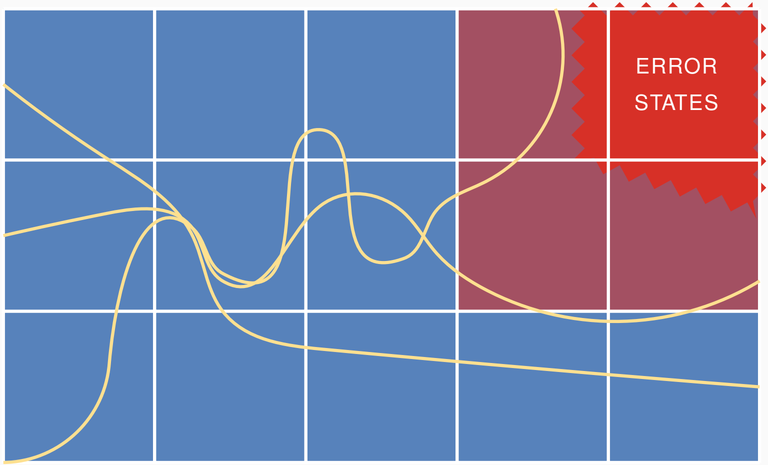
As usual, we want to analyze the behavior of a program, checking whether any runs enter error states.

CEGAR model checking: overview



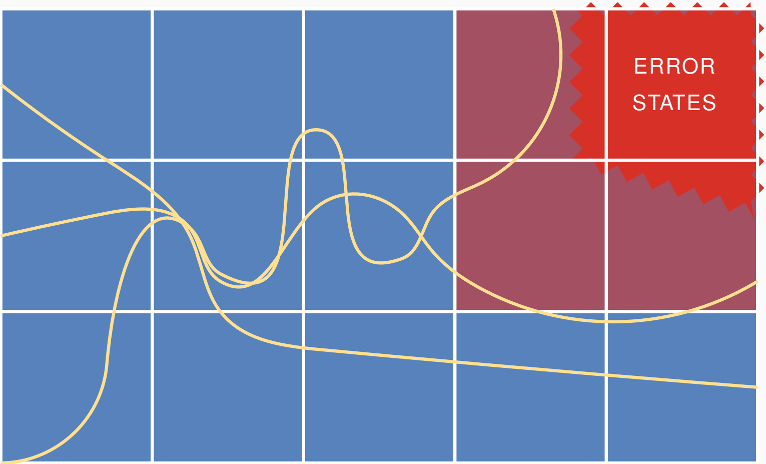
Instead of analyzing the program's infinite-state behavior, we build a finite-state abstraction of the program.

CEGAR model checking: overview



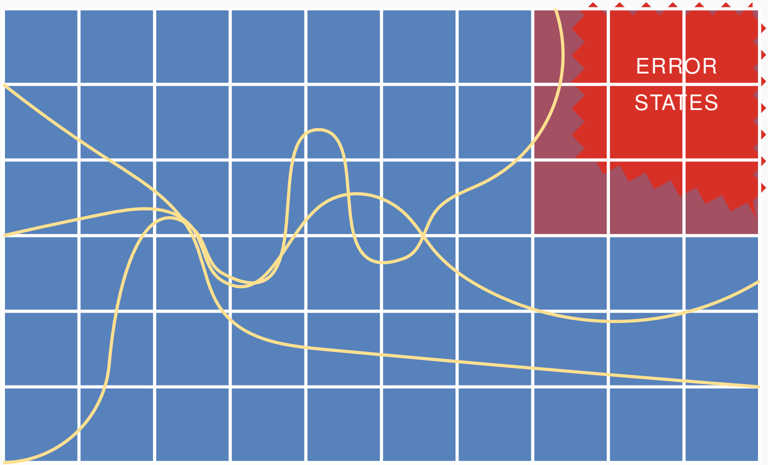
The finite-state abstraction is an over approximation of the program's behavior, and hence it is sound but introduces imprecision.

CEGAR model checking: overview



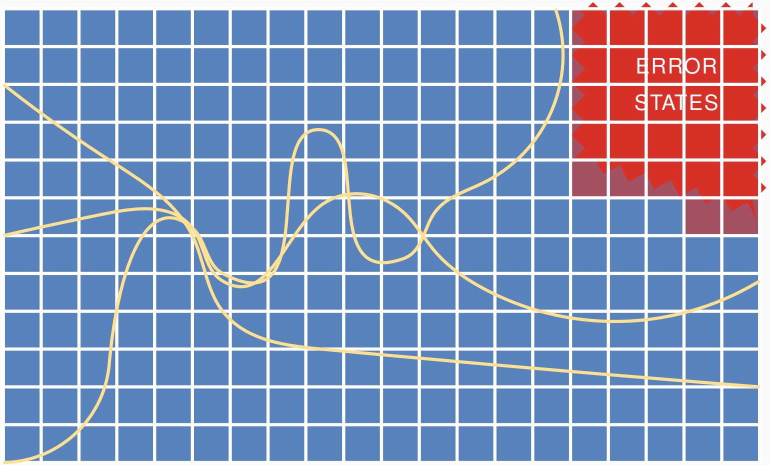
Because of imprecision, some counterexamples produced by model checking the finite-state abstraction may be spurious – that is false positives due to the abstraction's imprecision.

CEGAR model checking: overview



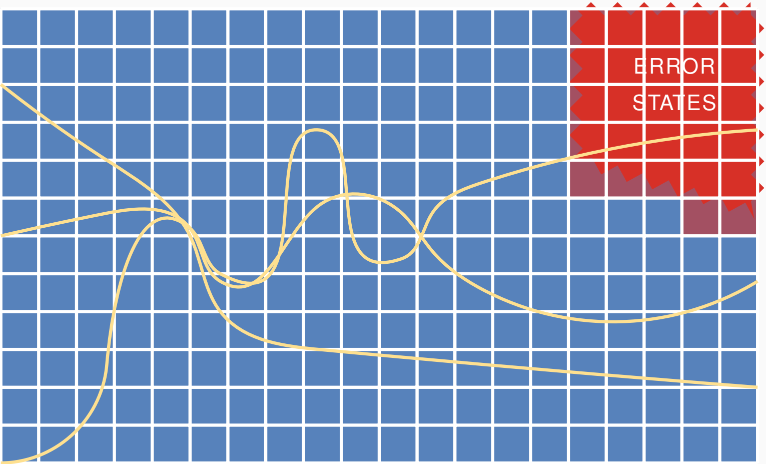
If we detect a spurious counterexample, we can refine the abstraction by making it more precise – but still finite state.

CEGAR model checking: overview



We can continue to refine until we verify the abstraction is correct.

CEGAR model checking: overview



Or until we find a true counterexample which reveals a real bug.

CEGAR software model checking

CEGAR model checking of software combines three program analysis techniques:

PREDICATE ABSTRACTION to build finite-state abstractions of programs that over approximate real behavior

SPURIOUS COUNTEREXAMPLE detection to determine whether an abstract counterexample is spurious or executable on the real program

PREDICATE DISCOVERY to refine the abstraction making it more precise

Software model checking

Predicate abstraction

Nondeterminism

To encode over approximations, we sometimes use nondeterministic value `?` as expressions in our programs.

Using value ? corresponds to introducing several computations – one for every value that `v` may take according to its type.

```
var b: Boolean
b := ?
```

corresponds to two computations:

- one where `b := true` executes
- another one where `b := false` executes

```
var v: Integer
if ?
   v := 3
else
   v := 10
```

corresponds to two computations:

- one where `v` is assigned value `3`
- another one where `v` is assigned value `10`

Nondeterminism

To encode over approximations, we sometimes use nondeterministic value `?` as expressions in our programs.

Using value ? corresponds to introducing several computations – one for every value that `v` may take according to its type.

```
var b: Boolean
b := ?
```

corresponds to two computations:

- one where `b := true` executes
- another one where `b := false` executes

```
var v: Integer
if ?
   v := 3
else
   v := 10
```

corresponds to two computations:

- one where `v` is assigned value `3`
- another one where `v` is assigned value `10`

Nondeterminism of this kind immediately translates to nondeterminism in a finite-state transition system or automaton.

Predicate sets

A predicate set B is a set of propositional symbols

$$B = \left\{b_1, b_2, \ldots, b_m\right\}$$

where proposition b_k corresponds to a quantifier-free predicate over program variables.

Example: $PE = \{(\texttt{x} \geq \texttt{0})^{\text{pos}}, (\texttt{y} = \texttt{x})^{\text{eq}}\}$ introduces symbol `pos` for predicate $\texttt{x} \geq \texttt{0}$, and symbol `eq` for predicate $\texttt{y} = \texttt{x}$.

Predicate sets

A predicate set B is a set of propositional symbols

$$B \quad = \quad \left\{b_1, b_2, \ldots, b_m\right\}$$

where proposition b_k corresponds to a quantifier-free predicate over program variables.

Example: $PE = \{(\mathtt{x} \geq \mathtt{0})^{\mathsf{pos}}, (\mathtt{y} = \mathtt{x})^{\mathsf{eq}}\}$ introduces symbol `pos` for predicate $\mathtt{x} \geq \mathtt{0}$, and symbol `eq` for predicate $\mathtt{y} = \mathtt{x}$.

When referring to a generic predicate set $\{b_1, b_2, \ldots, b_m\}$ we overload the notation so that b_k may indicate a propositional symbol or a predicate – which one it is will be always clear from the context.

Boolean programs

A Boolean program is a program that only uses Boolean variables.

A Boolean program over predicate set B is a program that only uses Boolean variables whose names are the propositional symbols in B.

In this presentation, a program is any Helium program – limited to integer and Boolean variables for simplicity.

Boolean programs

A Boolean program is a program that only uses Boolean variables.

A Boolean program over predicate set B is a program that only uses Boolean variables whose names are the propositional symbols in B.

In this presentation, a program is any Helium program – limited to integer and Boolean variables for simplicity.

Example Boolean program over $PE = \{(\texttt{x} \geq \texttt{0})^{\text{pos}}, (\texttt{y} = \texttt{x})^{\text{eq}}\}$:

```
var pos, eq: Boolean
if pos
  eq := true
else
  eq := ?
```

Abstract and concrete runs

A run is a sequence of states $s[0]\, s[1]\, s[2]\, \ldots\, s[n]$ that a complete program's execution goes through.

A program H defines a set $\mathcal{L}(H)$ of runs – all possible executions of H according to its operational semantics.

- If H is a Boolean program its runs are called abstract runs and are sequences of abstract states
- If H is a concrete program its runs are called concrete runs and are sequences of concrete states

```
var pos, eq: Boolean
if pos
  eq := true
else
  eq := ?
```

```
var x, y: Integer
if x ≥ 0
  y := x
else
  y := -x
```

Some abstract runs:

$a_1 = (\mathtt{pos}, \mathtt{eq})\,(\mathtt{pos}, \mathtt{eq})$

$a_2 = (\neg\mathtt{pos}, \mathtt{eq})\,(\neg\mathtt{pos}, \mathtt{eq})$

Some concrete runs:

$c_1 = (\mathtt{x} = 3, \mathtt{y} = 0)\,(\mathtt{x} = 3, \mathtt{y} = 3)$

$c_2 = (\mathtt{x} = -1, \mathtt{y} = -1)\,(\mathtt{x} = -1, \mathtt{y} = 1)$

Abstraction of concrete runs

The abstraction $A(c, B)$ of a concrete run $c = c[0]\, c[1] \, \ldots \, c[n]$ with respect to predicates B is the abstract run over B defined as:

$$B(c[0])\, B(c[1]) \, \ldots \, B(c[n])$$

where $B(c[k])$ is the Boolean value of predicates in B in state $c[k]$.

Example: given $PE = \{\texttt{(x} \geq \texttt{0)}^{\texttt{pos}}, \texttt{(y} = \texttt{x)}^{\texttt{eq}}\}$:

$$c_1 = (\mathtt{x} = 3, \mathtt{y} = 0)\,(\mathtt{x} = 3, \mathtt{y} = 3) \qquad A(c_1, PE) = (\texttt{pos}, \neg\texttt{eq})\,(\texttt{pos}, \texttt{eq})$$

$$c_2 = (\mathtt{x} = -1, \mathtt{y} = -1)\,(\mathtt{x} = -1, \mathtt{y} = 1) \qquad A(c_2, PE) = (\neg\texttt{pos}, \texttt{eq})\,(\neg\texttt{pos}, \neg\texttt{eq})$$

Abstraction of concrete programs

Thus, a Boolean program A over B also defines a set of concrete runs $\mathcal{C}(A)$: all runs of every program over variables in $\mathcal{V}(B)$ whose abstractions with respect to B are in $\mathcal{L}(A)$:

$$\mathcal{C}(A) \ = \ \bigcup\{\mathcal{L}(C) \mid A(C, B) \in \mathcal{L}(A)\}$$

The Boolean program over $PE = \{(\mathtt{x} \geq \mathtt{0})^{\mathsf{pos}}, (\mathtt{y} = \mathtt{x})^{\mathsf{eq}}\}$ on the left defines the concrete runs of the nondeterministic program over $\mathcal{V}(PE) = \{\mathtt{x}, \mathtt{y}\}$ on the right:

```
var pos, eq: Boolean
if pos
  eq := true
else
  eq := ?
```

```
var x, y: Integer
if ?
  x, y := ?, ?
  assume x ≥ 0
  assume x = y
else
  x, y := ?, ?
  assume x < 0
```

Predicate abstraction: definition

The predicate abstraction $PA(C, B)$ of program C with predicates B is a Boolean program over P which is the strongest over-approximation of C with respect to B.

over-approximation: $\mathcal{L}(C) \subseteq \mathcal{C}(PA(C, B))$

strongest: for every other Boolean program A over B that over-approximates C, it is $\mathcal{L}(PA(C, B)) \subseteq \mathcal{L}(A)$

The predicate abstraction is the most precise approximation of C given the information that is captured by the predicates B.

Predicate abstraction: definition

The predicate abstraction $PA(C, B)$ of program C with predicates B is a Boolean program over P which is the strongest over-approximation of C with respect to B.

over-approximation: $\mathcal{L}(C) \subseteq \mathcal{C}(PA(C, B))$

strongest: for every other Boolean program A over B that over-approximates C, it is $\mathcal{L}(PA(C, B)) \subseteq \mathcal{L}(A)$

The predicate abstraction is the most precise approximation of C given the information that is captured by the predicates B.

The Boolean program on the left is the predicate abstraction of the program on the right with $PE = \{(\mathtt{x} \geq 0)^{\mathsf{pos}}, (\mathtt{y} = \mathtt{x})^{\mathsf{eq}}\}$.

```
var pos, eq: Boolean
if pos
  eq := true
else
  eq := ?
```

```
var x, y: Integer
if x ≥ 0
  y := x
else
  y := -3
```

Predicate abstraction: informal overview

To over approximate a statement:

- compute what follows with certainty from the abstract state by computing the weakest under-approximation of the concrete state
- update the abstract state accordingly
- in all other cases, use nondeterministic values – corresponding to “don’t know” – to have an over-approximation

Predicate abstraction: informal overview

To over approximate a statement:

- compute what follows with certainty from the abstract state by computing the weakest under-approximation of the concrete state
- update the abstract state accordingly
- in all other cases, use nondeterministic values – corresponding to “don’t know” – to have an over-approximation

To over approximate a program: build the over approximation of each statement, while maintaining the same program structure.

```
var pos, eq: Boolean
if pos
  eq := true
else
  eq := ?
```

```
var x, y: Integer
if x ≥ 0
  y := x
else
  y := -3
```

Under approximation of concrete predicates

$\mathcal{U}(b, B)$ is the weakest Boolean expression over predicates B which is at least as strong as b (that is, an under approximation):

under-approximation: $\mathcal{U}(b, B) \Longrightarrow b$

weakest: for every Boolean expression a over B such that $a \Longrightarrow b$, it is $a \Longrightarrow \mathcal{U}(p, B)$

Under approximation of concrete predicates

$\mathcal{U}(b, B)$ is the weakest Boolean expression over predicates B which is at least as strong as b (that is, an under approximation):

under-approximation: $\mathcal{U}(b, B) \Longrightarrow b$

weakest: for every Boolean expression a over B such that $a \Longrightarrow b$, it is $a \Longrightarrow \mathcal{U}(p, B)$

To build $\mathcal{U}(b, B)$:

1. start from the strongest under-approximation: $\perp$
2. weaken the under-approximation by adding conjunctions of predicates (negated or unnegated), as long as the expression still implies b
3. stop after trying all possible conjunctions

Under approximations: example

$$B = \left\{ (\mathtt{x} = 1)^{p}, (\mathtt{x} = 2)^{q}, (\mathtt{x} \leq 3)^{r} \right\}$$

$$\mathcal{U}(\mathtt{x} = 1, B) \quad =$$

Under approximations: example

$$B = \left\{ (\mathtt{x} = \mathtt{1})^{p}, (\mathtt{x} = \mathtt{2})^{q}, (\mathtt{x} \leq \mathtt{3})^{r} \right\}$$

$$\begin{aligned} \mathcal{U}(\mathtt{x} = \mathtt{1}, B) &= p \\ \mathcal{U}(\mathtt{x} = \mathtt{0}, B) &= \end{aligned}$$

Under approximations: example

$$B = \left\{ (\mathtt{x} = \mathtt{1})^{\mathsf{p}}, (\mathtt{x} = \mathtt{2})^{\mathsf{q}}, (\mathtt{x} \leq \mathtt{3})^{\mathsf{r}} \right\}$$

$$\begin{aligned}
\mathcal{U}(\mathtt{x} = \mathtt{1}, B) &= p \\
\mathcal{U}(\mathtt{x} = \mathtt{0}, B) &= \perp \\
\mathcal{U}(\mathtt{x} \leq \mathtt{2}, B) &=
\end{aligned}$$

Under approximations: example

$$B = \left\{ (\mathtt{x} = \mathtt{1})^{p}, (\mathtt{x} = \mathtt{2})^{q}, (\mathtt{x} \leq \mathtt{3})^{r} \right\}$$

$$\begin{array}{lcl}
\mathcal{U}(\mathtt{x} = \mathtt{1}, B) & = & p \\
\mathcal{U}(\mathtt{x} = \mathtt{0}, B) & = & \perp \\
\mathcal{U}(\mathtt{x} \leq \mathtt{2}, B) & = & p \vee q \\
\mathcal{U}(\mathtt{x} \neq \mathtt{0}, B) & = &
\end{array}$$

Under approximations: example

$$B = \left\{ (\mathtt{x} = \mathtt{1})^{p}, (\mathtt{x} = \mathtt{2})^{q}, (\mathtt{x} \leq \mathtt{3})^{r} \right\}$$

$$\begin{array}{lcl}
\mathcal{U}(\mathtt{x} = \mathtt{1}, B) & = & p \\
\mathcal{U}(\mathtt{x} = \mathtt{0}, B) & = & \perp \\
\mathcal{U}(\mathtt{x} \leq \mathtt{2}, B) & = & p \vee q \\
\mathcal{U}(\mathtt{x} \neq \mathtt{0}, B) & = & p \vee q \vee \neg r
\end{array}$$

These examples show that, in general, $\mathcal{U}(\neg p, B) \neq \neg\mathcal{U}(p, B)$.

