Software Verification

Lecture 7: Model Checking

Carlo A. Furia
Program Verification: the very idea

P: a program

max (a, b: INTEGER): INTEGER is
  do
    if a > b then
      Result := a
    else
      Result := b
    end
  end

S: a specification

require
  true

ensure
  Result >= a
  Result >= b

Does \( P \models S \) hold?

The Program Verification problem:

- **Given**: a program \( P \) and a specification \( S \)
- **Determine**: if any execution of \( P \), for any value of input parameters, satisfies \( S \)
Why is Verification Difficult?

The very nature of (universal) computation entails the impossibility of solving automatically the program verification problem.

\[ P: \text{a program} \quad S: \text{a specification} \]

\[ \Leftrightarrow \quad \Leftrightarrow \]

\[ \text{TM}(P): \text{a Turing machine} \quad \text{F}(S): \text{a first-order formula} \]

Does \[ \text{TM}(P) \models F(S) \] hold?

UNDECIDABLE
Decidability vs. Expressiveness Trade-Off

If we restrict the expressiveness of:

- the computational model
  and/or
- the specification language
the verification problem can become decidable

Does $P \models S$ hold?

**Def. Expressiveness**: capability of describing extensive classes of:

- computations
- properties
Verification of Finite-state Programs
Verification of Finite-state Programs

In Model Checking we typically assume:

- finite-state programs
  - every variable has finite domain
  - equivalently: finite-state automata of some kind
- monadic first-order logic
  - restricted first-order logic fragment where the ordering of state values during a computation can be expressed
  - equivalently: temporal logic of some kind

\[
\begin{align*}
P: \text{a program} & \iff & S: \text{a specification} \\
FSA(P): \text{a finite-state automaton} & \iff & TL(S): \text{a temporal-logic formula}
\end{align*}
\]

Does \( FSA(P) \models TL(S) \) hold? DECIDABLE
Model-Checking by Pictures

is_locked: BOOLEAN

toggle_lock: is
do
  is_locked := not is_locked
end

ensure
is_locked = not old is_locked

P: a program

S: a specification

FSA(P): a finite-state automaton

TL(S): a temporal-logic formula

\[ \vdash (\text{toggle}_\text{lock} \Leftrightarrow \Box \text{toggle}_\text{lock}) \]
Finite-state Programs in the Real World

Can finite-state models capture significant aspects of real programs? Yes!

A few examples:

- Behavior of hardware
  - inherently finite-state
- Concurrency aspects
  - access to critical regions, scheduling of processes, ...
- Security aspects
  - access policies, simple protocols, ...
- Reactive systems
  - ongoing interaction between software and physical environment
Is the Abstraction Correct?

How to guarantee that the finite-state abstraction of an infinite-state program is correct?

- In hardware verification, the real system is finite-state, so no abstraction is needed
- The finite-state model can be built and verified before the real implementation is produced
  - A formal high-level model
  - Increased confidence in some key features of the system under development
  - Model-driven development: the implementation is derived (almost) automatically from the high-level finite-state model
Is the Abstraction Correct?

How to guarantee that the finite-state abstraction of an infinite-state program is correct?

- **Software model-checking**: the abstraction is built automatically and refined iteratively until we can guarantee that it is an accurate model of the real implementation for the properties under verification.
The Model-checking Paradigm
The Model Checking Paradigm

The Model Checking problem:

- **Given**: a finite-state automaton $A$ and a temporal-logic formula $F$
- **Determine**: if any run of $A$ satisfies $F$ or not
  - if not, also provide a counterexample:
    a run of $A$ where $F$ does not hold

$A$: a finite-state automaton

$F$: a temporal-logic formula

$\models \square (\text{toggle\_lock} \Leftrightarrow X \text{toggle\_lock})$
The Model-Checking Paradigm

A: a finite-state automaton

F: a temporal-logic formula

\[ \models \square (\text{toggle\_lock} \leftrightarrow X \text{toggle\_lock}) \]

Different choices are possible for the family of automata and of formulae.

