Today
Today

Last lecture

• Symbolic execution and concolic testing
Today

Last lecture
  • Symbolic execution and concolic testing

Today
  • Introduction to model checking
Today

Last lecture
  • Symbolic execution and concolic testing

Today
  • Introduction to model checking

Reminders
  • Homework 3 is due on Tuesday, November 18, at 11pm
You are already halfway through your final project, right?

Today

Last lecture
  • Symbolic execution and concolic testing

Today
  • Introduction to model checking

Reminders
  • Homework 3 is due on Tuesday, November 18, at 11pm
What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

$M, s \models P$
What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

A mathematical model of the system, given as a Kripke structure (a finite state machine).

\[ M, s \models P \]
What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

\[ M, s \models P \]

A state of the system (e.g., an initial state).

A mathematical model of the system, given as a Kripke structure (a finite state machine).
What is model checking?

An automated technique for verifying that a concurrent finite state system satisfies a given temporal property.

A state of the system (e.g., an initial state).

A temporal logic formula (e.g., a request is eventually acknowledged).

A mathematical model of the system, given as a Kripke structure (a finite state machine).

\[ M, s \models P \]
Why model checking?

Model checking

Classic & bounded verification
Why model checking?

**Model checking**

**Classic & bounded verification**

- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).
Why model checking?

Model checking

- *Reactive systems*: concurrent finite-state programs with ongoing input/output behavior.
- *Control-intensive* but without a lot of data manipulation.
- Fully automatic checking of properties in less expressive (temporal) logics.

Classic & bounded verification

- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).
Why model checking?

Model checking

- *Reactive systems*: concurrent finite-state programs with *ongoing* input/output behavior.
- *Control-intensive* but without a lot of data manipulation.
- Fully automatic checking of properties in less expressive (temporal) logics.

- Microprocessors and device drivers
- Embedded controllers (e.g., cars, planes)
- Protocols (e.g., cache coherence)

Classic & bounded verification

- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).
# Why model checking?

## Model checking
- **Reactive systems**: concurrent finite-state programs with ongoing input/output behavior.
- **Control-intensive** but without a lot of data manipulation.
- Fully automatic checking of properties in less expressive (temporal) logics.
- Microprocessors and device drivers
- Embedded controllers (e.g., cars, planes)
- Protocols (e.g., cache coherence)

## Classic & bounded verification
- Deterministic, single-threaded, possibly infinite-state, terminating programs.
- Fully described by their input/output behavior.
- Semi-automatic or bounded-automatic checking of properties in expressive logics (e.g., FOL).
- Libraries and ADT implementations
- Heap-manipulating programs (e.g., OO)
- Tricky deterministic algorithms
A brief history of model checking
Modern modal logic (Lewis).
A brief history of model checking

Modern modal logic (Lewis).

Standard semantics for modal logics (Kripke).
Temporal logic (Prior).
1977: Using LTL to reason about concurrent programs (Pnueli).
1985: Automata-theoretic approach for LTL model checking (Vardi & Wolper).

Modern modal logic (Lewis).
Standard semantics for modal logics (Kripke).
Temporal logic (Prior).
A brief history of model checking

1930: Modern modal logic (Lewis).

1960: Standard semantics for modal logics (Kripke).

1977: Using LTL to reason about concurrent programs (Pnueli).


1985: Automata-theoretic approach for LTL model checking (Vardi & Wolper).


1989: SPIN (Holzmann)

1992: SMV (McMillan)

1994: Pentium bug

1995: Futurebus+ verified

1990: 1990
A brief history of model checking

1996: Pnueli wins the Turing award “for seminal work introducing temporal logic into computing science and for outstanding contributions to program and system verification.”

2007: Clarke, Emerson and Sifakis jointly win the Turing award “for their role in developing Model-Checking into a highly effective verification technology that is widely adopted in the hardware and software industries.”
Kripke structures
Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$
Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$

- $S$ is a finite set of states.
Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$

- $S$ is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
**A Kripke structure is a tuple** \( M = \langle S, S_0, R, L \rangle \)

- \( S \) is a finite set of states.
- \( S_0 \subseteq S \) is the set of initial states.
- \( R \subseteq S \times S \) is the transition relation, which must be total.
Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$

- $S$ is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the transition relation, which must be total.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state.
Kripke structures

A Kripke structure is a tuple $M = \langle S, S_0, R, L \rangle$

- $S$ is a finite set of states.
- $S_0 \subseteq S$ is the set of initial states.
- $R \subseteq S \times S$ is the transition relation, which must be total.
- $L : S \rightarrow 2^{AP}$ is a function that labels each state with a set of atomic propositions true in that state.