Under approximations: uniqueness

When some predicates in B imply some other predicates, $\mathcal{U}(b, B)$ may not be syntactically unique – that is, there may be different equivalent ways of expressing it.

Example:

$$B = \left\{ (\mathtt{x} < 2)^{p}, (\mathtt{x} \leq 2)^{q} \right\}$$

$$\begin{aligned} \mathcal{U}(\mathtt{x} \leq 3, B) &= p \vee q \\ &= q \end{aligned}$$

which is equivalent to

Under approximations: uniqueness

When some predicates in B imply some other predicates, $\mathcal{U}(b, B)$ may not be syntactically unique – that is, there may be different equivalent ways of expressing it.

Example:

$$B = \left\{ (\mathtt{x} < 2)^{\mathtt{p}}, (\mathtt{x} \leq 2)^{\mathtt{q}} \right\}$$

$$\begin{aligned} \mathcal{U}(\mathtt{x} \leq 3, B) &= p \vee q && \text{which is equivalent to} \\ &= q \end{aligned}$$

Using any equivalent variant is correct. However, we will be able to simplify the construction of predicate abstraction when the predicate set B does not have any predicates that imply some other.

Conditional Boolean expressions

To express program transformations involved in predicate abstraction, we will use this notation for conditional Boolean expressions:

$$(c : d : e)$$

which is a shorthand for:

```
if c then
  true
else (
  if d then
    false
  else
    e
)
```

if c then the expression evaluates to `true`

if d (and $\neg c$) then the expression evaluates to `false`

otherwise ($\neg c \wedge \neg d$) the expression evaluates to e

Predicate abstraction: assignments

The predicate abstraction $PA(s, B)$ of an assignment

$$s: \quad \texttt{v} \ \texttt{:=} \ E$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is the parallel conditional assignment

$$b_1, b_2, \ldots, b_m \ \texttt{:=} \ \texttt{(}c_1 \texttt{:} d_1 \texttt{:?)}\texttt{,}\texttt{(}c_2 \texttt{:} d_2 \texttt{:?)}\texttt{, ..., }\texttt{(}c_m \texttt{:} d_m \texttt{:?)}$$

where the kth component is defined as

$$c_k = \mathcal{U}(\overleftarrow{b_k}, B) \qquad\qquad d_k = \mathcal{U}(\overleftarrow{\neg b_k}, B)$$

and $\overleftarrow{b}$ is the weakest precondition $\mathbf{wp}(\texttt{v} \ \texttt{:=} \ E, b)$ of predicate b through the concrete assignment we are abstracting.

Abstraction of assignments: example

$$PA(\texttt{z := x}, B) \quad \text{where} \quad B = \{(\texttt{x} > \texttt{y})^{\texttt{p}}, (\texttt{z} \geq \texttt{x})^{\texttt{q}}, (\texttt{z} \geq \texttt{y})^{\texttt{r}}\}$$

The predicate abstraction is ($\mathcal{U}(x)$ is a shorthand for $\mathcal{U}(x, B)$):

$$\texttt{p, q, r :=} (\mathcal{U}(\overleftarrow{\texttt{p}}) : \mathcal{U}(\overleftarrow{\neg\texttt{p}}) : ?), (\mathcal{U}(\overleftarrow{\texttt{q}}) : \mathcal{U}(\overleftarrow{\neg\texttt{q}}) : ?), (\mathcal{U}(\overleftarrow{\texttt{r}}) : \mathcal{U}(\overleftarrow{\neg\texttt{r}}) : ?)$$

x	$\overleftarrow{x}$	$\mathcal{U}(\overleftarrow{x}, B)$
p	x > y	p
¬p	x ≤ y	¬p
q	x ≥ x	$\top$
¬q	x < x	$\bot$
r	x ≥ y	p
¬r	x < y	$\bot$

```
p, q, r := (p:¬ p:?),(true:false:?),(p:false:?)
```

Abstraction of assignments: intuition

$PA(\texttt{z := x}, B)$ where $B = \{(\texttt{x} > \texttt{y})^{\texttt{p}}, (\texttt{z} \geq \texttt{x})^{\texttt{q}}, (\texttt{z} \geq \texttt{y})^{\texttt{r}}\}$ is:

```
p, q, r := (p:¬p:?),(true:false:?),(p:false:?)
```

- **p** keeps its current value – since an assignment to z does not affect the value of $\texttt{x} > \texttt{y}$
- **q** becomes `true` unconditionally – since $\texttt{z} = \texttt{x}$ implies that $\texttt{z} \geq \texttt{x}$
- **r** becomes `true` if p, otherwise it is nondeterministically assigned – since $\texttt{z} = \texttt{x} > \texttt{y}$ implies $\texttt{z} \geq \texttt{y}$, but if $\texttt{z} = \texttt{x} \leq \texttt{y}$ we cannot conclude that $\texttt{z} < \texttt{y}$

Abstraction of assignments: intuition

$PA(\texttt{z := x}, B)$ where $B = \{(\texttt{x} > \texttt{y})^{\texttt{p}}, (\texttt{z} \geq \texttt{x})^{\texttt{q}}, (\texttt{z} \geq \texttt{y})^{\texttt{r}}\}$ is:

$$\texttt{p, q, r := (p:}\neg\texttt{p:?),(true:false:?),(p:false:?)}$$

- **p** keeps its current value – since an assignment to z does not affect the value of $\texttt{x} > \texttt{y}$
- **q** becomes `true` unconditionally – since $\texttt{z} = \texttt{x}$ implies that $\texttt{z} \geq \texttt{x}$
- **r** becomes `true` if p, otherwise it is nondeterministically assigned – since $\texttt{z} = \texttt{x} > \texttt{y}$ implies $\texttt{z} \geq \texttt{y}$, but if $\texttt{z} = \texttt{x} \leq \texttt{y}$ we cannot conclude that $\texttt{z} < \texttt{y}$

The conditional assignment can be rewritten into the equivalent, simpler form:

$$\texttt{q, r := true,(p:false:?)}$$

Abstraction of assignments: intuition

In general, the kth predicate $b_k \in B$ in $PA(\mathtt{v} := E, B)$ is updated to:

$$b_k \mathtt{:=} \ (\mathcal{U}(\overleftarrow{b_k}, B) \mathtt{:} \mathcal{U}(\overleftarrow{\neg b_k}, B) \mathtt{:?})$$

- $\overleftarrow{b_k}$ is the exact pre-condition that implies b_k after executing $\mathtt{v} := E$
- $\mathcal{U}(\overleftarrow{b_k}, B)$ is its best sound approximation expressible using predicates in B
- if $\mathcal{U}(\overleftarrow{b_k}, B)$ is true, we are sure that b_k is true after executing the assignment
- a similar reasoning shows that if $\mathcal{U}(\overleftarrow{\neg b_k}, B)$ is true, we are sure that b_k is false after executing the assignment
- in all other cases, we over-approximate by assigning a nondeterministic value to b_k

Abstraction of assignments: example

$$PA(\mathtt{y} \ := \ \mathtt{x}, B) \qquad \text{where} \qquad B = \{(\mathtt{x} \ = \ \mathtt{1})^{\mathtt{p}}, (\mathtt{y} \ = \ \mathtt{1})^{\mathtt{q}}, (\mathtt{x} \ > \ \mathtt{y})^{\mathtt{r}}\}$$

The predicate abstraction is ($\mathcal{U}(x)$ is a shorthand for $\mathcal{U}(x, B)$):

$$\mathtt{p,\ q,\ r} := (\mathcal{U}(\overleftarrow{\mathtt{p}}):\mathcal{U}(\overleftarrow{\neg\mathtt{p}}):?),(\mathcal{U}(\overleftarrow{\mathtt{q}}):\mathcal{U}(\overleftarrow{\neg\mathtt{q}}):?),(\mathcal{U}(\overleftarrow{\mathtt{r}}):\mathcal{U}(\overleftarrow{\neg\mathtt{r}}):?)$$

x	$\overleftarrow{x}$	$\mathcal{U}(\overleftarrow{x}, B)$
$\mathtt{p}$	$\mathtt{x} = \mathtt{1}$	$\mathtt{p}$
$\neg\mathtt{p}$	$\mathtt{x} \neq \mathtt{1}$	$\neg\mathtt{p}$
$\mathtt{q}$	$\mathtt{x} = \mathtt{1}$	$\mathtt{p}$
$\neg\mathtt{q}$	$\mathtt{x} \neq \mathtt{1}$	$\neg\mathtt{p}$
$\mathtt{r}$	$\mathtt{x} > \mathtt{x}$	$\bot$
$\neg\mathtt{r}$	$\mathtt{x} \leq \mathtt{x}$	$\top$

$$\mathtt{p,\ q,\ r} := (\mathtt{p}:\neg\,\mathtt{p}:?),(\mathtt{p}:\neg\,\mathtt{p}:?),(\mathtt{false}:\mathtt{true}:?)$$

equivalent to $\mathtt{q,\ r} := \mathtt{p},\mathtt{false}$.

Predicate abstraction: assertions

The predicate abstraction $PA(s, B)$ of an assert

$$s: \quad \texttt{assert}\ E$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is the assertion

$$\texttt{assert}\ \mathcal{U}(E, B)$$

Predicate abstraction: assertions

The predicate abstraction $PA(s, B)$ of an assert

$$s: \quad \texttt{assert}\ E$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is the assertion

$$\texttt{assert}\ \mathcal{U}(E, B)$$

This is an over-approximation because its correctness implies the original program's correctness, but there may be spurious failures:

- if $\mathcal{U}(E, B)$ holds in every abstract run, then E holds in every concrete run
- if $\mathcal{U}(E, B)$ fails in some abstract run, then E may or may not hold in every concrete run

Predicate abstraction: assumptions

The predicate abstraction $PA(s, B)$ of an assumption

$$s: \quad \texttt{assume}\ E$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is an assumption followed by a parallel conditional assignment

$$\texttt{assume}\ \neg\mathcal{U}(\neg E, B)$$

$$b_1, b_2, \ldots, b_m \texttt{ := } \texttt{(}c_1\texttt{:}d_1\texttt{:?),(}c_2\texttt{:}d_2\texttt{:?),} \ldots \texttt{,(}c_m\texttt{:}d_m\texttt{:?)}$$

where the kth component is defined as

$$c_k = \mathcal{U}(\overleftarrow{b_k}, B) \qquad\qquad d_k = \mathcal{U}(\overleftarrow{\neg b_k}, B)$$

and $\overleftarrow{b}$ is the weakest precondition $\mathbf{wp}(\texttt{assume}\ E, b)$ of predicate b through the concrete assumption we are abstracting.

Abstraction of assumptions: example

$$PA(\texttt{assume x} \leq \texttt{2}, B) \quad \text{where} \quad B = \{(\texttt{x = 1})^{\texttt{p}}, (\texttt{x = 2})^{\texttt{q}}, (\texttt{x} \leq \texttt{3})^{\texttt{r}}\}$$

The predicate abstraction is ($\mathcal{U}(x)$ is a shorthand for $\mathcal{U}(x, B)$):

$$\texttt{assume } \neg\mathcal{U}(\neg(\texttt{x} \leq \texttt{2}))$$

$$\texttt{p, q, r := } (\mathcal{U}(\overleftarrow{\texttt{p}}):\mathcal{U}(\overleftarrow{\neg\texttt{p}}):?), (\mathcal{U}(\overleftarrow{\texttt{q}}):\mathcal{U}(\overleftarrow{\neg\texttt{q}}):?), (\mathcal{U}(\overleftarrow{\texttt{r}}):\mathcal{U}(\overleftarrow{\neg\texttt{r}}):?)$$

x	$\overleftarrow{x}$	$\mathcal{U}(\overleftarrow{x}, B)$
	$\texttt{x} \leq \texttt{2}$	$\texttt{p} \vee \texttt{q}$
	$\texttt{x} > \texttt{2}$	$\neg\texttt{r}$
$\texttt{p}$	$\texttt{x} \leq \texttt{2} \implies \texttt{x = 1}$	$\texttt{p} \vee \neg\texttt{r}$
$\neg\texttt{p}$	$\texttt{x} \leq \texttt{2} \implies \texttt{x} \neq \texttt{1}$	$\neg\texttt{p} \vee \neg\texttt{r}$
$\texttt{q}$	$\texttt{x} \leq \texttt{2} \implies \texttt{x = 2}$	$\texttt{q} \vee \neg\texttt{r}$
$\neg\texttt{q}$	$\texttt{x} \leq \texttt{2} \implies \texttt{x} \neq \texttt{2}$	$\neg\texttt{q} \vee \neg\texttt{r}$
$\texttt{r}$	$\texttt{x} \leq \texttt{2} \implies \texttt{x} \leq \texttt{3}$	$\top$
$\neg\texttt{r}$	$\texttt{x} \leq \texttt{2} \implies \texttt{x} > \texttt{3}$	$\neg\texttt{r}$

`assume r` followed by the parallel conditional assignment

Abstraction of assumptions: intuition

$PA(\texttt{assume x} \leq 2, B)$ where $B = \{(\texttt{x} = 1)^{\texttt{p}}, (\texttt{x} = 2)^{\texttt{q}}, (\texttt{x} \leq 3)^{\texttt{r}}\}$ is:

```
assume r
p, q, r := (p ∨ ¬ r:¬ p ∨ ¬ r:?),(q ∨ ¬ r:¬ q ∨ ¬ r:?),true
```

The double negation $\texttt{assume}\ \neg\mathcal{U}(\neg E, B)$ builds, by duality, an over-approximation on the state from the under-approximation $\mathcal{U}$:

- $\texttt{assume}\ \mathcal{U}(\texttt{x} \leq 2, B) = \texttt{assume p} \vee \texttt{q}$: if we can prove the abstract program correct under the stronger assumption $\texttt{x} = 1 \vee \texttt{x} = 2$, the concrete program may still misbehave under the assumption $\texttt{x} \leq 2 \wedge \texttt{x} \neq 1 \wedge \texttt{x} \neq 2 = \texttt{x} < 1$
- $\texttt{assume}\ \neg\mathcal{U}(\texttt{x} > 2, B) = \texttt{assume r}$: whenever we can prove the abstract program correct under the weaker assumption $\texttt{x} \leq 3$, the concrete program is also correct under the stronger assumption $\texttt{x} \leq 2$

The parallel conditional assignment propagates the assumed state to all predicates correctly.