- We now explore more details for linear-time model-checking where:
  - A is a (nondeterministic) finite state automaton
  - F is a propositional linear temporal logic formula
Def. Nondeterministic Finite State Automaton (FSA):

- a tuple \([\Sigma, S, I, \rho, F]\):
  - \(\Sigma\): finite nonempty (input) alphabet
  - \(S\): finite nonempty set of states
  - \(I \subseteq S\): set of initial states
  - \(F \subseteq S\): set of accepting states
  - \(\rho: S \times \Sigma \rightarrow 2^S\): transition function
Finite State Automata: Syntax

Def. Nondeterministic Finite State Automaton (FSA):
   a tuple \([\Sigma, S, I, \rho, F]\):
   - \(\Sigma\): finite nonempty (input) alphabet
   - \(S\): finite nonempty set of states
   - \(I \subseteq S\): set of initial states
   - \(F \subseteq S\): set of accepting states
   - \(\rho: S \times \Sigma \rightarrow 2^S\): transition function

- \(\Sigma = \{\text{pull, push, turn\_on, turn\_off, start, stop, cook}\}\)
- \(S = \{\text{closed-off, open-off, closed-on, open-on, closed-cooking}\}\)
- \(I = \{\text{closed-off}\}\)
- \(F = \{\text{closed-off}\}\)
- \(\rho(\text{closed-off, turn\_on}) = \{\text{closed-on}\}\)
- \(\rho(\ldots, \ldots) = \ldots\)
- Deterministic, in this example (“microwave oven”)
Def. An accepting run of an FSA $A=\langle \Sigma, S, I, \rho, F \rangle$ over input word $w = w(1) w(2) \ldots w(n) \in \Sigma^*$ is a sequence $r = r(0) r(1) r(2) \ldots r(n) \in S^*$ of states such that:

- it starts from an initial state: $r(0) \in I$
- it ends in an accepting state: $r(n) \in F$
- it respects the transition function: $r(i+1) \in \rho(r(i), w(i))$ for all $0 \leq i < n$
Finite State Automata: Semantics

Def. An accepting run of an FSA $A = [\Sigma, S, I, \rho, F]$ over input word $w = w(1) \ldots w(n) \in \Sigma^*$ is a sequence $r = r(0) r(1) \ldots r(n) \in S^*$ of states such that:

- it starts from an initial state: $r(0) \in I$
- it ends in an accepting state: $r(n) \in F$
- it respects the transition function: $r(i+1) = \rho(r(i), w(i))$ for all $0 \leq i < n$

• Accepting run
  $r = \text{closed-off closed-on closed-cooking closed-cooking closed-on closed-off}$

• Over input word
  $w = \text{turn_on start cook stop turn_off}$

• In practice: any path on the directed graph that starts in an initial state and ends in an accepting state
Def. Any FSA $A=\{\Sigma, S, I, \rho, F\}$ defines a set of input words $\langle A \rangle$:

$$\langle A \rangle \doteq \{ w \in \Sigma^* \mid \text{there is an accepting run of A over } w \}$$

$\langle A \rangle$ is called the language of $A$. 
Finite State Automata: Semantics

Def. Any FSA $A = [\Sigma, S, I, \rho, F]$ defines a set of input words $\langle A \rangle$:

$$\langle A \rangle \triangleq \{ w \in \Sigma^* \mid \text{there is an accepting run of } A \text{ over } w \}$$

$\langle A \rangle$ is called the language of $A$

With regular expressions:

$$\langle A \rangle = ( (\text{pull push})^* \text{ turn}_\text{on} \ (\text{pull push})^* \ (\text{start cook}^* \text{ stop})^* \ (\text{pull push})^* \text{ turn}_\text{off} )^*$$
Linear Temporal Logic: Syntax

Def. Propositional Linear Temporal Logic (LTL) formulae are defined by the grammar:

\[ F ::= p \mid \neg F \mid F \land G \mid X F \mid F \cup G \]

with \( p \in P \) any atomic proposition from a fixed set \( P \).