A path in $M$ is an infinite sequence of states $\pi = s_0s_1\ldots$ such that for all $i \geq 0$, $(s_i, s_{i+1}) \in R$. 
Modeling systems with Kripke structures

// x==1, y==1
x := (x + y) % 2

• In a finite-state program, system variables V range over a finite domain D: V = \{x, y\} and D = \{0, 1\}.
• A state of the system is a valuation s : V \rightarrow D.
Modeling systems with Kripke structures

\[
\begin{align*}
S &\equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1) \\
S_0 &\equiv (x = 1) \land (y = 1) \\
R(x, y, x', y') &\equiv (x' = (x + y) \mod 2) \land (y' = y)
\end{align*}
\]

// x==1, y==1
x := (x + y) % 2

• In a finite-state program, system variables \( V \) range over a finite domain \( D: V = \{x, y\} \) and \( D = \{0, 1\} \).

• A state of the system is a valuation \( s : V \to D \).

• Use FOL to describe the (initial) states and the transition relation.
Modeling systems with Kripke structures

// x==1, y==1
x := (x + y) % 2

S ≡ (x = 0 ∨ x = 1) ∧ (y = 0 ∨ y = 1)
S₀ ≡ (x = 1) ∧ (y = 1)
R(x, y, x′, y′) ≡ (x′ = (x + y) % 2) ∧ (y′ = y)

• In a finite-state program, system variables V range over a finite domain D: V = {x, y} and D = {0, 1}.
• A state of the system is a valuation s : V → D.
• Use FOL to describe the (initial) states and the transition relation.
• Extract a Kripke structure from the FOL description.
Modeling systems with Kripke structures

// x==1, y==1
x := (x + y) % 2

$S \equiv (x = 0 \lor x = 1) \land (y = 0 \lor y = 1)$

$S_0 \equiv (x = 1) \land (y = 1)$

$R(x, y, x', y') \equiv (x' = (x + y) \% 2) \land (y' = y)$

- In a finite-state program, system variables $V$ range over a finite domain $D$: $V = \{x, y\}$ and $D = \{0, 1\}$.
- A state of the system is a valuation $s : V \to D$.
- Use FOL to describe the (initial) states and the transition relation.
- Extract a Kripke structure from the FOL description.
Modeling systems with Kripke structures

// x==1, y==1
x := (x + y) % 2

S ≡ (x = 0 ∨ x = 1) ∧ (y = 0 ∨ y = 1)
S₀ ≡ (x = 1) ∧ (y = 1)
R(x, y, x', y') = (x' = (x + y) % 2) ∧ (y' = y)

• In a finite-state program, system variables V range over a finite domain D: V = {x, y} and D = {0, 1}.
• A state of the system is a valuation s : V → D.
• Use FOL to describe the (initial) states and the transition relation.
• Extract a Kripke structure from the FOL description.
Modeling systems with Kripke structures

// x==1, y==1
x := (x + y) % 2

S ≡ (x = 0 ∨ x = 1) ∧ (y = 0 ∨ y = 1)
S₀ ≡ (x = 1) ∧ (y = 1)
R(x, y, x', y') ≡ (x' = (x + y) % 2) ∧ (y' = y)

In a finite-state program, system variables V range over a finite domain D: V = {x, y} and D = {0, 1}.

A state of the system is a valuation s : V → D.

Use FOL to describe the (initial) states and the transition relation.

Extract a Kripke structure from the FOL description.
Modeling systems with Kripke structures

In a finite-state program, system variables $V$ range over a finite domain $D$: $V = \{x, y\}$ and $D = \{0, 1\}$.

- A state of the system is a valuation $s : V \rightarrow D$.
- Use FOL to describe the (initial) states and the transition relation.
- Extract a Kripke structure from the FOL description.