Predicate abstraction of assumptions: simplification

$$PA(\texttt{assume}\ E, B) \qquad \text{where} \qquad B = \{b_1, b_2, \ldots, b_m\} \qquad \text{is}$$

$$\texttt{assume}\ \neg\mathcal{U}(\neg E, B) \qquad \ldots, b_k, \ldots\ \texttt{:=}\ \ldots\texttt{,}\,\texttt{(}\mathcal{U}(\overleftarrow{b_k})\texttt{:}\mathcal{U}(\overleftarrow{\neg b_k})\texttt{:?)}\texttt{,}\ \ldots$$

If $E \Longrightarrow b_k$ and $E \Longrightarrow b_k$ are not unconditionally valid:

$$\mathcal{U}(\overleftarrow{b_k}) = \neg\mathcal{U}(\neg E) \Longrightarrow b_k \qquad \mathcal{U}(\overleftarrow{\neg b_k}) = \neg\mathcal{U}(\neg E) \Longrightarrow \neg b_k$$

which, under the assumption $\texttt{assume}\ \neg\mathcal{U}(\neg E)$, simplify to

$$\mathcal{U}(\overleftarrow{b_k}) = b_k \qquad \mathcal{U}(\overleftarrow{\neg b_k}) = \neg b_k$$

Thus $\quad b_k\ \texttt{:=}\ \texttt{(}b_k\texttt{:}\neg b_k\texttt{:?)} \quad$ which has no effect on b_k

For example:

$$\begin{aligned}
\mathcal{U}(\overleftarrow{b_k}) &= \mathcal{U}(E \Longrightarrow b_K) = \mathcal{U}(\neg E \vee b_k) && \text{definition of implication}\\
&= \mathcal{U}(\neg E) \vee \mathcal{U}(b_k) && \text{provable from the definition of } \mathcal{U}\\
&= \neg\mathcal{U}(\neg E) \Longrightarrow \mathcal{U}(b_k) && \text{definition of implication}
\end{aligned}$$

Predicate abstraction of assumptions: simplified rule

The predicate abstraction $PA(s, B)$ of an assumption

$$s: \quad \texttt{assume } E$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is an assumption

$$\texttt{assume } \neg\mathcal{U}(\neg E, B) \;\wedge\; \bigwedge_{k \in J^+} b_k \;\wedge\; \bigwedge_{k \in J^-} \neg b_k$$

where J^+ and J^- are defined as:

$$J^+ = \{k \mid E \Longrightarrow p_k \text{ is valid}\} \qquad J^- = \{k \mid E \Longrightarrow \neg p_k \text{ is valid}\}$$

Predicate abstraction: conditionals

The predicate abstraction $PA(s, B)$ of a conditional

$$s: \quad \texttt{if}\ (C)\ T\ \texttt{else}\ E$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is a nondeterministic conditional:

$$\texttt{if (?)}\ PA(\{\texttt{assume}\ C\ ;T\}, B)\ \texttt{else}\ PA(\{\texttt{assume}\ \neg C\ ;E\}, B)$$

that recursively applies predicate abstraction to the then and else branches under the right conditions.

Predicate abstraction: loops

The predicate abstraction $PA(s, B)$ of a loop

$$s: \quad \texttt{while}\,(C)\;L$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is a nondeterministic loop:

$$\Big\{\texttt{while (?)}\; PA(\{\texttt{assume}\; C\,;L\}, B)\Big\}\;\texttt{;}\; PA(\texttt{assume}\;\neg C, B)$$

that recursively applies predicate abstraction to the body L and to the exit condition of the loop.

Predicate abstraction: procedure definitions

The predicate abstraction $PA(s, B)$ of a procedure definition

$$s: \quad \texttt{procedure proc: require } P \texttt{ ensure } Q\ R$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is a program:

$$\texttt{var } b_1, b_2, \ldots, b_m \texttt{: Boolean ; } PA\left(\begin{array}{l}\texttt{assume } P \\ R \\ \texttt{assert } Q\end{array}, B\right)$$

Predicate abstraction: procedure definitions

The predicate abstraction $PA(s, B)$ of a procedure definition

$$s: \quad \texttt{procedure proc: require } P \texttt{ ensure } Q\ R$$

with predicates

$$B: \quad \{b_1, b_2, \ldots, b_m\}$$

is a program:

$$\texttt{var } b_1, b_2, \ldots, b_m \texttt{: Boolean ; } PA\left(\begin{array}{l} \texttt{assume } P \\ R \\ \texttt{assert } Q \end{array}, B \right)$$

In our examples we always choose B so that precondition P and postcondition Q can be expressed exactly.

Predicate abstraction: complete program example

$$B = \{(\texttt{x} > \texttt{y})^{\texttt{p}}, (\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

Predicate abstraction: complete program example

$$B = \{(\texttt{x} > \texttt{y})^{p}, (\texttt{max} \geq \texttt{x})^{q}, (\texttt{max} \geq \texttt{y})^{r}\}$$

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

```
var p, q, r: Boolean
if x > y
  max := x
else
  max := y
assert max ≥ x ∧ max ≥ y
```

Predicate abstraction: complete program example

$$B = \{(\mathtt{x} > \mathtt{y})^{\mathtt{p}}, (\mathtt{max} \geq \mathtt{x})^{\mathtt{q}}, (\mathtt{max} \geq \mathtt{y})^{\mathtt{r}}\}$$

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

```
var p, q, r: Boolean
if ?
  assume x > y
  max := x
else
  assume x ≤ y
  max := y
assert max ≥ x ∧ max ≥ y
```

Predicate abstraction: complete program example

$$B = \{(\texttt{x} > \texttt{y})^{\mathsf{p}}, (\texttt{max} \geq \texttt{x})^{\mathsf{q}}, (\texttt{max} \geq \texttt{y})^{\mathsf{r}}\}$$

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

```
var p, q, r: Boolean
if ?
  assume p
  q, r := true, (p:false:?)
else
  assume ¬ p
  max := y
assert q ∧ r
```

Predicate abstraction: complete program example

$$B = \{(\texttt{x} > \texttt{y})^{\texttt{p}}, (\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

```
var p, q, r: Boolean
if ?
  assume p
  q, r := true, (p:false:?)
else
  assume ¬p
  q, r := (¬p:false:?), true
assert q ∧ r
```

Predicate abstraction: complete program example

$$B = \{(\mathtt{x} > \mathtt{y})^{\mathsf{p}}, (\mathtt{max} \geq \mathtt{x})^{\mathsf{q}}, (\mathtt{max} \geq \mathtt{y})^{\mathsf{r}}\}$$

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

```
var p, q, r: Boolean
if ?
  assume p
  q, r := true, true
else
  assume ¬ p
  q, r := true, true
assert q ∧ r
```

Predicate abstraction: complete program example

$$B = \{(\texttt{x} > \texttt{y})^{\mathsf{p}}, (\texttt{max} \geq \texttt{x})^{\mathsf{q}}, (\texttt{max} \geq \texttt{y})^{\mathsf{r}}\}$$

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

```
procedure PA_max
  (p, q, r: Boolean):
ensure q ∧ r
  if p
    q, r := true, true
  else
    q, r := true, true
```

Model checking Boolean programs

Every Boolean program $P = \ell_1, \ldots, \ell_n$ over variables $B = \{b_1, \ldots, b_m\}$ defines a finite-state transition system $\langle S, I, R, B, L \rangle$, which can be model checked.

Model checking Boolean programs

Every Boolean program $P = \ell_1, \ldots, \ell_n$ over variables $B = \{b_1, \ldots, b_m\}$ defines a finite-state transition system $\langle S, I, R, B, L \rangle$, which can be model checked.

S: $\mathbb{B}^k \times \{1, \ldots, n+1, \mathit{error}\}$
The states are the set of all Boolean values of variables in B, plus a program counter indicating the next statement in P to be executed (or $n+1$ if the program has terminated). There is also a distinct *error* location.

I: $\mathbb{B}^k \times \{1\}$
The initial states are all those pointing to the first statement ℓ_1 in program P.

L: $L((p_1, \ldots, p_m, k)) = k$
The labeling function just projects the program location of each state.

Model checking Boolean programs

Every Boolean program $P = \ell_1, \ldots, \ell_n$ over variables $B = \{b_1, \ldots, b_m\}$ defines a finite-state transition system $\langle S, I, R, B, L \rangle$, which can be model checked.

R: $(p_1, \ldots, p_m, k)\ R\ (q_1, \ldots, q_m, h)$ iff one of the following holds:

- $\ell_k :\ p_x$ `:=` E, $e = E[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$, $q_1, \ldots, q_m = p_1, \ldots, p_m[p_x \mapsto e]$, and $h = k + 1$ is the next statement
- $\ell_k :$, $e = E[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$, $q_1, \ldots, q_m = p_1, \ldots, p_m[p_x \mapsto e]$, and $h = k + 1$ is the next statement
- $\ell_k :$ `assert` E, $E[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$ evaluates to true in the pre-state, $q_1, \ldots, q_m = p_1, \ldots, p_m[p_x \mapsto e]$, and $h = k + 1$ is the next statement
- $\ell_k :$ `assert` E, $E[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$ evaluates to false in the pre-state, and $h = \mathit{error}$ the error location
- $\ell_k :$ `assume` E, $E[b_1, \ldots, b_m \mapsto q_1, \ldots, q_m]$ evaluates to true in the post-state, and $h = k + 1$ is the next statement
- $\ell_k :$ **`if`** `C {T}`^{t} **`else`** `{E}`^{e}, condition $C[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$ evaluates to true in the pre-state, $q_1, \ldots, q_m = p_1, \ldots, p_m$, and $h = t$ is the then branch's location
- $\ell_k :$ **`if`** `C {T}`^{t} **`else`** `{E}`^{e}, condition $C[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$ evaluates to false in the pre-state, $q_1, \ldots, q_m = p_1, \ldots, p_m$, and $h = e$ is the else branch's location
- $\ell_k :$ **`while`** `C {L}`^{i} `;`ℓ_o, condition $C[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$ evaluates to true in the pre-state, $q_1, \ldots, q_m = p_1, \ldots, p_m$, and $h = i$ is the loop body's first statement
- $\ell_k :$ **`while`** `C {L}`^{i} `;`ℓ_o, condition $C[b_1, \ldots, b_m \mapsto p_1, \ldots, p_m]$ evaluates to false in the pre-state, $q_1, \ldots, q_m = p_1, \ldots, p_m$, and $h = o$ is the first statement after the loop

Model checking Boolean programs

Every Boolean program $P = \ell_1, \ldots, \ell_n$ over variables $B = \{b_1, \ldots, b_m\}$ defines a finite-state transition system $\langle S, I, R, B, L \rangle$, which can be model checked.

With this model, a Boolean program is correct iff there are no assertion violations, iff model checking verifies $P \models \Box(\neg \mathit{error})$.

Software model checking

Spurious counterexample detection

Verification of Boolean programs

The predicate abstraction $PA(C, B)$ of program C with predicates B is a Boolean program over P which is the strongest over-approximation of C with respect to B.

$PA(C, B)$ ✔	$PA(C, B)$ ✖
C is correct	model checking may return: • a real counterexample (true positive) if C is not correct • a spurious counterexample (false positive) if C is correct

Verification of Boolean programs

The predicate abstraction $PA(C, B)$ of program C with predicates B is a Boolean program over P which is the strongest over-approximation of C with respect to B.

$PA(C, B)$ ✔	$PA(C, B)$ ✖
C is correct	model checking may return: • a real counterexample (true positive) if C is not correct • a spurious counterexample (false positive) if C is correct

We present a technique to detect if an abstract counterexample is spurious (not executable) or real (executable).

Run concretization

Every statement in a concrete program C correspond to one (possibly compound) statement in its predicate abstraction $PA(C, B)$. Thus, given an abstract run we can always build a sequence of concrete statements of the concrete program the abstract program emulated.

ABSTRACT RUN	CONCRETE RUN
a_0	
A_1	C_1
a_1	
A_2	C_2
$\vdots$	$\vdots$
A_N	C_N
a_N	

- $a_0\ a_1\ \ldots\ a_N$ is the abstract run's sequence of abstract states
- $A_1\ \ldots\ A_N$ are the run's abstract statements executed
- $C_1\ \ldots\ C_N$ are the corresponding concrete statements

Run concretization: example

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$

```
procedure QR_max
  (q, r: Boolean):
ensure q ∧ r
  if ?
    q, r := true, ?
  else
    q, r := ?, true
```

ABSTRACT RUN A^1	CONCRETE RUN C^1
$q \wedge \neg r$	
`if ?`	`if x > y`
$q \wedge \neg r$	
`q, r := true, ?`	`max := x`
$q \wedge \neg r$	

Run concretization: example

```
procedure max
  (x, y: Integer): (max: Integer)
ensure max ≥ x ∧ max ≥ y
  if x > y
    max := x
  else
    max := y
```

$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$

```
procedure QR_max
  (q, r: Boolean):
ensure q ∧ r
  if ?
    q, r := true, ?
  else
    q, r := ?, true
```

ABSTRACT RUN A^2	CONCRETE RUN C^2
$q \wedge \neg r$	
`if ?`	`if x > y`
$q \wedge \neg r$	
`q, r := ?, true`	`max := y`
$q \wedge r$	

Run feasibility constraints

To determine if a concrete run is feasible (executable):

- transform every concrete branch statement C (**if** E, **while** E, assume E, and assert E) into a feasibility constraint C' : assert E
- compute the weakest precondition of a_n through each concrete statement C_k

The concrete run C obtained from an abstract run A is feasible iff, for every $k = 0, \ldots, N$, $a_k \wedge c_k$ is satisfiable.

ABSTRACT RUN	CONCRETE RUN	
a_0	c_0	$= \mathbf{wp}(C_1, c_1)$
A_1	C'_1	
a_1	c_1	$= \mathbf{wp}(C_2, c_2)$
A_2	C'_2	
$\vdots$	$\vdots$	
a_{N-1}	c_{N-1}	$= \mathbf{wp}(C_N, c_N)$
A_N	C'_N	
a_N	c_N	$= a_N$

Run feasibility constraints

To determine if a concrete run is feasible (executable):

- transform every concrete branch statement C (`if` E, `while` E, `assume` E, and `assert` E) into a feasibility constraint C' : `assert` E
- compute the weakest precondition of a_n through each concrete statement C_k

The concrete run C obtained from an abstract run A is feasible iff, for every $k = 0, \ldots, N$, $a_k \wedge c_k$ is satisfiable.

ABSTRACT RUN	CONCRETE RUN	
a_0	c_0	$= \mathbf{wp}(C_1, c_1)$
A_1	C_1'	
a_1	c_1	$= \mathbf{wp}(C_2, c_2)$
A_2	C_2'	
$\vdots$	$\vdots$	
a_{N-1}	c_{N-1}	$= \mathbf{wp}(C_N, c_N)$
A_N	C_N'	
a_N	c_N	$= a_N$

every model of $a_0 \wedge c_0$ gives concrete inputs that determine run C

c_{N-1} is the requirement for a computation to reach and execute C_N

Spurious run detection

An abstract run A is spurious iff
there exists $k = 0, \ldots, N$ such that $a_k \wedge c_k$ is unsatisfiable.

Spurious run detection can be automated:

- computing the weakest precondition along a run does not involve loops, and hence can be automated
- an SMT solver can decide satisfiability of $a_k \wedge c_k$ (if it uses a decidable fragment)

ABSTRACT RUN	CONCRETE RUN	
a_0	c_0	$= \mathbf{wp}(C_1, c_1)$
A_1	C_1'	
a_1	c_1	$= \mathbf{wp}(C_2, c_2)$
A_2	C_2'	
$\vdots$	$\vdots$	
a_{N-1}	c_{N-1}	$= \mathbf{wp}(C_N, c_N)$
A_N	C_N'	
a_N	c_N	$= a_N$

Spurious run detection: example

An abstract run A is spurious iff
there exists $k = 0, \ldots, N$ such that $a_k \wedge c_k$ is unsatisfiable.

$$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

Spurious run detection: example

An abstract run A is spurious iff
there exists $k = 0, \ldots, N$ such that $a_k \wedge c_k$ is unsatisfiable.

$$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

ABSTRACT RUN A^1	CONCRETE RUN C^1
$q \wedge \neg r$	$\texttt{x} > \texttt{y} \wedge \texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$
`if ?`	`assert x > y`
$q \wedge \neg r$	$\texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$
`q, r := true, ?`	`max := x`
$q \wedge \neg r$	$\texttt{max} \geq \texttt{x} \wedge \texttt{max} < \texttt{y}$

$\texttt{x} > \texttt{y} \wedge \texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$ is unsatisfiable,
and hence the abstract run is spurious.
There is no concrete run that is abstracted by the abstract run A^1.

Spurious run detection: example

An abstract run A is spurious iff
there exists $k = 0, \ldots, N$ such that $a_k \wedge c_k$ is unsatisfiable.

$$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

ABSTRACT RUN A^2	CONCRETE RUN C^2
$q \wedge \neg r$	$\texttt{x} \leq \texttt{y} \wedge \texttt{y} \geq \texttt{x} \wedge \texttt{y} \geq \texttt{y}$
`if ?`	`assert x <= y`
$q \wedge \neg r$	$\texttt{y} \geq \texttt{x} \wedge \texttt{y} \geq \texttt{y}$
`q, r := ?, true`	`max := y`
$q \wedge r$	$\texttt{max} \geq \texttt{x} \wedge \texttt{max} \geq \texttt{y}$

The abstract run is feasible. Every input x, y that satisfies $\texttt{x} \leq \texttt{y} \equiv \texttt{x} \leq \texttt{y} \wedge \texttt{y} \geq \texttt{x} \wedge \texttt{y} \geq \texttt{y}$ determines a concrete run that is abstracted by the abstract run A^2.

Concrete counterexample detection: example

Is procedure `negpow` correct with respect to its specification?

```
procedure negpow
    (x, y: Integer): (pow: Integer)
require x < 0 ∧ y > 0
ensure pow > 0
  pow := 1
  while y > 0
    pow := pow * x
    y := y - 1
```

Concrete counterexample detection: example

Is procedure `negpow` correct with respect to its specification?