Temporal (modal) operators:

- next: \( X F \)
- until: \( F \cup G \)
- release: \( F R G \triangleq \neg (\neg F \cup \neg G) \)
- eventually: \( \lozenge F \triangleq \text{True} U F \)
- always: \( \Box F \triangleq \neg \lozenge \neg F \)

Propositional connectives:

- not: \( \neg F \)
- and: \( F \land G \)
- or: \( F \lor G \triangleq \neg (\neg F \land \neg G) \)
- implies: \( F \Rightarrow G \triangleq \neg F \lor G \)
- iff: \( F \Leftrightarrow G \triangleq (F \Rightarrow G) \land (G \Rightarrow F) \)
Def. Propositional Linear Temporal Logic (LTL) formulae are defined by the grammar:

\[ F ::= p \mid \neg F \mid F \land G \mid X F \mid F \cup G \]

with \( p \in P \) any atomic proposition from a fixed set \( P \).

\[ \square (\text{start} \Rightarrow X (\text{cook} \cup \text{stop})) \]
Def. A word $w = w(1) w(2) \ldots w(n) \in (2^\mathbb{P})^*$ satisfies an LTL formula $F$

at position $1 \leq i \leq n$, denoted $w, i \models F$,
under the following conditions:

- $w, i \models p$ iff $p \in w(i)$
- $w, i \models \neg F$ iff $w, i \models F$ does not hold
- $w, i \models F \land G$ iff both $w, i \models F$ and $w, i \models G$ hold
- $w, i \models X F$ iff $i < n$ and $w, i+1 \models F$

• i.e., $F$ holds in the next step

- $w, i \models F U G$ iff for some $i \leq j \leq n$ it is: $w, j \models G$ and for all $i \leq k < j$ it is $w, k \not\models F$

• i.e., $F$ holds until $G$ will hold
For derived operators:

\[-w, i \models \Diamond F \quad \text{iff} \quad \text{for some } i \leq j \leq n \text{ it is: } w, j \models F\]
- i.e., \( F \) holds \textit{eventually} (in the future)

\[-w, i \models \Box F \quad \text{iff} \quad \text{for all } i \leq j \leq n \text{ it is: } w, j \models F\]
- i.e., \( F \) holds \textit{always} (in the future)
Def. Satisfaction:

\[ w \models F \triangleq w, 1 \models F \]

i.e., word \( w \) satisfies formula \( F \) initially

Def. Any LTL formula \( F \) defines a set of words \( \langle F \rangle \):

\[ \langle F \rangle \triangleq \{ w \in (2^\mathcal{P})^* \mid w \models F \} \]

\( \langle F \rangle \) is called the language of \( F \)
Def. Any LTL formula $F$ defines a set of words $\langle F \rangle$:

$$\langle F \rangle \triangleq \{ w \in (2^P)^* \mid w \models F \}$$

$\langle F \rangle$ is called the language of $F$

$$\langle \Box \text{start} \rangle = \{ \text{start, ...} \} \{ \text{start, ...} \} \{ \text{start, ...} \} \ldots$$
Verification as Emptiness Checking

The Model Checking problem:

• **Given:** a finite-state automaton \( A \) and a temporal-logic formula \( F \)