State explosion: Kripke structure usually exponential in the size of the program.
A Kripke structure for a concurrent program

Two processes executing concurrently and asynchronously, using the shared variable $\text{turn}$ to ensure \textit{mutual exclusion}:

They are never in the critical section at the same time.

\begin{verbatim}
P_1
while (true) {
    wait(turn == 0);
    // critical section
    turn := 1;
}

P_2
while (true) {
    wait(turn == 1);
    // critical section
    turn := 0;
}
\end{verbatim}
A Kripke structure for a concurrent program

\[ P_1 \]
10 while (true) {
11 \hspace{1em} wait(turn == 0);
12 \hspace{1em} // critical section
13 \hspace{1em} turn := 1;
14 }

\[ P_2 \]
20 while (true) {
21 \hspace{1em} wait(turn == 1);
22 \hspace{1em} // critical section
23 \hspace{1em} turn := 0;
24 }

Two processes executing concurrently and asynchronously, using the shared variable turn to ensure *mutual exclusion*:

They are never in the critical section at the same time.

State of the program described by the variable turn and the *program counters* for the two processes.
A Kripke structure for a concurrent program

\[ P_1 \]
10 while (true) {
11 wait(turn == 0);
12 // critical section
13 turn := 1;
14 }

\[ P_2 \]
20 while (true) {
21 wait(turn == 1);
22 // critical section
23 turn := 0;
24 }
A Kripke structure for a concurrent program

\[ P_1 \]
10 while (true) {
11 \hspace{1em} wait(turn == 0);
12 \hspace{1em} // critical section
13 \hspace{1em} turn := 1;
14 }
A Kripke structure for a concurrent program

\[ P_1 \]
10 while (true) {
11 \hspace{1em} \texttt{wait}(\texttt{turn} == 0);
12 \hspace{1em} // critical section
13 \hspace{1em} \texttt{turn} := 1;
14 }

\[ P_2 \]
20 while (true) {
21 \hspace{1em} \texttt{wait}(\texttt{turn} == 1);
22 \hspace{1em} // critical section
23 \hspace{1em} \texttt{turn} := 0;
24 }
A Kripke structure for a concurrent program

\[ \text{P}_1 \]
10 while (true) {
11 \hspace{1em} wait(turn == 0);
12 \hspace{1em} // critical section
13 \hspace{1em} turn := 1;
14 }

\[ \text{P}_2 \]
20 while (true) {
21 \hspace{1em} wait(turn == 1);
22 \hspace{1em} // critical section
23 \hspace{1em} turn := 0;
24 }

\[ \text{turn=0, 10, 20} \]
\[ \text{turn=0, 11, 20} \]
\[ \text{turn=0, 12, 20} \]
\[ \text{turn=1, 10, 20} \]
\[ \text{turn=1, 11, 20} \]
\[ \text{turn=1, 11, 21} \]
\[ \text{turn=1, 11, 22} \]
\[ \text{turn=1, 10, 21} \]
\[ \text{turn=1, 10, 22} \]
\[ \text{turn=1, 11, 21} \]
\[ \text{turn=1, 11, 22} \]
A Kripke structure for a concurrent program

\[ P_1 \]
10 while (true) {
11 wait(turn == 0);
12 \hspace{2em} // critical section
13 \hspace{2em} turn := 1;
14 }

\[ P_2 \]
20 while (true) {
21 wait(turn == 1);
22 \hspace{2em} // critical section
23 \hspace{2em} turn := 0;
24 }
A Kripke structure for a concurrent program

\[ P_1 \]

10 while (true) {
11 wait(turn == 0);
12 // critical section
13 turn := 1;
14 }

\[ P_2 \]

20 while (true) {
21 wait(turn == 1);
22 // critical section
23 turn := 0;
24 }
Safety & liveness properties of reactive systems

Safety

• “Nothing bad will happen.”

• $\phi$ is a safety property iff every infinite path $\pi$ violating $\phi$ has a finite prefix $\pi'$ such that every extension of $\pi'$ violates $\phi$.

Liveness

• “Something good will happen.”

• $\psi$ is a liveness property iff every finite path (prefix) $\pi$ can be extended so that it satisfies $\psi$. 
Safety & liveness properties of reactive systems

Safety

- “Nothing bad will happen.”
- \( \varphi \) is a safety property iff every infinite path \( \pi \) violating \( \varphi \) has a finite prefix \( \pi' \) such that every extension of \( \pi' \) violates \( \varphi \).

Liveness

- “Something good will happen.”
- \( \psi \) is a liveness property iff every finite path (prefix) \( \pi \) can be extended so that it satisfies \( \psi \).

Finite witnesses (counterexamples).
Reducible to checking reachability in the state transition graph.
Safety & liveness properties of reactive systems

Safety
• “Nothing bad will happen.”
• \( \varphi \) is a safety property iff every infinite path \( \pi \) violating \( \varphi \) has a finite prefix \( \pi' \) such that every extension of \( \pi' \) violates \( \varphi \).