Let's analyze it using a predicate abstraction over predicates:

$$NP = \{(\mathtt{x} < \mathtt{0})^{p}, (\mathtt{y} > \mathtt{0})^{q}, (\mathtt{pow} > \mathtt{0})^{r}\}$$

```
procedure negpow
    (x, y: Integer): (pow: Integer)
require x < 0 ∧ y > 0
ensure pow > 0
  pow := 1
  while y > 0
    pow := pow * x
    y := y - 1
```

```
procedure NP_negpow
    (p, q, r: Boolean):
require p ∧ q
ensure r
  r := true
  while q
    r := (false:p ∧ r:?)
    q := ?
```

Concrete counterexample detection: example

Model checking `NP_negpow` finds the abstract counterexample A^e:

ABSTRACT RUN A^e	CONCRETE RUN C^e
$p \wedge q \wedge r$	
`r := true`	`pow := 1`
$p \wedge q \wedge r$	
`while q`	`assert y > 0`
$p \wedge q \wedge r$	
`r := (false:p ∧ r: ?)`	`pow := pow * x`
$p \wedge q \wedge \neg r$	
`q := ?`	`y := y - 1`
$p \wedge \neg q \wedge \neg r$	
`while q`	`assert y ≤ 0`
$p \wedge \neg q \wedge \neg r$	

$$NP = \{(\texttt{x} < 0)^{\texttt{q}}, (\texttt{y} > 0)^{\texttt{q}}, (\texttt{pow} > 0)^{\texttt{r}}\}$$

Concrete counterexample detection: example

Model checking `NP_negpow` finds the abstract counterexample A^e:

ABSTRACT RUN A^e	CONCRETE RUN C^e
$p \land q \land r$	`y > 0` $\land$ `x < 0` $\land$ `y ≤ 1` $\land$ `x ≤ 0`
`r := true`	`pow := 1`
$p \land q \land r$	`y > 0` $\land$ `x < 0` $\land$ `y ≤ 1` $\land$ `pow * x ≤ 0`
`while` `q`	`assert y > 0`
$p \land q \land r$	`x < 0` $\land$ `y ≤ 1` $\land$ `pow * x ≤ 0`
`r := (false:p ∧ r: ?)`	`pow := pow * x`
$p \land q \land \neg r$	`x < 0` $\land$ `y ≤ 1` $\land$ `pow ≤ 0`
`q := ?`	`y := y - 1`
$p \land \neg q \land \neg r$	`x < 0` $\land$ `y ≤ 0` $\land$ `pow ≤ 0`
`while` `q`	`assert y ≤ 0`
$p \land \neg q \land \neg r$	`x < 0` $\land$ `y ≤ 0` $\land$ `pow ≤ 0`

$$NP = \{(\texttt{x < 0})^{\texttt{q}}, (\texttt{y > 0})^{\texttt{q}}, (\texttt{pow > 0})^{\texttt{r}}\}$$

Concrete counterexample detection: example

Model checking `NP_negpow` finds the abstract counterexample A^e:

ABSTRACT RUN A^e	CONCRETE RUN C^e
$p \land q \land r$	$\texttt{y} > 0 \land \texttt{x} < 0 \land \texttt{y} \le 1 \land \texttt{x} \le 0$
`r := true`	`pow := 1`
$p \land q \land r$	$\texttt{y} > 0 \land \texttt{x} < 0 \land \texttt{y} \le 1 \land \texttt{pow} * \texttt{x} \le 0$
`while q`	`assert y > 0`
$p \land q \land r$	$\texttt{x} < 0 \land \texttt{y} \le 1 \land \texttt{pow} * \texttt{x} \le 0$
`r := (false:p ∧ r: ?)`	`pow := pow * x`
$p \land q \land \neg r$	$\texttt{x} < 0 \land \texttt{y} \le 1 \land \texttt{pow} \le 0$
`q := ?`	`y := y - 1`
$p \land \neg q \land \neg r$	$\texttt{x} < 0 \land \texttt{y} \le 0 \land \texttt{pow} \le 0$
`while q`	`assert y <= 0`
$p \land \neg q \land \neg r$	$\texttt{x} < 0 \land \texttt{y} \le 0 \land \texttt{pow} \le 0$

$$NP = \{(\texttt{x} < 0)^{\texttt{q}}, (\texttt{y} > 0)^{\texttt{q}}, (\texttt{pow} > 0)^{\texttt{r}}\}$$

The abstract counterexample is feasible. It reveals a real error, triggered for every input that satisfies $\texttt{x} < 0 \land \texttt{y} = 1$.

Software model checking

Predicate discovery

Refinement

A spurious counterexample indicates that the predicate abstraction $PA(C, B)$ is too imprecise to conclusively verify C.

In these case, we want to refine the abstraction – getting a more precise one.

Refinement

A spurious counterexample indicates that the predicate abstraction $PA(C, B)$ is too imprecise to conclusively verify C.

In these case, we want to refine the abstraction – getting a more precise one.

Predicate discovery is a technique to find a new predicate $b \notin B$ such that $PA(C, B \cup \{b\})$ refines $PA(C, B)$.

Predicate discovery starts from a spurious counterexample A of $PA(C, B)$ and finds b such that A is not feasible in $PA(C, B \cup \{b\})$.

Syntactic predicate discovery

We look for a predicate b that:

- holds in the concrete run
- is not traced by any predicates in B
- makes the abstract state unsatisfiable

ABSTRACT RUN	CONCRETE RUN	
a_0	c_0	$= \mathbf{wp}(C_1, c_1)$
A_1	C'_1	
a_1	c_1	$= \mathbf{wp}(C_2, c_2)$
A_2	C'_2	
$\vdots$	$\vdots$	
a_{N-1}	c_{N-1}	$= \mathbf{wp}(C_N, c_N)$
A_N	C'_N	
a_N	c_N	$= a_N$

Syntactic predicate discovery

We look for a predicate b that:

- holds in the concrete run
- is not traced by any predicates in B
- makes the abstract state unsatisfiable

$\mathcal{P}(F)$ denotes the set of all elementary predicates in a formula F.

ABSTRACT RUN	CONCRETE RUN	
a_0	c_0	$= \mathbf{wp}(C_1, c_1)$
A_1	C'_1	
a_1	c_1	$= \mathbf{wp}(C_2, c_2)$
A_2	C'_2	
$\vdots$	$\vdots$	
a_{N-1}	c_{N-1}	$= \mathbf{wp}(C_N, c_N)$
A_N	C'_N	
a_N	c_N	$= a_N$

Syntactic predicate discovery

$\mathcal{P}(F)$ denotes the set of all elementary predicates in a formula F.

For any k such that $c_k \wedge a_k$ is unsatisfiable, $B' = \mathcal{P}(c_k) \setminus \mathcal{P}(a_k)$ gives new predicates. Any $b \in B'$ that is not equivalent to $\mathcal{U}(b, B)$ refines the abstraction.

ABSTRACT RUN	CONCRETE RUN	
a_0	c_0	$= \mathbf{wp}(C_1, c_1)$
A_1	C'_1	
a_1	c_1	$= \mathbf{wp}(C_2, c_2)$
A_2	C'_2	
$\vdots$	$\vdots$	
a_{N-1}	c_{N-1}	$= \mathbf{wp}(C_N, c_N)$
A_N	C'_N	
a_N	c_N	$= a_N$

Syntactic predicate discovery

$\mathcal{P}(F)$ denotes the set of all elementary predicates in a formula F.

For any k such that $c_k \wedge a_k$ is unsatisfiable, $B' = \mathcal{P}(c_k) \setminus \mathcal{P}(a_k)$ gives new predicates. Any $b \in B'$ that is not equivalent to $\mathcal{U}(b, B)$ refines the abstraction.

ABSTRACT RUN	CONCRETE RUN		NEW PREDICATES
a_0	c_0	$= \mathbf{wp}(C_1, c_1)$	$\mathcal{P}(c_0) \setminus \mathcal{P}(a_0)$
A_1	C'_1		
a_1	c_1	$= \mathbf{wp}(C_2, c_2)$	$\mathcal{P}(c_1) \setminus \mathcal{P}(a_1)$
A_2	C'_2		
$\vdots$	$\vdots$		
a_{N-1}	c_{N-1}	$= \mathbf{wp}(C_N, c_N)$	$\mathcal{P}(c_{N-1}) \setminus \mathcal{P}(a_{N-1})$
A_N	C'_N		
a_N	c_N	$= a_N$	$\mathcal{P}(c_N) \setminus \mathcal{P}(a_N)$

Predicate discovery: example

$$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

ABSTRACT RUN A^1	CONCRETE RUN C^1
$q \wedge \neg r$	$\texttt{x} > \texttt{y} \wedge \texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$
`if ?`	`assert x > y`
$q \wedge \neg r$	$\texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$
`q, r := true, ?`	`max := x`
$q \wedge \neg r$	$\texttt{max} \geq \texttt{x} \wedge \texttt{max} < \texttt{y}$

$\texttt{x} > \texttt{y} \wedge \texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$ is unsatisfiable,
and hence the abstract run is spurious.

Predicate discovery: example

$$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

ABSTRACT RUN A^1	CONCRETE RUN C^1	$c_k \wedge a_k$ SAT?
$q \wedge \neg r$	$\texttt{x} > \texttt{y} \wedge \texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$	✗
`if ?`	`assert x > y`	
$q \wedge \neg r$	$\texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$	✓
`q, r := true, ?`	`max := x`	
$q \wedge \neg r$	$\texttt{max} \geq \texttt{x} \wedge \texttt{max} < \texttt{y}$	✓

Predicate discovery: example

$$QR = \{(\texttt{max} \geq \texttt{x})^{\texttt{q}}, (\texttt{max} \geq \texttt{y})^{\texttt{r}}\}$$

ABSTRACT RUN A^1	CONCRETE RUN C^1	$c_k \wedge a_k$ SAT?	$\mathcal{P}(c_k) \setminus \mathcal{P}(a_k)$
$q \wedge \neg r$	$\texttt{x} > \texttt{y} \wedge \texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$	✖	$\{\texttt{x} > \texttt{y}\}$
`if ?`	`assert x > y`		
$q \wedge \neg r$	$\texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$	✔	$\{\texttt{x} < \texttt{y}\}$
`q, r := true, ?`	`max := x`		
$q \wedge \neg r$	$\texttt{max} \geq \texttt{x} \wedge \texttt{max} < \texttt{y}$	✔	$\{\ \}$

Predicate discovery: example

$$QR = \{(\texttt{max} \geq \texttt{x})^{q}, (\texttt{max} \geq \texttt{y})^{r}\}$$

ABSTRACT RUN A^1	CONCRETE RUN C^1	$c_k \wedge a_k$ SAT?	$\mathcal{P}(c_k) \setminus \mathcal{P}(a_k)$
$q \wedge \neg r$	$\texttt{x} > \texttt{y} \wedge \texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$	✖	$\{\texttt{x} > \texttt{y}\}$
`if ?`	`assert x > y`		
$q \wedge \neg r$	$\texttt{x} \geq \texttt{x} \wedge \texttt{x} < \texttt{y}$	✔	$\{\texttt{x} < \texttt{y}\}$
`q, r := true, ?`	`max := x`		
$q \wedge \neg r$	$\texttt{max} \geq \texttt{x} \wedge \texttt{max} < \texttt{y}$	✔	$\{\,\}$

We use $\texttt{x} > \texttt{y}$ for refinement since it contradicts the abstract state, is a new predicate, and $\mathcal{U}(\texttt{x} > \texttt{y}, QR) = \neg q \wedge r \not\Leftarrow \texttt{x} > \texttt{y}$.

As we already know, the refined abstraction $PA(\texttt{max}, QR \cup \{\texttt{x} > \texttt{y}\})$ passes verification without spurious counterexamples.

Software model checking

Software model checking tools

A short demo of CPAchecker

CPAchecker is a software model checker for C code, which includes numerous analysis techniques based on automatic abstraction – including specialized variants of the CEGAR model checking algorithm we have seen.

Let’s run some of the examples we have seen before using CPAchecker.

A short demo of CPAchecker: examples

By default, CPAchecker analyzes programs looking for paths that reach a special program label `ERROR`. We can encode assertions as reachability as usual.

```c
int max(int x, int y)
{
  int max;
  if (x > y)
        max = x;
  else
        max = y;
  if (max >= x && max >= y)
        return max;
  else
  ERROR: return max;
}
```

```c
int negpow(int x, int y)
{
  int pow;
  if (!(x < 0 && y > 0))
        return;
  pow = 1;
  while (y > 0) {
        pow *= x;
        y--;
  }
  if (pow > 0)
        return pow;
  else
  ERROR: return pow;
}
```

A short demo of CPAchecker: analysis and results

The script `cpa` provided with the Docker image runs CPAchecker using the analysis `predicateAnalysis-PredAbsRefiner-SBE`, which is a form of CEGAR analysis.

```
> cpa max.c
```

The tool produces a detailed output and error report in subdirectory `output`. The Docker image includes the main HTML reports for each example:

1. `max.c` (correctly verified): `max.ok.html`
2. `max.c` (error after assigning `max = y` in both branches): `max.error.html`
3. `negpow.c` (real error): `negpow.error.html`
4. `maxa.c` (analysis breaks down – even if we inline the function call): `maxa.error.html`

A short demo of CPAchecker: unsupported features

- Function `maxa` computes the maximum `max` of input array `a`
- Function `is_max` checks that `max` is indeed an upper bound on the values in `a`

The checking loop, which is executed a variable number of times depending on the size `n` of `a`, cannot be abstracted effectively using CPAchecker's analysis.

```c
int maxa(int a[], int n)
{
  int max;
  if (n <= 0)
        return;
  max = a[0];
  for (int k = 1; k < n; k++) {
        if (a[k] > max)
              max = a[k];
  }
  is_max(a, n, max);
}
```

```c
int is_max(int a[], int n, int m)
{
  for (int k = 0; k < n; k++) {
        if (a[k] > m)
              goto ERROR;
  }
  return 1;
 ERROR:
  return 0;
}
```

A short demo of CPAchecker: unsupported features

- Function `maxa` computes the maximum `max` of input array `a`
- Function `is_max` checks that `max` is indeed an upper bound on the values in `a`

The checking loop, which is executed a variable number of times depending on the size `n` of `a`, cannot be abstracted effectively using CPAchecker's analysis.

CPAchecker can scale to much larger programs than this small example, but only if the properties we want to check are simple and can be effectively captured with a reasonable number of predicates.

Is software model checking model checking?

Model checking proper is only one ingredient of CEGAR software model checking:

PREDICATE ABSTRACTION uses static analysis (expressible as abstract interpretation)

SPURIOUS COUNTEREXAMPLE detection uses deductive verification

PREDICATE DISCOVERY is based on symbolic execution of traces

Is software model checking model checking?

Model checking proper is only one ingredient of CEGAR software model checking:

PREDICATE ABSTRACTION uses static analysis (expressible as abstract interpretation)

SPURIOUS COUNTEREXAMPLE detection uses deductive verification

PREDICATE DISCOVERY is based on symbolic execution of traces

> *The term "software model checker" is probably a misnomer [...] We retain the term solely to reflect historical development.*
>
> *Jhala & Majumdar, 2009*

Software model checking tools

Microsoft Research's SLAM (Ball, Rajamani, et al.) was the first complete CEGAR model checker.

UC Berkeley's BLAST (Jhala, Majumdar, Henzinger, et al.) was the first open-source CEGAR model checker, which introduced several improvements to scalability.

Software model checkers that are currently maintained include:

CPAchecker is an extensible configurable verifier for C code, with a modular architecture that includes CEGAR-style verification algorithms

CBMC is a verifier for C/C++ using bounded model checking, which is part of a toolset that includes tools for Java (JBMC) and for predicate abstraction (SatAbs)

ESBMC is a bounded model checker for C/C++ supporting the analysis of multi-threaded programs

JPF (Java PathFinder) is a model checker for Java source code supporting concurrency and abstraction

Real-time model checking

Real-time systems

Real-time model checking applies model checking techniques to <u>finite-state</u> models of real-time systems.

Real-time systems

Real-time model checking applies model checking techniques to finite-state models of real-time systems.

A real-time specification is a specification that includes quantitative (exact) timing information.

A real-time system is a system whose correctness is defined by a real-time specification.

Real-time systems

Real-time model checking applies model checking techniques to finite-state models of real-time systems.

A real-time specification is a specification that includes quantitative (exact) timing information.

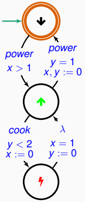
A real-time system is a system whose correctness is defined by a real-time specification.