• **Determine:** if any run of \( A \) satisfies \( F \) or not
  
  - if not, also provide a **counterexample:**
    a run of \( A \) where \( F \) does not hold

\[
\begin{align*}
? & \\
A: \text{a finite-state automaton} & \models F: \text{a temporal-logic formula} \\
\iff & \\
\langle A \rangle & = \text{words accepted by } A \quad \langle F \rangle & = \text{words satisfying } F
\end{align*}
\]
Verification as Emptiness Checking

\[ A: \text{a finite-state automaton} \quad \not\models F: \text{a temporal-logic formula} \]

\[ \langle A \rangle = \text{words accepted by } A \quad \langle F \rangle = \text{words satisfying } F \]

\[ A \models F \quad \text{means: “every accepting run of } A \text{ produces a word that satisfies } F \” \]

\[ A \models F \quad \text{iff:} \quad w \in \langle A \rangle \implies w \in \langle F \rangle \]
\[ \text{iff:} \quad \langle A \rangle \subseteq \langle F \rangle \]
\[ \text{iff:} \quad \langle A \rangle \cap \langle F \rangle^c = \emptyset \]
\[ \text{iff:} \quad \langle A \rangle \cap \langle \neg F \rangle = \emptyset \]
Automata-theoretic Model Checking

An semantic view of the Model Checking problem:

- **Given:** a finite-state automaton \( A \) and a temporal-logic formula \( F \)

- if \( \langle A \rangle \cap \langle \neg F \rangle \) is empty then any run of \( A \) satisfies \( F \)

- if \( \langle A \rangle \cap \langle \neg F \rangle \) is not empty then some run of \( A \) does not satisfy \( F \)

  - any member of the nonempty intersection \( \langle A \rangle \cap \langle \neg F \rangle \) is a counterexample
Automata-theoretic Model Checking

How to check $\langle A \rangle \cap \langle \neg F \rangle = \emptyset$ algorithmically (given $A$, $F$)?

Combination of three different algorithms:

- **LTL2FSA**: given LTL formula $F$ build automaton $a(F)$ such that $\langle F \rangle = \langle a(F) \rangle$

- **FSA-Intersection**: given automata $A$, $B$ build automaton $C$ such that $\langle A \rangle \cap \langle B \rangle = \langle C \rangle$

- **FSA-Emptiness**: given automaton $A$ check whether $\langle A \rangle = \emptyset$ is the case
LTL2FSA: from LTL to FSA

Given an LTL formula $F$, it is always possible to build automatically an FSA $a(F)$ that accepts precisely the same words that satisfy $F$.

There are various algorithms to achieve this, with various degrees of sophistication and efficiency. Let us skip the details and just demonstrate the idea on an example.
LTL2FSA: from LTL to FSA

□ ( start ⇒ X (cook U stop) )

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop

As long as start does not occur, everything's fine.

start occurs: move to a different (non-accepting) state and start monitoring.

stop must occur in the future for things to be fine.

cook can occur before stop does.
LTL2FSA: from LTL to FSA

\(\square (\text{start} \Rightarrow X (\text{cook} U \text{stop}) )\)

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop

A few remarks:
- \(\neg\text{start}\) means “any events other than start”
- what happens if start alone occurs in B2?
- what happens if start occurs in B2 with cook or stop?
- what happens if cook \& stop occurs in B2?
- what happens if neither cook nor stop occur in B2?

Remember that any element of the input alphabet is a set of propositions!
LTL2FSA: complete the transitions

![Diagram showing state transitions]

□ ( start ⇒ X (cook U stop) )

- Always:
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop
  - if this doesn't happen, fail
LTL2FSA: complement

□ ( start ⇒ X (cook U stop) )

¬□ ( start ⇒ X (cook U stop) )

≡

◊ ( start ∧ X (¬cook R ¬stop) )

- **Always:**
  - when start occurs:
    - stop will occur in the future and
    - cook holds until the occurrence of stop
  - if this doesn't happen, fail

- **Sometimes:**
  - start occurs and from that moment on:
    - cook becomes false after stop
**FSA-Intersection: running FSA in parallel**

Given automata $A$, $B$ it is always possible to build automatically an FSA $C$ that accepts precisely the words that both $A$ and $B$ accept.