Liveness
• “Something good will happen.”
• \( \psi \) is a liveness property iff every finite path (prefix) \( \pi \) can be extended so that it satisfies \( \psi \).

Finite witnesses (counterexamples).
Reducible to checking reachability in the state transition graph.

No finite witnesses (counterexamples).
Safety & liveness properties of reactive systems

Safety

• “Nothing bad will happen.”

• $\varphi$ is a safety property iff every infinite path $\pi$ violating $\varphi$ has a finite prefix $\pi'$ such that every extension of $\pi'$ violates $\varphi$.

Liveness

• “Something good will happen.”

• $\psi$ is a liveness property iff every finite path (prefix) $\pi$ can be extended so that it satisfies $\psi$.

Mutual exclusion: $P_1$ and $P_2$ will never be in their critical regions simultaneously.
Safety & liveness properties of reactive systems

**Safety**
- “Nothing bad will happen.”
- $\varphi$ is a safety property iff every infinite path $\pi$ violating $\varphi$ has a finite prefix $\pi'$ such that every extension of $\pi'$ violates $\varphi$.

**Liveness**
- “Something good will happen.”
- $\psi$ is a liveness property iff every finite path (prefix) $\pi$ can be extended so that it satisfies $\psi$.

**Mutual exclusion:** $P_1$ and $P_2$ will never be in their critical regions simultaneously.

**Starvation freedom:** whenever $P_1$ is ready to enter its critical section, it will eventually succeed (provided that the scheduler is *fair* and does not let $P_2$ stay in its critical section forever).
Expressing properties in temporal logics

**Linear time:** properties of computation paths

- $a\ b \rightarrow b\ c \rightarrow a\ b \rightarrow \ldots$
- $a\ b \rightarrow c \rightarrow c \rightarrow \ldots$

**Branching time:** properties of computation trees

- $a\ b \rightarrow b\ c \rightarrow a\ b \rightarrow b\ c \rightarrow \ldots$
- $a\ b \rightarrow c \rightarrow c \rightarrow \ldots$
- $a\ b \rightarrow c \rightarrow \ldots$
Computation tree logic CTL*

Path quantifiers describe the branching structure of the computation tree:

- A (for all paths)
- E (there exists a path)

Temporal operators describe properties of a path through a tree:

- Xp (p holds “next time”)
- Fp (p holds “eventually” or “in the future”)
- Gp (p holds “always” or “globally”)
- p U q (p holds “until” q holds)
Syntax of CTL*

State formulas
- Atomic propositions: \( a \in AP \)
- \( \neg f, f \land g, f \lor g \), where \( f \) and \( g \) are state formulas
- \( Ap \) and \( Ep \), where \( p \) is a path formula

Path formulas
- \( f \), where \( f \) is a state formula
- \( \neg p, p \land q, p \lor q \), where \( p \) and \( q \) are path formulas
- \( Xp, Fp, Gp, p \ U q \), where \( p \) and \( q \) are path formulas
Semantics of CTL*

State formulas

- $M, s \models a$ iff $a \in L(s)$
- $M, s \models Ap$ iff $M, \pi \models p$ for all paths $\pi$ that start at $s$
- $M, s \models Ep$ iff $M, \pi \models p$ for some path $\pi$ that starts at $s$

Path formulas ($\pi^k$ is suffix of $\pi$ starting at $s_k$)

- $M, \pi \models f$ iff $M, s \models f$ and $s$ is the first state of $\pi$
- $M, \pi \models Xp$ iff $M, \pi^1 \models p$
- $M, \pi \models Fp$ iff $M, \pi^k \models p$ for some $k \geq 0$
- $M, \pi \models Gp$ iff $M, \pi^k \models p$ for all $k \geq 0$
- $M, \pi \models p \bigcup q$ iff $M, \pi^k \models q$ and $M, \pi^j \models q$ for some $k \geq 0$ and for all $0 \leq j < k$
CTL and Linear Temporal Logic (LTL)

Computation Tree Logic (CTL)

- Fragment of CTL* in which each temporal operator is prefixed with a path quantifier.
- $AG(\ EF\ p)$: From any state, it is possible to get to a state where $p$ holds.

Linear Temporal Logic (LTL)

- Fragment of CTL* with formulas of the form $A\ p$, where $p$ contains no path quantifiers.
- $A(\ FG\ p)$: Along every path, there is some state from which $p$ will hold forever.
Computation Tree Logic (CTL)

- Fragment of CTL* in which each temporal operator is prefixed with a path quantifier.
- $\text{AG}(\text{EF} \ p)$: From any state, it is possible to get to a state where $p$ holds.