Many embedded computing devices are real-time systems:

- a car's braking system (ABS)
- a multi-media player (or smart TV)
- Boeing 737 MAX's Maneuvering Characteristics Augmentation System
- …

Model-checking real-time systems

Real-time model checking verifies timed automata models against real-time temporal-logic specifications



$$\Box\left(\begin{array}{c} \mathit{cook} \\ \Longrightarrow \\ \Diamond_{<1} \neg \mathit{cook} \end{array}\right)$$

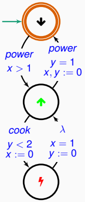
model checker A

✔ correct

✖ incorrect: timed counterexample
$\langle \mathit{power}, 1.1\rangle \, \langle \mathit{cook}, 1.5\rangle \, \langle \lambda, 2.5\rangle \, \ldots$

Model-checking real-time systems

Real-time model checking verifies timed automata models against real-time temporal-logic specifications



timed automaton, or other automata models of real-time behavior

some flavor of real-time temporal logic

$$\Box \left(\begin{array}{c} \mathit{cook} \\ \Longrightarrow \\ \Diamond_{<1} \neg \mathit{cook} \end{array} \right)$$

model checker A

✔ correct

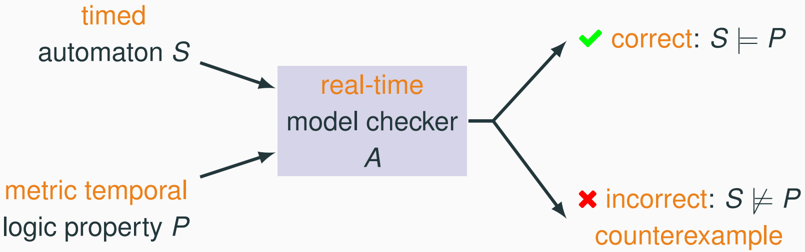
✖ incorrect: timed counterexample $\langle \mathit{power}, 1.1 \rangle \langle \mathit{cook}, 1.5 \rangle \langle \lambda, 2.5 \rangle \dots$

Real-time model checking of timed automata and MTL

Real-time model checking problem: given

- S: a timed automaton (TA)
- P: a metric temporal logic (MTL) property

determine if every run of S satisfies P, or
provide a counterexample: a timed run of S that violates P

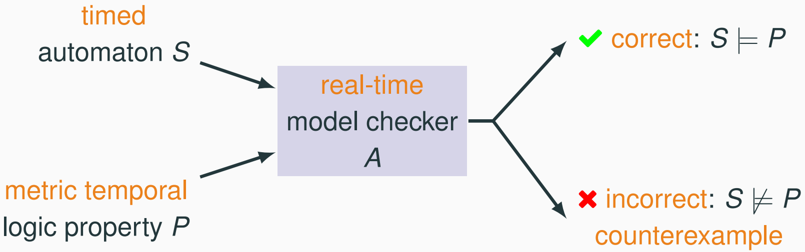


Real-time model checking of timed automata and MTL

Real-time model checking problem: given

- S: a timed automaton (TA)
- P: a metric temporal logic (MTL) property

determine if every run of S satisfies P, or
provide a counterexample: a timed run of S that violates P



We first describe syntax and semantics of TAs and MTL, and then describe an algorithm for real-time model checking.

Dense vs. discrete time domain

A fundamental choice in real-time modeling is the nature of the time domain.

DENSE/CONTINUOUS	DISCRETE
time domain: $\mathbb{R}_{\geq 0}$	time domain: $\mathbb{N}$
sequence of isolated instants	arbitrarily close instants in time

Dense vs. discrete time domain

A fundamental choice in real-time modeling is the nature of the time domain.

DENSE/CONTINUOUS	DISCRETE
time domain: $\mathbb{R}_{\geq 0}$	time domain: $\mathbb{N}$
sequence of isolated instants	arbitrarily close instants in time
can model truly asynchronous behavior	can only approximate continuous dynamics

Dense vs. discrete time domain

A fundamental choice in real-time modeling is the nature of the time domain.

DENSE/CONTINUOUS	DISCRETE
time domain: $\mathbb{R}_{\geq 0}$	time domain: $\mathbb{N}$
sequence of isolated instants	arbitrarily close instants in time
can model truly asynchronous behavior	can only approximate continuous dynamics
verification usually highly complex	verification reduces to "untimed" case (standard model checking)

Dense vs. discrete time domain

A fundamental choice in real-time modeling is the nature of the time domain.

DENSE/CONTINUOUS	DISCRETE
time domain: $\mathbb{R}_{\geq 0}$	time domain: $\mathbb{N}$
sequence of isolated instants	arbitrarily close instants in time
can model truly asynchronous behavior	can only approximate continuous dynamics
verification usually highly complex	verification reduces to "untimed" case (standard model checking)

This lecture focuses on the more challenging problem of using a dense time domain $\mathbb{R}_{\geq 0}$.

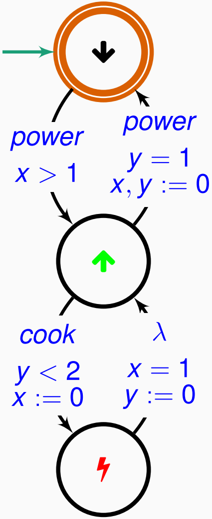
Later, we will also mention the complexity of the simpler real-time model checking using the discrete time domain $\mathbb{N}$.

Real-time model checking

Timed automata

Timed automata: example

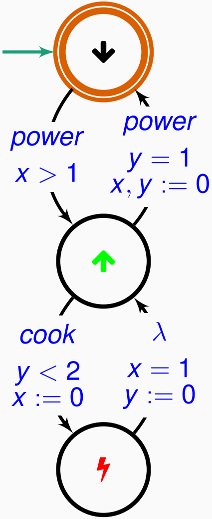
We model the timed behavior of a microwave oven using a TA, which is a finite state automaton equipped with clocks.



The oven may be powered <u>on</u> ⬆ or <u>off</u> ⬇.

Timed automata: example

We model the timed behavior of a microwave oven using a TA, which is a finite state automaton equipped with clocks.

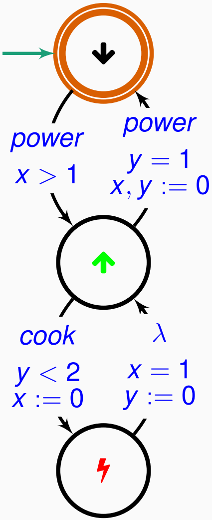


The oven may be powered on or off.

When the oven is on, pressing the cook button *cook* starts cooking.

Timed automata: example

We model the timed behavior of a microwave oven using a TA, which is a finite state automaton equipped with clocks.



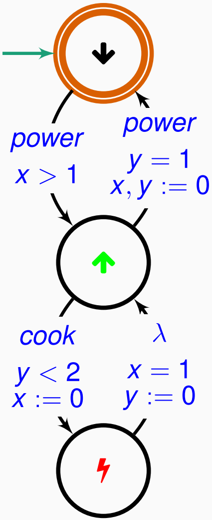
The oven may be powered on ↑ or off ↓.

When the oven is on, pressing the cook button *cook* starts cooking ⚡.

Each cooking round lasts exactly one time unit.

Timed automata: example

We model the timed behavior of a microwave oven using a TA, which is a finite state automaton equipped with clocks.



The oven may be powered on 🡅 or off 🡇.

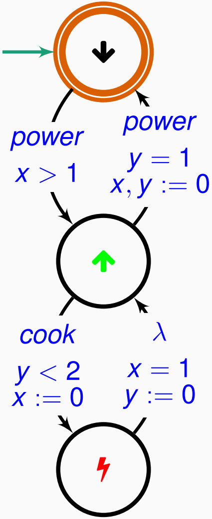
When the oven is on, pressing the cook button *cook* starts cooking ϟ.

Each cooking round lasts exactly one time unit.

When cooking ends, you can turn off the oven exactly after another time unit.

Timed automata: example

We model the timed behavior of a microwave oven using a TA, which is a finite state automaton equipped with clocks.



The oven may be powered on or off.

When the oven is on, pressing the cook button *cook* starts cooking.

Each cooking round lasts exactly one time unit.

When cooking ends, you can turn off the oven exactly after another time unit.

After the oven is turned off, you have to one time unit before turning it on again.

Timed automata: syntax

A nondeterministic timed automaton (TA) A
is a tuple $\langle \Sigma, S, I, F, C, E \rangle$:

- Σ: finite nonempty input alphabet
- S: finite nonempty set of locations (discrete states)
- $I, F \subseteq S$: initial and final (accepting) states
- C: finite set of clocks
- $E \subseteq S \times \Sigma \times \chi \times C \times S$: finite set of transitions (edges)

An edge $E \ni e = (s_1, \sigma, c, \rho, s_2)$ indicates a transition:

- from location s_1 to location s_2
- reading input symbol $\sigma \in \Sigma$
- subject to clock constraint $c \in \chi$
- resetting clocks in $\rho \subseteq C$

Timed automata: syntax

A nondeterministic timed automaton (TA) A
is a tuple $\langle \Sigma, S, I, F, C, E \rangle$:

- Σ: finite nonempty input alphabet
- S: finite nonempty set of locations (discrete states)
- $I, F \subseteq S$: initial and final (accepting) states
- C: finite set of clocks
- $E \subseteq S \times \Sigma \times \chi \times C \times S$: finite set of transitions (edges)

An edge $E \ni e = (s_1, \sigma, c, \rho, s_2)$ indicates a transition:

- from location s_1 to location s_2
- reading input symbol $\sigma \in \Sigma$
- subject to clock constraint $c \in \chi$
- resetting clocks in $\rho \subseteq C$

We commonly represent TAs with a graph similar to FSAs.

Clock constraints: syntax

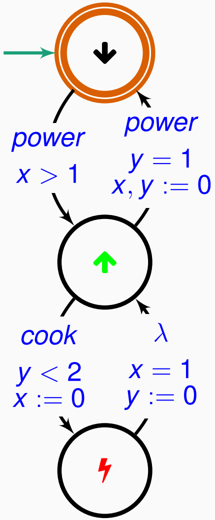
A clock constraint c is Boolean combination
of predicates comparing a clock to an integer constant:

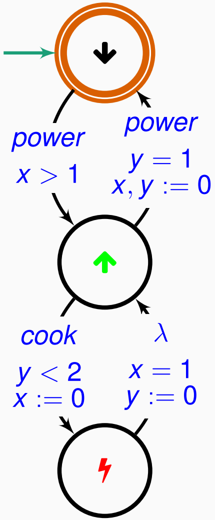
$$\chi \ni c ::= \; x \bowtie n \mid \neg c \mid c_1 \wedge c_2 \mid c_1 \vee c_2$$

where $\bowtie$ can be any of $<, \leq, =, \neq, \geq, >$
and $n \in \mathbb{N}$ can be any nonnegative integer (natural number).

Note that clock constraints compare clocks to integers even though the clock themselves range over the nonnegative reals. Later we will comment on the impact of this restriction.

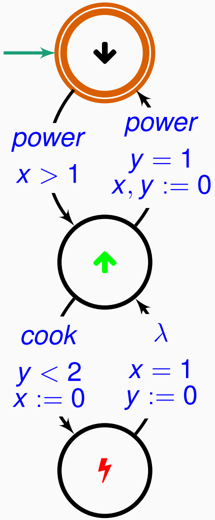
TA syntax: example





alphabet $\Sigma = \{power, cook, \lambda\}$

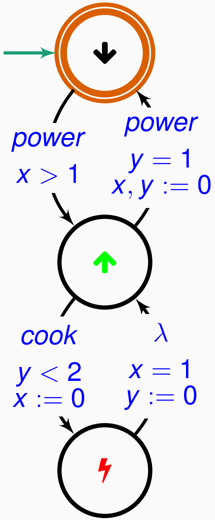
TA syntax: example



alphabet $\Sigma = \{power, cook, \lambda\}$

locations $S = \{\downarrow, \uparrow, ϟ\}$

TA syntax: example

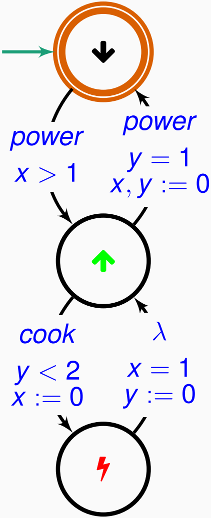


alphabet $\Sigma = \{power, cook, \lambda\}$

locations $S = \{$🡇, 🡅, ϟ$\}$

initial and final location $I = F = \{$🡇$\}$

TA syntax: example



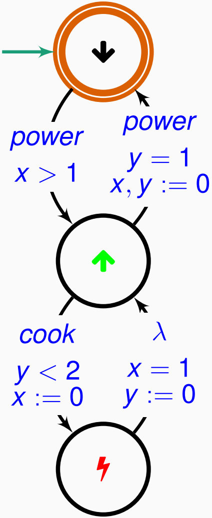
alphabet $\Sigma = \{power, cook, \lambda\}$

locations $S = \{$🡇, 🡅, ϟ$\}$

initial and final location $I = F = \{$🡇$\}$

clocks $C = \{x, y\}$

TA syntax: example



alphabet $\Sigma = \{power, cook, \lambda\}$

locations $S = \{\downarrow, \uparrow, \lightning\}$

initial and final location $I = F = \{\downarrow\}$

clocks $C = \{x, y\}$

transitions E:

- $(\downarrow, power, x > 1, \{\ \}, \uparrow)$
- $(\uparrow, power, y < 2, \{x\}, \lightning)$
- $(\lightning, \lambda, x = 1, \{y\}, \uparrow)$
- $(\uparrow, power, y = 1, \{x, y\}, \downarrow)$

Timed words

Let $A = \langle \Sigma, S, I, F, C, E \rangle$ be a TA.

An input timed word is an input sequence of any (finite) length with timestamps (time values ranging over the nonnegative reals):

$$\begin{aligned} w = \; & w[1]\, w[2] \, \ldots \, w[n] \in (\Sigma \times \mathbb{R}_{\geq 0})^* \\ = \; & \langle \sigma[1], t[1] \rangle \, \langle \sigma[2], t[2] \rangle \, \ldots \, \langle \sigma[n], t[n] \rangle \end{aligned}$$

such that the sequence $t[1]\, t[2] \, \ldots \, t[n]$ of timestamps is increasing.

A timed word element $\langle \sigma[k], t[k] \rangle$ denotes input $\sigma[k]$ read at time $t[k]$.

As in the untimed case, the empty word ϵ has zero length ($n = |\epsilon| = 0$).

Timed words

Let $A = \langle \Sigma, S, I, F, C, E \rangle$ be a TA.

An input timed word is an input sequence of any (finite) length with timestamps (time values ranging over the nonnegative reals):

$$\begin{aligned} w = {} & w[1]\, w[2]\, \ldots\, w[n] \in (\Sigma \times \mathbb{R}_{\geq 0})^* \\ = {} & \langle \sigma[1], t[1] \rangle \langle \sigma[2], t[2] \rangle \ldots \langle \sigma[n], t[n] \rangle \end{aligned}$$

such that the sequence $t[1]\, t[2]\, \ldots\, t[n]$ of timestamps is increasing.

A timed word element $\langle \sigma[k], t[k] \rangle$ denotes input $\sigma[k]$ read at time $t[k]$.

As in the untimed case, the empty word ϵ has zero length ($n = |\epsilon| = 0$).

Examples:

w_1: $\langle power, 3.0 \rangle$

w_2: $\langle power, 1.3 \rangle \langle cook, 1.7 \rangle \langle \lambda, 2.7 \rangle \langle power, 3.7 \rangle$

Timed automata: runs

A run of A over w is a sequence of states (location + clock values):

$$r = r[0]\, r[1] \ldots r[n] \in (S \times \mathbb{R}_{\geq 0}^{|C|})^*$$
$$= \langle s[0], x_1[0], \ldots, x_{|C|}[0] \rangle \ldots \langle s[n], x_1[n], \ldots, x_{|C|}[n] \rangle \qquad \text{that:}$$

- starts from an initial location with all clocks reset to zero: $s[0] \in I$ and, for all $1 \leq k \leq |C|$, $x_k[0] = 0$
- follows A's transitions: for all $0 \leq i < n$, the ith transition $\langle s[i], x_1[i], \ldots, x_{|C|}[i] \rangle \longrightarrow \langle s[i+1], x_1[i+1], \ldots, x_{|C|}[i+1] \rangle$
  - follows an edge: $(s[i], \sigma[i+1], c, \rho, s[i+1])$
  - the updated clock values $x_1[i] + \Delta_i, \ldots, x_{|C|}[i] + \Delta_i$ satisfy clock constraint c, where $\Delta_i = t[i+1] - t[i]$ is the time spent in $s[i]$
  - all clocks $x_r \in \rho$ are reset: $x_r[i+1] = 0$
  - all other clocks $x_u \notin \rho$ are updated: $x_u[i+1] = x_u[i] + \Delta_i$

Timed automata: runs

A run of A over w is a sequence of states (location + clock values):

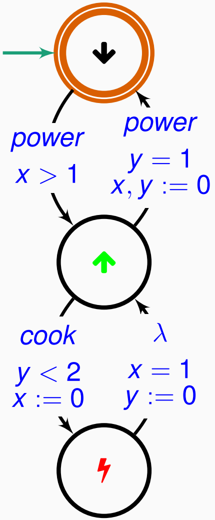
$$r = r[0]\, r[1] \ \ldots\ r[n] \in (S \times \mathbb{R}_{\geq 0}^{|C|})^*$$

$$= \langle s[0], x_1[0], \ldots, x_{|C|}[0] \rangle \ \ldots\ \langle s[n], x_1[n], \ldots, x_{|C|}[n] \rangle \qquad \text{that:}$$

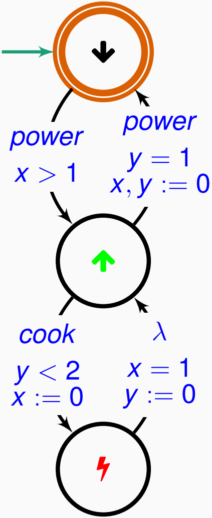
- starts from an initial location with all clocks reset to zero: $s[0] \in I$ and, for all $1 \leq k \leq |C|$, $x_k[0] = 0$
- follows A's transitions: for all $0 \leq i < n$, the ith transition $\langle s[i], x_1[i], \ldots, x_{|C|}[i] \rangle \longrightarrow \langle s[i+1], x_1[i+1], \ldots, x_{|C|}[i+1] \rangle$
  - follows an edge: $(s[i], \sigma[i+1], c, \rho, s[i+1])$
  - the updated clock values $x_1[i] + \Delta_i, \ldots, x_{|C|}[i] + \Delta_i$ satisfy clock constraint c, where $\Delta_i = t[i+1] - t[i]$ is the time spent in $s[i]$
  - all clocks $x_r \in \rho$ are reset: $x_r[i+1] = 0$
  - all other clocks $x_u \notin \rho$ are updated: $x_u[i+1] = x_u[i] + \Delta_i$

A run r is accepting if it ends in a final location: $s[n] \in F$.
In this case we say that A accepts w.