Automaton $C$ represents all possible parallel runs of $A$ and $B$ where a word is accepted if and only if both $A$ and $B$ would accept it. The (simple) construction is called “product automaton”.
Def. Given FSA $A = [\Sigma, S^A, I^A, \rho^A, F^A]$ and $B = [\Sigma, S^B, I^B, \rho^B, F^B]$

let $C \triangleq A \times B \triangleq [\Sigma^C, S^C, I^C, \rho^C, F^C]$ be defined as:

• $\Sigma^C \triangleq \Sigma$
• $S^C \triangleq S^A \times S^B$
• $I^C \triangleq \{ (s, t) \mid s \in I^A \text{ and } t \in I^B \}$
• $\rho^C((s, t), \sigma) \triangleq \{ (s', t') \mid s' \in \rho^A(s, \sigma) \text{ and } t' \in \rho^B(t, \sigma) \}$
• $F^C \triangleq \{ (s, t) \mid s \in F^A \text{ and } t \in F^B \}$

Theorem.

$\langle A \times B \rangle$

= $\langle A \rangle \cap \langle B \rangle$
FSA-Intersection: running FSA in parallel

\[
\begin{array}{c}
\text{closed off} \quad \text{open off} \\
\text{push} \quad \text{pull} \\
\text{turn off} \quad \text{turn on} \\
\text{closed on} \quad \text{open on} \\
\text{stop} \quad \text{start} \\
\text{closed cooking} \quad \text{cook} \\
\end{array}
\times
\begin{array}{c}
\text{B1} \\
\text{start} \quad \text{stop} \\
\text{cook stop} \quad \text{cooking B2} \\
\text{B0} \\
\text{start} \quad \text{stop} \\
\text{closed cooking B2} \\
\end{array}
= 
\begin{array}{c}
\text{closed off B1} \\
\text{pull push} \\
\text{turn off} \\
\text{stop} \quad \text{start} \\
\text{closed on B1} \\
\text{pull push} \\
\text{turn on} \\
\text{closed cooking B2} \\
\text{cook stop} \\
\end{array}
\]
**FSA-Emptiness: node reachability**

Given an automaton $A$ it is always possible to check automatically if it accepts some word.

It suffices to check whether any final state can be reached starting from any initial state.

This amount to checking reachability on the graph representing the automaton: if a path is found, it corresponds to an accepted word; otherwise the automaton accepts an empty language.
FSA-Emptiness: node reachability

It suffices to check whether any final state can be reached starting from any initial state.

From the initial state B1 both accepting states can be reached.

Correspondingly we find the accepted words:
  • start
  • start cook cook
  • start stop start
  • ...

The accepted language is not empty.
Automata-theoretic Model Checking Algorithm:

• **Given**: a finite-state automaton $A$ and a temporal-logic formula $F$

1. **TL2FSA**: build “tableau” automaton $a(\neg F)$
2. **FSA-Intersection**: build “product” automaton $A \times a(\neg F)$
3. **FSA-Emptiness**: check whether $A \times a(\neg F) = \emptyset$

• If $A \times a(\neg F) = \emptyset$ then any run of $A$ satisfies $F$
• If $A \times a(\neg F) \neq \emptyset$ then show a run of $A$ where $F$ does not hold
Automata-theoretic Model Checking

\[ \Rightarrow \square (\text{start} \Rightarrow X (\text{cook} \cup U \text{stop})) \]
Variants of the Model-Checking Algorithm
Variants of the Model-Checking Algorithm

The basic model-checking algorithm:

- **TL2FSA**: build automaton \( a(\neg F) \)
- **FSA-Intersection**: build automaton \( A \times a(\neg F) \)
- **FSA-Emptiness**: check whether \( A \times a(\neg F) = \emptyset \)

Can be refined in different ways giving rise to the variants:

- **Explicit-state** model-checking
- **Symbolic (BDD-based)** model-checking
- **Bounded (SAT-based)** model-checking

The variants differ in how they represent automata and formulae and how they analyze them. Hybrid approaches are also possible.