Linear Temporal Logic (LTL)

- Fragment of CTL* with formulas of the form $\text{A}p$, where $p$ contains no path quantifiers.
- $\text{A}(\text{FG} \ p)$: Along every path, there is some state from which $p$ will hold forever.
CTL and Linear Temporal Logic (LTL)

Computation Tree Logic (CTL)

• Fragment of CTL* in which each temporal operator is prefixed with a path quantifier.

• $\text{AG}(\text{EF } p)$: From any state, it is possible to get to a state where $p$ holds.

Linear Temporal Logic (LTL)

• Fragment of CTL* with formulas of the form $\text{Ap}$, where $p$ contains no path quantifiers.

• $\text{A(FG } p)$: Along every path, there is some state from which $p$ will hold forever.
Expressive power of CTL, LTL, and CTL∗

CTL
AG(EF p)

LTL
A(FG p)

CTL∗
Fairness

Cannot be expressed in CTL

Can be expressed in LTL

Handling fairness:
- Changed semantics to use fair Kripke structures.
- A fair Kripke structure $M = \langle S, S_0, R, L, F \rangle$ includes an additional set $F \subseteq 2^S$.
- For each $P \in F$, a fair path $\pi$ includes some states from $P$ infinitely often.

Path quantifiers interpreted only with respect to fair paths.

Fairness expressed in LTL:
- Absolute fairness: $A(\text{GF} p \text{exec})$
- Strong fairness: $A(\text{GF} p \text{ready} \Rightarrow \text{GF} p \text{exec} p \text{exec})$
- Weak fairness: $A(\text{FG} p \text{ready} \Rightarrow \text{GF} p \text{ready} \land p \text{exec})$
Cannot be expressed in CTL

- Handled by changing the semantics to use fair Kripke structures.
- A fair Kripke structure $M = \langle S, S_0, R, L, F \rangle$ includes an additional set of sets of states $F \subseteq 2^S$.
- For each $P \in F$, a fair path $\pi$ includes some states from $P$ infinitely often.
- Path quantifiers interpreted only with respect to fair paths.

Can be expressed in LTL

- Absolute fairness: $A(\text{GF}p_{\text{exec}})$
- Strong fairness: $A((\text{GF}p_{\text{ready}}) \equiv (\text{GF}p_{\text{ready}} \land p_{\text{exec}}))$
- Weak fairness: $A((\text{FG}p_{\text{ready}}) \equiv (\text{GF}p_{\text{ready}} \land p_{\text{exec}}))$
**Fairness**

Cannot be expressed in CTL

- Handled by changing the semantics to use fair Kripke structures.
- A *fair* Kripke structure $M = \langle S, S_0, R, L, F \rangle$ includes an additional set of sets of states $F \subseteq 2^S$.
- For each $P \in F$, a *fair path* $\pi$ includes some states from $P$ infinitely often.
- Path quantifiers interpreted only with respect to fair paths.

Can be expressed in LTL

- Absolute fairness: $A(GF p_{\text{exec}})$
- Strong fairness:
  $A((GF p_{\text{ready}}) \Rightarrow (GF p_{\text{ready}} \land p_{\text{exec}}))$
- Weak fairness:
  $A((FG p_{\text{ready}}) \Rightarrow (GF p_{\text{ready}} \land p_{\text{exec}}))$
Model checking complexity for CTL, LTL, CTL*

**Polynomial Time for CTL**
- Best known algorithm: $O(|M| \times |f|)$

**PSPACE-complete for LTL**
- Best known algorithm: $O(|M| \times 2^{|f|})$

**PSPACE-complete for CTL**
- Best known algorithm: $O(|M| \times 2^{|f|})$
Model checking techniques for CTL and LTL

**CTL**

- Graph-theoretic explicit-state model checking (EMC)
- Symbolic model checking with Ordered Binary Decision Diagrams (SMV, NuSMV)
- Bounded model checking based on SAT (NuSMV)

**LTL**

- Automata-theoretic model checking:
  - Explicit-state (SPIN) or
  - Symbolic (NuSMV)
Summary

Today

- Basics of model checking:
  - Kripke structures
  - Temporal logics (CTL, LTL, CTL*)
  - Model checking techniques

Next lecture

- Software model checking