TA runs: example

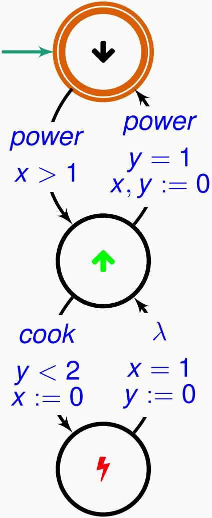


TA runs: example



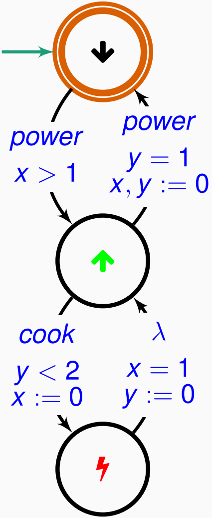
Run $r_1 = \langle$ ↓ $, x = 0, y = 0\rangle\ \langle$ ↑ $, x = 3.0, y = 3.0\rangle$
over $w_1 = \langle power, 3.0\rangle$

TA runs: example



Run $r_1 = \langle$🡇$, x = 0, y = 0\rangle\ \langle$🡅$, x = 3.0, y = 3.0\rangle$
over $w_1 = \langle power, 3.0\rangle$
is not accepting.

TA runs: example



Run $r_1 = \langle \downarrow, x = 0, y = 0\rangle \, \langle \uparrow, x = 3.0, y = 3.0\rangle$
over $w_1 = \langle power, 3.0\rangle$
is not accepting.

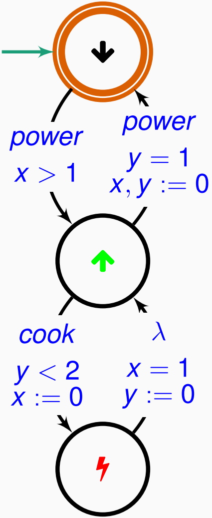
Run

$$r_2 = \langle \downarrow, x = 0, y = 0\rangle \, \langle \uparrow, x = 1.3, y = 1.3\rangle$$
$$\langle ϟ, x = 0, y = 1.7\rangle \, \langle \uparrow, x = 1, y = 0\rangle \, \langle \downarrow, x = 0, y = 0\rangle$$

over

$$w_2 = \langle power, 1.3\rangle \, \langle cook, 1.7\rangle \, \langle \lambda, 2.7\rangle \, \langle power, 3.7\rangle$$

TA runs: example



Run $r_1 = \langle \downarrow, x = 0, y = 0 \rangle \, \langle \uparrow, x = 3.0, y = 3.0 \rangle$
over $w_1 = \langle \mathit{power}, 3.0 \rangle$
is not accepting.

Run

$r_2 = \langle \downarrow, x = 0, y = 0 \rangle \, \langle \uparrow, x = 1.3, y = 1.3 \rangle$
$\quad \langle ϟ, x = 0, y = 1.7 \rangle \, \langle \uparrow, x = 1, y = 0 \rangle \, \langle \downarrow, x = 0, y = 0 \rangle$

over

$w_2 = \langle \mathit{power}, 1.3 \rangle \, \langle \mathit{cook}, 1.7 \rangle \, \langle \lambda, 2.7 \rangle \, \langle \mathit{power}, 3.7 \rangle$
is accepting.

Timed automata: semantics

The language of a timed automaton $A = \langle \Sigma, S, I, F, C, E \rangle$ is the set of timed words that A accepts:

$$\mathcal{L}(A) = \{w \in (\Sigma \times \mathbb{R}_{\geq 0})^* \mid \text{there is an accepting run of } A \text{ over } w\}$$

Timed automata: semantics

The language of a timed automaton $A = \langle \Sigma, S, I, F, C, E \rangle$ is the set of timed words that A accepts:

$$\mathcal{L}(A) = \{ w \in (\Sigma \times \mathbb{R}_{\geq 0})^* \mid \text{there is an accepting run of } A \text{ over } w \}$$

The emptiness problem is the fundamental decision problem:

The emptiness problem: given an automaton A, determine if it accepts any words – that is if A's language is empty.

Timed automata: semantics

The language of a timed automaton $A = \langle \Sigma, S, I, F, C, E \rangle$ is the set of timed words that A accepts:

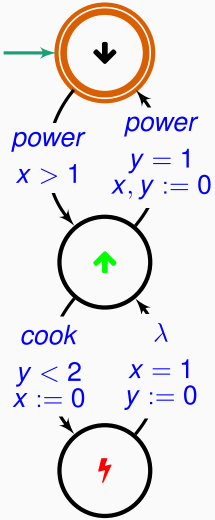
$$\mathcal{L}(A) = \{w \in (\Sigma \times \mathbb{R}_{\geq 0})^* \mid \text{there is an accepting run of } A \text{ over } w\}$$

The emptiness problem is the fundamental decision problem:

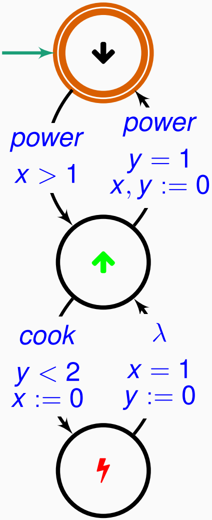
The emptiness problem: given an automaton A, determine if it accepts any words – that is if A's language is empty.

Note that even though TAs have a finite number of locations, their extended states (including clock values) may be infinite. Therefore, checking emptiness is not as straightforward as for FSAs.

TA semantics: example

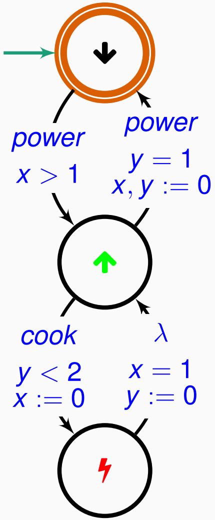


TA semantics: example



The language of the microwave timed automaton is not empty.

TA semantics: example



The language of the microwave timed automaton is not empty.

Words in the automaton's language include:

- ϵ
- w_2
- ...

Real-time model checking

Metric temporal logic

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box\left(\mathit{power} \Longrightarrow \Box_{\leq 1}\, \neg\mathit{power}\right)$$

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box\left(\mathit{power} \Longrightarrow \Box_{\leq 1} \neg \mathit{power}\right)$$

Every cooking session lasts exactly one time unit:

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box \left(\mathit{power} \Longrightarrow \Box_{\leq 1} \neg \mathit{power}\right)$$

Every cooking session lasts exactly one time unit:

$$\Box \left(\mathit{cook} \Longrightarrow \Box_{(0,1)} \bot \;\wedge\; \Diamond_{=1} \top\right)$$

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box\left(\mathit{power} \Longrightarrow \Box_{\leq 1} \neg \mathit{power}\right)$$

Every cooking session lasts exactly one time unit:

$$\Box\left(\mathit{cook} \Longrightarrow \Box_{(0,1)} \bot \;\wedge\; \Diamond_{=1} \top\right)$$

There exist two pressings of *cook* separated by exactly one time unit:

Metric temporal logic: example

We can model properties of the timed microwave oven using MTL formulas.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box\left(\mathit{power} \Longrightarrow \Box_{\leq 1} \neg \mathit{power}\right)$$

Every cooking session lasts exactly one time unit:

$$\Box\left(\mathit{cook} \Longrightarrow \Box_{(0,1)} \perp \;\wedge\; \Diamond_{=1} \top\right)$$

There exist two pressings of *cook* separated by exactly one time unit:

$$\Diamond\left(\mathit{cook} \wedge \Diamond_{=1}\, \mathit{cook}\right)$$

Metric temporal logic: syntax

Formulas of propositional metric temporal logic (MTL) are defined as:

$$
\begin{aligned}
F ::= \;& p \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 && \text{(propositional connectives)} \\
\mid \;& \mathsf{X}\, F \mid \Box_J\, F \mid \Diamond_J\, F \mid F_1\, \mathsf{U}_J\, F_2 && \text{(metric temporal operators)}
\end{aligned}
$$

where $p \in \Pi$ is any proposition from a set Π,
and $J \subseteq \mathbb{R}_{\geq 0}$ is any interval of the reals with integer endpoints.

Metric temporal logic: syntax

Formulas of propositional metric temporal logic (MTL) are defined as:

$$F ::= p \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 \quad \text{(propositional connectives)}$$
$$\mid \ \mathsf{X}\, F \mid \Box_J F \mid \Diamond_J F \mid F_1 \,\mathsf{U}_J\, F_2 \quad \text{(metric temporal operators)}$$

where $p \in \Pi$ is any proposition from a set Π,
and $J \subseteq \mathbb{R}_{\geq 0}$ is any interval of the reals with integer endpoints.

adds real-time constraint to LTL operator

Metric temporal logic: syntax

Formulas of propositional metric temporal logic (MTL) are defined as:

$$F ::= p \mid \neg F \mid F_1 \wedge F_2 \mid F_1 \vee F_2 \mid F_1 \Longrightarrow F_2 \qquad \text{(propositional connectives)}$$

$$\mid \mathsf{X}\, F \mid \Box_J\, F \mid \Diamond_J\, F \mid F_1\, \mathsf{U}_J\, F_2 \qquad \text{(metric temporal operators)}$$

where $p \in \Pi$ is any proposition from a set Π,
and $J \subseteq \mathbb{R}_{\geq 0}$ is any interval of the reals with integer endpoints.

Intervals can be:

open: $(3, 7)$ denotes all reals t such that $3 < t < 7$
closed: $[2, 5]$ denotes all reals t such that $2 \leq t \leq 5$
half-open: $[0, 3)$ denotes all reals t such that $0 \leq t < 3$
unbounded: $[1, \infty)$ denotes all reals t such that $1 \leq t$
pointwise: $[3, 3]$ denotes only the real $t = 3$

We often use abbreviations for special intervals:

- ≥ 3 for $[3, \infty)$, < 4 for $[0, 4)$, $= 7$ for $[7, 7]$
- omitting J stands for the whole time domain $[0, \infty)$

MTL syntax: examples

The examples we have seen before are MTL formulas over propositions in $\Pi = \{\mathit{power}, \mathit{cook}, \lambda\}$.

MTL syntax: examples

The examples we have seen before are MTL formulas over propositions in $\Pi = \{\mathit{power}, \mathit{cook}, \lambda\}$.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

MTL syntax: examples

The examples we have seen before are MTL formulas over propositions in $\Pi = \{\mathit{power}, \mathit{cook}, \lambda\}$.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box\left(\mathit{power} \Longrightarrow \Box_{\leq 1}\, \neg\mathit{power}\right)$$

MTL syntax: examples

The examples we have seen before are MTL formulas over propositions in $\Pi = \{\mathit{power}, \mathit{cook}, \lambda\}$.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box\left(\mathit{power} \Longrightarrow \Box_{\leq 1} \neg \mathit{power}\right)$$

Every cooking session lasts exactly one time unit:

$$\Box\left(\mathit{cook} \Longrightarrow \Box_{(0,1)} \bot \;\wedge\; \Diamond_{=1} \top\right)$$

MTL syntax: examples

The examples we have seen before are MTL formulas over propositions in $\Pi = \{\mathit{power}, \mathit{cook}, \lambda\}$.

After turning the oven on initially, cooking starts within two time units:

$$\mathit{power} \Longrightarrow \Diamond_{<2}\, \mathit{cook}$$

When we push *power*, we have to wait more than one time unit to push it again:

$$\Box\left(\mathit{power} \Longrightarrow \Box_{\leq 1} \neg \mathit{power}\right)$$

Every cooking session lasts exactly one time unit:

$$\Box\left(\mathit{cook} \Longrightarrow \Box_{(0,1)} \bot \ \wedge\ \Diamond_{=1} \top\right)$$

There exist two pressings of *cook* separated by exactly one time unit:

$$\Diamond\left(\mathit{cook} \wedge \Diamond_{=1}\, \mathit{cook}\right)$$

Metric temporal logic: satisfaction relation

A timed word $w \in (\Pi \times \mathbb{R}_{\geq 0})^*$

$$w = \; w[1]\, w[2] \;\ldots\; w[n] = \langle \sigma[1], t[1] \rangle \, \langle \sigma[2], t[2] \rangle \;\ldots\; \langle \sigma[n], t[n] \rangle$$

satisfies MTL formula F at position $1 \leq k \leq n$, written $w, k \models F$, iff:

Metric temporal logic: satisfaction relation

A timed word $w \in (\Pi \times \mathbb{R}_{\geq 0})^*$

$$w = w[1]\, w[2] \ldots w[n] = \langle \sigma[1], t[1] \rangle \langle \sigma[2], t[2] \rangle \ldots \langle \sigma[n], t[n] \rangle$$

satisfies MTL formula F at position $1 \leq k \leq n$, written $w, k \models F$, iff:

$$\begin{array}{rcl}
w, k \models p & \text{iff} & p = \sigma[k] \\
w, k \models \neg F & \text{iff} & w, k \not\models F \\
w, k \models F_1 \wedge F_2 & \text{iff} & w, k \models F_1 \text{ and } w, k \models F_2 \\
w, k \models F_1 \vee F_2 & \text{iff} & w, k \models F_1 \text{ or } w, k \models F_2 \\
w, k \models F_1 \Longrightarrow F_2 & \text{iff} & w, k \models \neg F_1 \vee F_2
\end{array}$$

Metric temporal logic: satisfaction relation

A timed word $w \in (\Pi \times \mathbb{R}_{\geq 0})^*$

$$w = \; w[1]\, w[2] \;\ldots\; w[n] = \langle \sigma[1], t[1] \rangle \, \langle \sigma[2], t[2] \rangle \;\ldots\; \langle \sigma[n], t[n] \rangle$$

satisfies MTL formula F at position $1 \leq k \leq n$, written $w, k \models F$, iff:

$$
\begin{array}{rcl}
w, k \models \mathsf{X}\, F & \text{iff} & k < n \text{ and } w, k+1 \models F \\
w, k \models \square_J\, F & \text{iff} & \text{for all } k \leq h \leq n\text{:} \\
 & & \text{if } t[h] - t[k] \in J \text{ then } w, h \models F \\
w, k \models \Diamond_J\, F & \text{iff} & \text{for some } k \leq h \leq n\text{:} \\
 & & t[h] - t[k] \in J \text{ and } w, h \models F \\
w, k \models F_1\, \mathsf{U}_J\, F_2 & \text{iff} & \text{for some } k \leq h \leq n\text{:} \\
 & & t[h] - t[k] \in J \text{ and } w, h \models F_2, \\
 & & \text{and, for all } k \leq j < h\text{: } w, j \models F_1
\end{array}
$$

MTL satisfaction: example

$$w \quad = \quad \underset{w[1]}{\langle power, 1.3\rangle} \quad \underset{w[2]}{\langle cook, 1.7\rangle} \quad \underset{w[3]}{\langle \lambda, 2.7\rangle} \quad \underset{w[4]}{\langle power, 3.7\rangle}$$

$w, 1 \models \Diamond_{<2}\, power$	$w, 3 \models \Diamond_{<2}\, power$	$w, 2 \not\models \Diamond_{<2}\, power$
$w, 1 \not\models \Box_{<2}\, power$	$w, 1 \models \Box_{(4,5)}\, power$	$w, 1 \models \Box_{(4,5)}\, \bot$
$w, 2 \models cook \wedge \Diamond_{=2}\, power$	$w, 2 \models cook \wedge \Box_{=2}\, power$	$w, 1 \models \Box\,(cook \implies \Diamond_{=2}\, power)$
$w, 1 \models power\ \mathsf{U}_{<1}\ cook$	$w, 1 \models \bot\ \mathsf{U}_{<1}\ power$	$w, 2 \models \mathsf{X}\,(\Diamond_{<3}\, power)$

Metric temporal logic: semantics

A timed word w satisfies an MTL formula F if it satisfies it initially:

$$w \models F \quad \text{iff} \quad w, 1 \models F$$

Metric temporal logic: semantics

A timed word w satisfies an MTL formula F if it satisfies it initially:

$$w \models F \quad \text{iff} \quad w, 1 \models F$$

The language of a metric temporal logic formula F is the set of timed words that satisfy F:

$$\mathcal{L}(F) = \{w \in (\Pi \times \mathbb{R}_{\geq 0})^* \mid w \models F\}$$

Real-time model checking

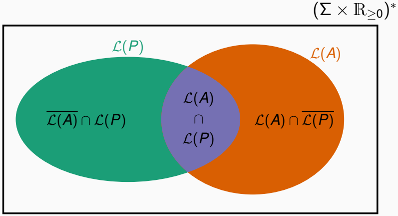
Real-time model-checking algorithm

Real-time model checking as emptiness checking

Real-time model checking: given

- A: a timed automaton with alphabet Σ
- P: a metric temporal logic property over propositions in Σ

$\mathcal{L}(A) \cap \mathcal{L}(\neg P)$ is empty	$\mathcal{L}(A) \cap \mathcal{L}(\neg P)$ is not empty
every timed word accepted by A satisfies P	some timed word accepted by A does not satisfy P
✔ $A \models P$	✖ $A \not\models P$
	every timed word in $\mathcal{L}(A) \cap \mathcal{L}(\neg P)$ is a counterexample



Real-time model-checking algorithm

To apply automata-based model checking to real-time verification, we need to extend its three algorithms to work with timed automata and metric temporal logic:

MONITOR: given a metric temporal logic formula P build a timed automaton $\mathcal{A}(P)$ such that $\mathcal{L}(\mathcal{A}(P)) = \mathcal{L}(P)$

INTERSECTION: given timed automata A and B, build a timed automaton $A \times B$ such that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$

EMPTINESS: given a timed automaton A determine whether $\mathcal{L}(A) = \emptyset$

Real-time model-checking algorithm

To apply automata-based model checking to real-time verification, we need to extend its three algorithms to work with timed automata and metric temporal logic:

MONITOR: given a metric temporal logic formula P build a timed automaton $\mathcal{A}(P)$ such that $\mathcal{L}(\mathcal{A}(P)) = \mathcal{L}(P)$

INTERSECTION: given timed automata A and B, build a timed automaton $A \times B$ such that $\mathcal{L}(A \times B) = \mathcal{L}(A) \cap \mathcal{L}(B)$

EMPTINESS: given a timed automaton A determine whether $\mathcal{L}(A) = \emptyset$

Things are more difficult with real-time:

- monitor: there exist MTL formulas that cannot be encoded as timed automata
- intersection works similarly as in the untimed case
- emptiness is still decidable but requires a significantly more complex algorithm

Monitors: from MTL to timed automata

There exist MTL formulas P such that it impossible to build a monitor of P: there exist no TA A that accepts precisely the timed words that satisfy P.

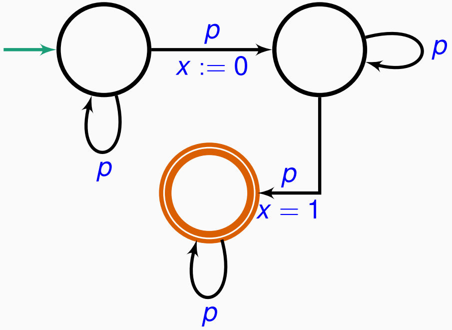
This is not a nonstarter for real-time model checking, since we can still build a monitor for interesting classes of MTL properties — but not all of them!

Expressiveness gap between MTL and timed automata

$O = \square\,(p \Longrightarrow \square_{=1}\,\neg p)$ no two p's are separated by one time unit

$\overline{O} = \diamond\,(p \wedge \diamond_{=1}\,p) = \neg O$ there exist two p's separated by one time unit

A timed automaton equivalent to $\overline{O}$:

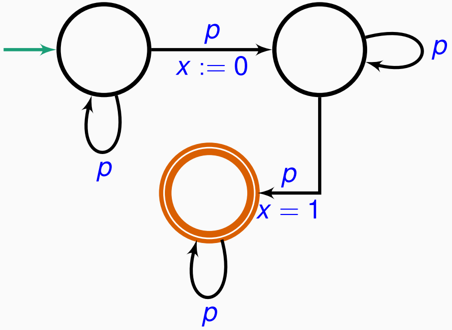


Expressiveness gap between MTL and timed automata

$O = \square\,(p \Longrightarrow \square_{=1}\,\neg p)$ — no two p's are separated by one time unit

$\overline{O} = \Diamond\,(p \wedge \Diamond_{=1}\,p) = \neg O$ — there exist two p's separated by one time unit

A timed automaton equivalent to $\overline{O}$:



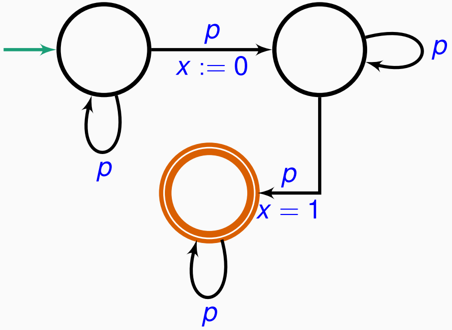
There is no timed automaton equivalent to O: we would need an infinite number of clocks to keep track of all pairs of p's – which can be arbitrarily close because the time domain is dense.

Expressiveness gap between MTL and timed automata

$O = \square\,(p \Longrightarrow \square_{=1} \neg p)$ no two p's are separated by one time unit

$\overline{O} = \diamond\,(p \wedge \diamond_{=1}\, p) = \neg O$ there exist two p's separated by one time unit

A timed automaton equivalent to $\overline{O}$:



There is no timed automaton equivalent to O: we would need an infinite number of clocks to keep track of all pairs of p's – which can be arbitrarily close because the time domain is dense.

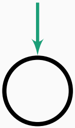
This also shows that timed automata are not closed under complement.

Monitors: bounded response

$P_1 = \Box\,(p \Longrightarrow \Diamond_{<2}\, q)$ p is followed by q within 2

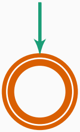
Monitors: bounded response

$P_1 = \square\,(p \Longrightarrow \Diamond_{<2}\, q)$ $\qquad$ p is followed by q within 2



Monitors: bounded response

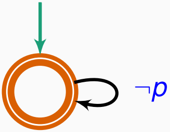
$$P_1 = \Box\,(p \Longrightarrow \Diamond_{<2}\, q) \qquad p \text{ is followed by } q \text{ within } 2$$



- the empty word satisfies P_1

Monitors: bounded response

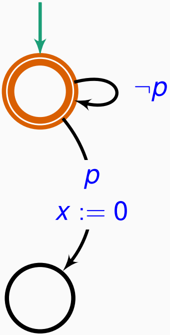
$$P_1 = \Box\,(p \Longrightarrow \Diamond_{<2}\, q)$$ p is followed by q within 2



- the empty word satisfies P_1
- as long as p does not occur, we accept

Monitors: bounded response

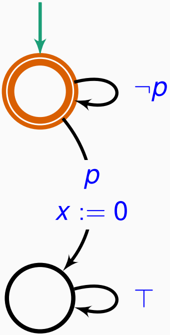
$$P_1 = \square\,(p \Longrightarrow \Diamond_{<2}\, q)$$ p is followed by q within 2



- the empty word satisfies P_1
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state

Monitors: bounded response

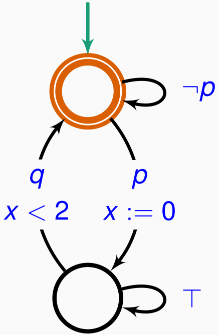
$$P_1 = \Box\,(p \Longrightarrow \Diamond_{<2}\, q) \qquad p \text{ is followed by } q \text{ within } 2$$



- the empty word satisfies P_1
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state
- any event may occur in the meanwhile (including more p's)

Monitors: bounded response

$$P_1 = \Box\,(p \Longrightarrow \Diamond_{<2}\, q) \qquad p \text{ is followed by } q \text{ within } 2$$



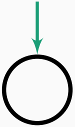
- the empty word satisfies P_1
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state
- any event may occur in the meanwhile (including more p's)
- a q must eventually occur while $x < 2$ ("covering" all p's that occurred since the last reset of x)

Monitors: bounded invariance

$$P_2 = \Box \left(p \Longrightarrow \Box_{(0,2]}\, q \right)$$ only q occur after p within 2

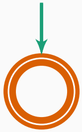
Monitors: bounded invariance

$$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$$ only q occur after p within 2



Monitors: bounded invariance

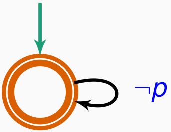
$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$ only q occur after p within 2



- the empty word satisfies P_2

Monitors: bounded invariance

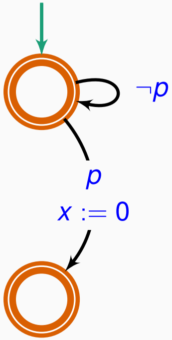
$$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$$ only q occur after p within 2



- the empty word satisfies P_2
- as long as p does not occur, we accept

Monitors: bounded invariance

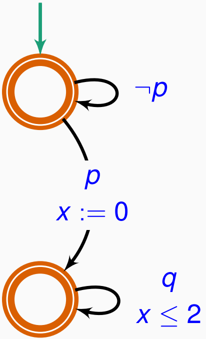
$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$ only q occur after p within 2



- the empty word satisfies P_2
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state

Monitors: bounded invariance

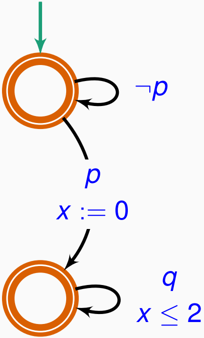
$$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$$ only q occur after p within 2



- the empty word satisfies P_2
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state
- if any event occurs within 2, it must be q

Monitors: bounded invariance

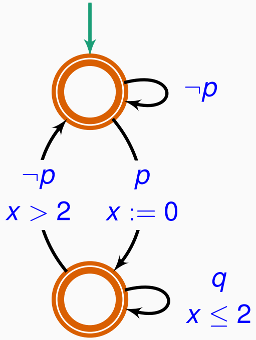
$$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$$ only q occur after p within 2



- the empty word satisfies P_2
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state
- if any event occurs within 2, it must be q
- if the timed word stops here, we also accept

Monitors: bounded invariance

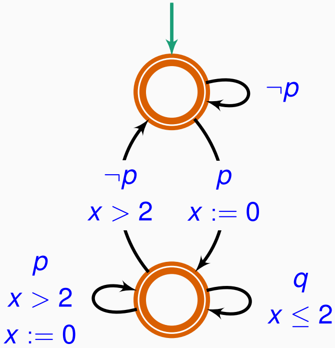
$$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$$ only q occur after p within 2



- the empty word satisfies P_2
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state
- if any event occurs within 2, it must be q
- if the timed word stops here, we also accept
- after 2 time units, we restart

Monitors: bounded invariance

$$P_2 = \Box\left(p \Longrightarrow \Box_{(0,2]}\, q\right)$$ only q occur after p within 2



- the empty word satisfies P_2
- as long as p does not occur, we accept
- when p occurs we reset clock x and move to a new state
- if any event occurs within 2, it must be q
- if the timed word stops here, we also accept
- after 2 time units, we restart
- unless we read p again, in which case we reset x

Intersection: running automata in parallel

A timed automaton C that accepts the intersection of two TAs A and B's languages runs A and B in parallel:

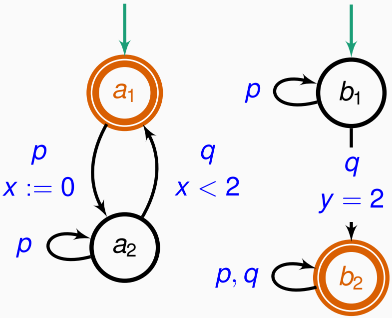
- starts from any combination of initial states of A and B
- transitions only when both A and B have a transition for the current input, with the same constraints and clock resets
- accepts when both A and B accept

Intersection: running automata in parallel

A timed automaton C that accepts the intersection of two TAs A and B's languages runs A and B in parallel:

- starts from any combination of initial states of A and B
- transitions only when both A and B have a transition for the current input, with the same constraints and clock resets
- accepts when both A and B accept

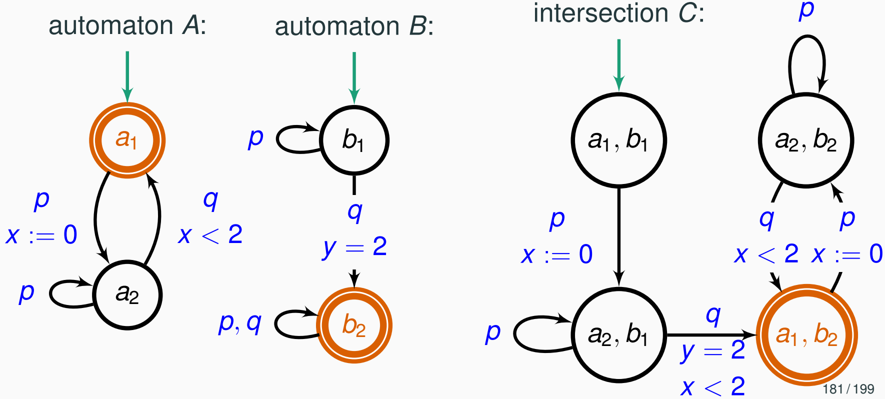
automaton A: automaton B:



Intersection: running automata in parallel

A timed automaton C that accepts the intersection of two TAs A and B's languages runs A and B in parallel:

- starts from any combination of initial states of A and B
- transitions only when both A and B have a transition for the current input, with the same constraints and clock resets
- accepts when both A and B accept



Product timed automaton construction

Given TAs $A = \langle \Sigma, S_A, I_A, F_A, C_A, E_A \rangle$ and $B = \langle \Sigma, S_B, I_B, F_B, C_B, E_B \rangle$, the product automaton $A \times B = \langle \Sigma, S, I, F, C, E \rangle$ is defined as:

$$
\begin{aligned}
S &= S_A \times S_B \\
I &= \{(a,b) \mid a \in I_A \text{ and } b \in I_B\} \\
F &= \{(a,b) \mid a \in F_A \text{ and } b \in F_B\} \\
C &= C_A \cup C_B \qquad \text{assuming } C_A \cap C_B = \emptyset \\
E &\ni ((a,b), \sigma, c_A \wedge c_B, \rho_A \cup \rho_B, (a_2, b_2)) \text{ iff} \\
&\qquad (a, \sigma, c_A, \rho_A, a_2) \in E_A \quad \text{and} \quad (b, \sigma, c_B, \rho_B, b_2) \in E_B
\end{aligned}
$$

Product timed automaton construction

Given TAs $A = \langle \Sigma, S_A, I_A, F_A, C_A, E_A \rangle$ and $B = \langle \Sigma, S_B, I_B, F_B, C_B, E_B \rangle$, the product automaton $A \times B = \langle \Sigma, S, I, F, C, E \rangle$ is defined as:

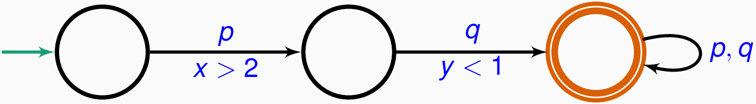
$$
\begin{aligned}
S &= S_A \times S_B \\
I &= \{(a, b) \mid a \in I_A \text{ and } b \in I_B\} \\
F &= \{(a, b) \mid a \in F_A \text{ and } b \in F_B\} \\
C &= C_A \cup C_B \qquad \text{assuming } C_A \cap C_B = \emptyset \\
E &\ni ((a, b), \sigma, c_A \wedge c_B, \rho_A \cup \rho_B, (a_2, b_2)) \text{ iff} \\
&\qquad (a, \sigma, c_A, \rho_A, a_2) \in E_A \quad \text{and} \quad (b, \sigma, c_B, \rho_B, b_2) \in E_B
\end{aligned}
$$

The language of the product automaton is the intersection of the intersected automata's languages:

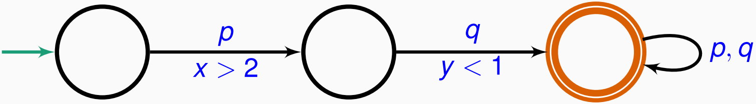
$$\mathcal{L}(A \times B) \;=\; \mathcal{L}(A) \cap \mathcal{L}(B)$$

Timed automata emptiness: overview

Emptiness of TAs is not just reachability: clocks introduce additional constraints that may be unsatisfiable. For example, the following TA is empty even if an accepting state is reachable on the graph.



Timed automata emptiness: overview



We present Alur and Dill's region automaton algorithm to determine the emptiness of TAs.

- a TA's run may traverse infinitely many extended states because clocks can take any value
- however, we may approximate the precise values of clocks by their region: an equivalence relation that only looks at the relative order of clocks compared to integer constants they are actually compared to in the automaton
- since the number of regions is always finite, we can build a finite-state automaton – the region automaton – whose language is empty iff the TA's language is empty

Equivalence between clock values

Consider a TA A with m clocks $x_1, \ldots, x_m$ such that M_m is the largest constant used in A's clock constraints involving x_m.

A clock evaluation is any tuple $(t_1, \ldots, t_m) \in \mathbb{R}^m_{\geq 0}$ of clock values, which is part of A's extended state.

Two clock evaluations $t = (t_1, \ldots, t_m)$ and $u = (u_1, \ldots, u_m)$ are equivalent (written $t \sim u$) iff all the following conditions hold:

1. corresponding clocks have the same integer value:
   for all $1 \leq k \leq m$: $\mathit{int}(t_k) = \mathit{int}(u_k)$ or $t_k, u_k > M_k$
2. corresponding clock pairs have the same order of fractional values:
   for all $1 \leq j \leq k \leq m$ such that $t_j \leq M_j$ and $t_k \leq M_k$:
   $\mathit{frac}(t_j) \leq \mathit{frac}(t_k)$ iff $\mathit{frac}(u_j) \leq \mathit{frac}(u_k)$
3. corresponding clocks agree on having integer or fractional value:
   for all $1 \leq k \leq m$ such that $t_k \leq M_j$: $\mathit{frac}(t_k) = 0$ iff $\mathit{frac}(u_k) = 0$

Here $\mathit{int}(x)$ is the integer part of x; $\mathit{frac}(x)$ is its fractional part.
For example: $\mathit{int}(3.1412) = 3$, $\mathit{frac}(3.1412) = 0.1412$.

Equivalence between clock values

Two clock evaluations $t = (t_1, \ldots, t_m)$ and $u = (u_1, \ldots, u_m)$ are equivalent (written $t \sim u$) iff all the following conditions hold:

1. corresponding clocks have the same integer value:
   for all $1 \leq k \leq m$: $\mathit{int}(t_k) = \mathit{int}(u_k)$ or $t_k, u_k > M_k$
2. corresponding clock pairs have the same order of fractional values:
   for all $1 \leq j \leq k \leq m$ such that $t_j \leq M_j$ and $t_k \leq M_k$:
   $\mathit{frac}(t_j) \leq \mathit{frac}(t_k)$ iff $\mathit{frac}(u_j) \leq \mathit{frac}(u_k)$
3. corresponding clocks agree on having integer or fractional value:
   for all $1 \leq k \leq m$ such that $t_k \leq M_j$: $\mathit{frac}(t_k) = 0$ iff $\mathit{frac}(u_k) = 0$

t_1, t_2	u_1, u_2	M_1	M_2	$t \sim u$?
$0.3, 2.2$	$0.4, 2.7$	3	3	✔
$0.3, 2.0$	$0.5, 4.9$	1	5	✖
$0.3, 2.0$	$0.5, 4.9$	1	1	✔
$1.0, 1.6$	$1.0, 1.8$	2	2	✔

Equivalence between clock values

1. corresponding clocks have the same integer value:
   for all $1 \leq k \leq m$: $\mathit{int}(t_k) = \mathit{int}(u_k)$ or $t_k, u_k > M_k$
2. corresponding clock pairs have the same order of fractional values:
   for all $1 \leq j \leq k \leq m$ such that $t_j \leq M_j$ and $t_k \leq M_k$:
   $\mathit{frac}(t_j) \leq \mathit{frac}(t_k)$ iff $\mathit{frac}(u_j) \leq \mathit{frac}(u_k)$
3. corresponding clocks agree on having integer or fractional value:
   for all $1 \leq k \leq m$ such that $t_k \leq M_j$: $\mathit{frac}(t_k) = 0$ iff $\mathit{frac}(u_k) = 0$

Intuition behind the equivalence: A only compares clocks to integer constants, and all clocks proceed at the same rate; hence, we only need to keep track of

- condition 1: whether a clock is less than, equal to (condition 3), or greater than an integer constant
- condition 2: which clock will reach an integer value next
- all conditions: when clock x_k is greater than M_k its exact value does not matter until a reset

Regions

Clock regions are the equivalence classes induced by the equivalence relation $\sim$ between clock evaluations.

$\|t\|$ denotes the region clock evaluation t belongs to.

Regions

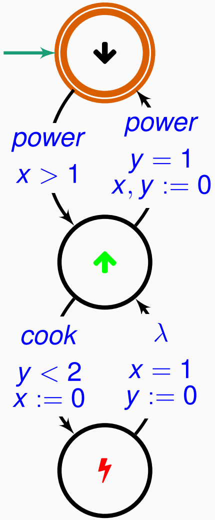
Clock regions are the equivalence classes induced by the equivalence relation $\sim$ between clock evaluations.

$\|t\|$ denotes the region clock evaluation t belongs to.

In practice, we keep track of which integers each clock is between, and the relative order between clocks (including if they are equal).

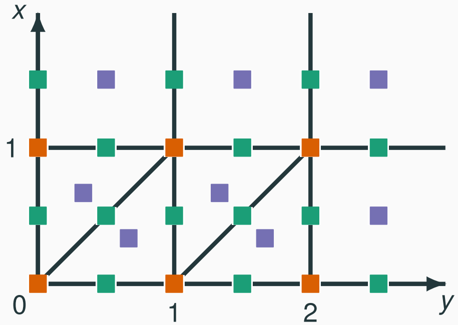
t_1, t_2	u_1, u_2	M_1	M_2	$t \sim u$?	$\|t\|$	$\|u\|$
$0.3, 2.2$	$0.4, 2.7$	3	3	✔	$0 < x_1 < 1$ $2 < x_2 < 3$	$0 < x_1 < 1$ $2 < x_2 < 3$
$0.3, 2.0$	$0.5, 4.9$	1	5	✘	$0 < x_1 < 1$ $x_2 = 2$	$0 < x_1 < 1$ $4 < x_2 < 5$
$0.3, 2.0$	$0.5, 4.9$	1	1	✔	$0 < x_1 < 1$ $x_2 > 1$	$0 < x_1 < 1$ $x_2 > 1$
$1.0, 1.6$	$1.0, 1.8$	2	2	✔	$x_1 = 1$ $1 < x_2 < 2$	$x_1 = 1$ $1 < x_2 < 2$

Clock regions: example



The microwave TA determines 28 possible regions:

- 8 open regions
- 14 open line segments
- 6 corner points



Time successors

The time successors $\overset{\rightsquigarrow}{r}$ of a region r are all regions (including r) that can be reached from r by letting time pass.

M_1	M_2	r	$\overset{\rightsquigarrow}{r}$
1	1	$0 < x_1 < x_2 < 1$	$0 < x_1 < x_2 < 1$ $0 < x_1 < 1 = x_2$ $x_1 = 1 < x_2$ $x_1, x_2 > 1$
2	2	$0 < x_1 < 1, x_2 = 2$	$0 < x_1 < 1, x_2 = 2$ $x_1 = 1, x_2 > 2$ $1 < x_1 < 2 < x_2$ $x_1 = 2 < x_2, x_2 > 2$
1	1	$0 < x_1 = x_2 < 1$	$0 < x_1 = x_2 < 1$ $x_1 = x_2 = 1$ $x_1, x_2 > 1$

Region automaton

The region automaton $\|A\|$ of timed automaton $A = \langle \Sigma, S, I, F, C, E\rangle$ is the finite-state automaton $\langle \Sigma, S_R, I_R, F_R, \rho_R\rangle$ defined as:

- $S_R = S \times R$, where R is the set of all possible regions determined by A
- $I_R = \{(s, \|(0, 0, \ldots, 0)\|) \mid s \in I\}$, corresponding to initial locations with all clocks reset to zero
- $F_R = \{(s, r \mid s \in F\}$ corresponding to final locations
- $\rho_R(\sigma, (s, r)) = \left\{ (s_2, r_2) \;\middle|\; \begin{array}{l} (s, \sigma, c, \rho, s_2) \in E, \text{ there exists} \\ r' \in \overset{\leadsto}{r} \text{ such that } r' \text{ satisfies } c, \\ \text{and } r_2 = r'[x \mapsto 0 \mid x \in \rho] \text{ is} \\ r' \text{ with all clocks } \rho \text{ reset to zero} \end{array} \right\}$

Region automaton

The region automaton $\|A\|$ of timed automaton $A = \langle \Sigma, S, I, F, C, E \rangle$ is the finite-state automaton $\langle \Sigma, S_R, I_R, F_R, \rho_R \rangle$ defined as:

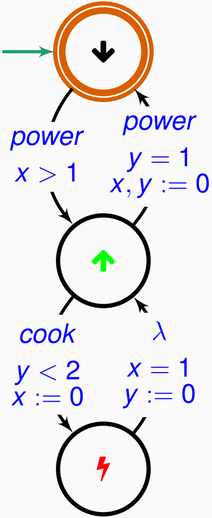
- $S_R = S \times R$, where R is the set of all possible regions determined by A
- $I_R = \{(s, \|(0, 0, \ldots, 0)\|) \mid s \in I\}$, corresponding to initial locations with all clocks reset to zero
- $F_R = \{(s, r \mid s \in F\}$ corresponding to final locations
- $\rho_R(\sigma, (s, r)) = \left\{ (s_2, r_2) \;\middle|\; \begin{array}{l} (s, \sigma, c, \rho, s_2) \in E, \text{ there exists} \\ r' \in \overset{\rightsquigarrow}{r} \text{ such that } r' \text{ satisfies } c, \\ \text{and } r_2 = r'[x \mapsto 0 \mid x \in \rho] \text{ is} \\ r' \text{ with all clocks } \rho \text{ reset to zero} \end{array} \right\}$

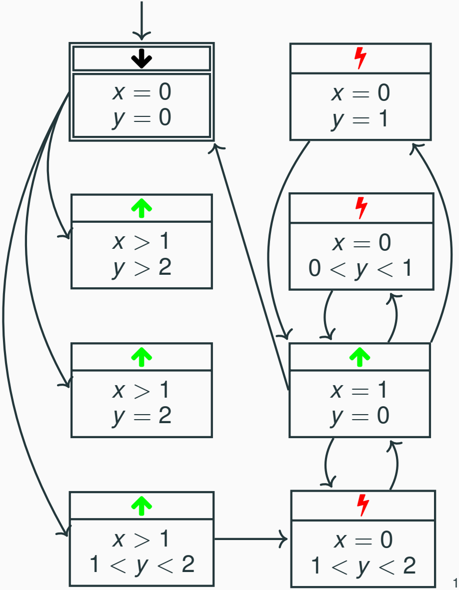
Emptiness: $\mathcal{L}(\|A\|) = \emptyset$ if and only if $\mathcal{L}(A) = \emptyset$

Since $\|A\|$ is a finite-state automaton, emptiness of a TA A is decidable by checking reachability on its region automaton $\|A\|$.

Region automaton: example

The region automaton of the running example's TA is not empty:





Real-time model checking

Complexity, variants, and tools

Complexity of the real-time model-checking algorithm

Real-time model-checking algorithm: given a timed automaton A and a metric temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

Complexity of the real-time model-checking algorithm

Real-time model-checking algorithm: given a timed automaton A and a metric temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

MONITOR construction for an MTL formula F is not always possible (for subsets of MTL it typically produces an automaton $\mathcal{A}(F)$ of size not larger than $2^{2^{O(|F|)}}$)

Complexity of the real-time model-checking algorithm

Real-time model-checking algorithm: given a timed automaton A and a metric temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

MONITOR construction for an MTL formula F is not always possible (for subsets of MTL it typically produces an automaton $\mathcal{A}(F)$ of size not larger than $2^{2^{\mathrm{O}(|F|)}}$)

INTERSECTION of two timed automata $|A|$ and $|B|$ creates a timed automaton of size $\mathrm{O}(|A| \cdot |B|)$

Complexity of the real-time model-checking algorithm

Real-time model-checking algorithm: given a timed automaton A and a metric temporal logic property P:

1. monitor: build $\mathcal{A}(\neg P)$
2. intersection: build $A \times \mathcal{A}(\neg P)$
3. emptiness: test whether $\mathcal{L}(A \times \mathcal{A}(\neg P)) = \emptyset$

Let's analyze the complexity of each step in the algorithm.

MONITOR construction for an MTL formula F is not always possible (for subsets of MTL it typically produces an automaton $\mathcal{A}(F)$ of size not larger than $2^{2^{\mathrm{O}(|F|)}}$)

INTERSECTION of two timed automata $|A|$ and $|B|$ creates a timed automaton of size $\mathrm{O}(|A| \cdot |B|)$

EMPTINESS of a timed automaton A is linear in the size of $\|A\|$, which has $2^{\mathrm{O}(|A|)}$ states (in particular, the number of regions is exponential in the number of clocks)

Complexity of the timed automata emptiness problem

The emptiness problem for timed automata is PSPACE-complete.

Therefore, the region-automaton construction is worst-case optimal.

There are algorithms that perform better in practice – based on more compact symbolic representations of regions (such as zones).

Complexity of the real-time model-checking problem

What is the worst-case complexity of the real-time model-checking problem as a whole?

The model-checking problem for timed automata and MTL formulas is:

- nonprimitive recursive for dense time domains and finite words
- undecidable for dense time domains and infinite words
- EXPSPACE-complete for discrete time

Complexity of the real-time model-checking problem

What is the worst-case complexity of the real-time model-checking problem as a whole?

The model-checking problem for timed automata and MTL formulas is:

- nonprimitive recursive for dense time domains and finite words
- undecidable for dense time domains and infinite words
- EXPSPACE-complete for discrete time

Dense time and finite words is the interpretation we have presented in this class. Nonprimitive recursive means that it is decidable but "barely so".

This result also implies that there is a way of performing real-time model checking over finite words that does not require to build timed automata monitors of MTL (since this is not always possible).

Complexity of the real-time model-checking problem

What is the worst-case complexity of the real-time model-checking problem as a whole?

The model-checking problem for timed automata and MTL formulas is:

- nonprimitive recursive for dense time domains and finite words
- undecidable for dense time domains and infinite words
- EXPSPACE-complete for discrete time

Dense time and finite words is the interpretation we have presented in this class. Nonprimitive recursive means that it is decidable but "barely so".

This result also implies that there is a way of performing real-time model checking over finite words that does not require to build timed automata monitors of MTL (since this is not always possible).

In contrast, if we stick with discrete time, timed automata and MTL are just exponentially more succinct versions of finite-state automata and LTL – and hence fully decidable.

Real-time and hybrid model checking tools

Model checkers and analyzers for real-time and hybrid models include:

UPPAAL is a verification system for timed automata – the first practical tool for such analysis – which has been extended with support for hybrid models and analysis that go beyond model checking (for example, synthesis)

PAT is an extensible model checking framework, including functionalities for real-time models and analysis as well as more traditional automata-based model checking

Prism is a model checker for stochastic models, which also support a notion of timed behavior

FDR4 is a modern reimplementation of the classic FDR2 model checker for CSPs (Communicating Sequential Processes) and timed CSPs

Summary

Case studies and industrial usage

Model checking has been the first formal verification technique to gain significant industrial adoption.

Case studies and industrial usage

Model checking has been the first formal verification technique to gain significant industrial adoption.

The hardware industry – especially big companies such as Intel – has been using, for over 20 years, model checking tools to verify parts of circuit logic. Since electronic circuits are intrinsically finite state, but with lots of parallelism, model checking is an ideal technique since it is sound and complete.

Case studies and industrial usage

Model checking has been the first formal verification technique to gain significant industrial adoption.

Adoption of model checking for software has been more selective and context dependent:

- NASA has been using the SPIN model checker to detect concurrency bugs, as part of a more general methodology to develop reliable control software
- Microsoft used the Slam model checker (the first implementation of CEGAR model checking) to verify memory safety of Windows device drivers

Case studies and industrial usage

Model checking has been the first formal verification technique to gain significant industrial adoption.

Real-time model checking is still not widespread – mainly due to the high complexity of the problem – but some of its techniques have become part of modern control theory, and hence connected to practice that way.

Case studies and industrial usage

Model checking has been the first formal verification technique to gain significant industrial adoption.

On a different level, model checking's success has helped make all of formal verification technology more appealing and practically useful.

Model checking: techniques

Model checking denotes analysis techniques for finite-state models and properties expressed in some form of temporal logic.

The classic automata-based techniques are based on reachability on finite (but possibly very large) graphs.

soundness/completeness: sound and complete with respect to the finite-state model, which typically is an under-approximation of an infinite-state system

complexity: complex (exponential and more) but decidable

automation: fully automated ("push button")

expressiveness: arbitrary properties expressible in temporal logic – such as event ordering, reachability, termination

Model checking: tools and practice

Model checking tools have been highly optimized so that they scale to systems with billions of states. Model checkers can return an explicit counterexample when verification fails; this significantly helps usability.

Case studies of model checking include an extensive usage in hardware verification and concurrency debugging. Software model checking has been applied to system code such as device drivers.

Main outstanding challenges:

- further improving scalability (fighting state-space explosion)
- in predicate abstraction: supporting advanced program features and inter-procedural analysis
- in real-time model checking: finding expressive yet tractable models

Credits and references

The presentation of automata-theoretic model checking revisits Vardi's classic presentation (Automata-theoretic model checking revisited, VMCAI, 2007; Automata-theoretic techniques for temporal reasoning) using finite words.

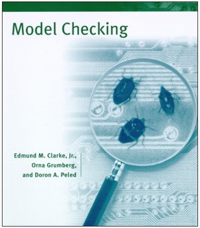
Parts of the presentation of model checking and real-time model checking are based on chapters in Furia et al.: Modeling time in computing, Springer, 2012.

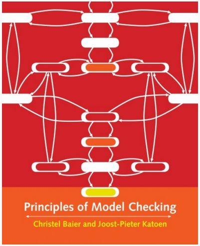
The presentation of Boolean programs and predicate abstraction is derived from Ball and Rajamani's for the software model checker SLAM.

The region automaton construction follows the original in Alur and Dill: A theory of timed automata, TCS, 1994.

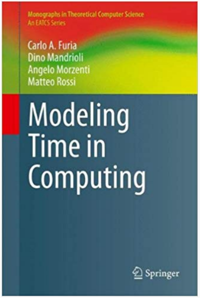
Further reading

For a comprehensive presentation of model checking see one of these two textbooks:





For more details on real-time models and verification techniques: Modeling time in computing.



These slides' license