CS 477: Model Checking

Sasa Misailovic

Based on previous slides by Elsa Gunter and Armando Solar-Lezama (used with permission)

University of Illinois at Urbana-Champaign
Another good time for a recap

• Propositional Logic
• Operational Program Semantics
• Dataflow Analysis (CFG + finite-height lattice)
• Abstract Interpretation (abstraction/concretization + CFG + infinite-height lattice)
• First order logic, as an engine for solving constraints extracted from Axiomatic program semantics
• Axiomatic Semantics

• Coming up next....
Model Checking Today

**Hardware Model Checking** - part of the standard toolkit for hardware design

- Intel has used it for production chips since Pentium 4
- For the Intel Core i7, most pre-silicon validation was done through formal methods (i.e. Model Checking + Theorem Proving)
- Many commercial products

**Software Model Checking**

- Static driver verifier now a commercial Microsoft product
- Java PathFinder used to verify code for mars rover
History of Model Checking

• Clarke and Emerson, “Design and Synthesis of Synchronization Skeletons using branching time temporal logic”

“Proof Construction is Unnecessary in the case of finite state concurrent systems and can be replaced by a model-theoretic approach which will mechanically determine if the system meets a specification expressed in propositional temporal logic”

• Obtained Turing Award

Precursors:

• Verification through exhaustive exploration of finite state models: G. V. Bochmann and J. Gecsei, A unified method for the specification and verification of protocols, Proc. IFIP Congress 1977

• Linear Temporal Logic, used for specifying system properties: A. Pnueli, The temporal semantics of concurrent programs. 1977
The model checking approach o (as characterized by Emerson)

• Start with a program that defines a finite state model $M$

• Search $M$ for patterns that tell you whether a specification $\phi$ holds

• Pattern specification is flexible

• The method is efficient in the sizes of $M$ and hopefully also $\phi$

• The method is algorithmic
Model Checking

Most generally Model Checking is
• an automated technique, that given
• a finite-state model $M$ of a system
• and a logical property $\varphi$,
• checks whether the property holds of model: $M \models \varphi$?
• or if it fails returns a counter-example (example of failure) – useful for debugging
Basic Notions of Model Theory

When an interpretation \( I \) makes \( S \) true, we say that \( I \) satisfies \( S \)

• or that \( I \) is a **model** of \( S \) (or \( I \models S \))

We are interested in deciding whether for the special case where

• \( I \) is a finite-state automaton with specific properties (e.g., Kripke structure or a labeled transition system)

• \( S \) is a temporal logic formula

**High-level Idea:**

• The program will determine the model – through the translation to the transition system

• Recall, in axiomatic semantics, the program was a part of the theorem
Kripke Structures as Models

• Kripke structure is a finite size model with labels

For a set AP of atomic propositions, Kripke structure = (S, S₀, R, L)
• S : finite set of states
• S₀ ⊆ S : set of initial states
• R ⊆ S x S : transition relation
• L : S → \wp(\text{AP}) : labels each state with a set of atomic propositions
Microwave Example

- $S = \{ s_1, s_2, s_3, s_4 \}$
- $S_0 = \{ s_1 \}$
- $R = \{ (s_1, s_2), (s_2, s_1), (s_1, s_4), (s_4, s_2), (s_2, s_3), (s_3, s_2), (s_3, s_3) \}$
- $L(s_1) = \{-close, -start, -cooking\}$
- $L(s_2) = \{close, -start, -cooking\}$
- $L(s_3) = \{close, start, cooking\}$
- $L(s_4) = \{-close, start, -cooking\}$

Q: Can the microwave cook with the door open (-close)?
Properties over States

State formula:

• Can be established as true or false on a given state
• If $p \in AP$ then $p$ is a state formula
• If $f$ and $g$ are state formulas, so are $(f \text{ and } g)$, $(\neg f)$, $(f \text{ or } g)$
• E.g.: not close and cooking
Linear Time Logic Syntax

• $\varphi ::= p \mid (\varphi) \mid \neg \varphi \mid \varphi \land \varphi' \mid \varphi \lor \varphi'$
  
  \begin{align*}
  &\mid \Diamond \varphi \mid \varphi \mathcal{U} \varphi' \mid \square \varphi \mid \lozenge \varphi \\
  \text{or} \mid X \varphi \mid \varphi \mathcal{U} \varphi' \mid G \varphi \mid F \varphi \quad \leftarrow \text{alternative notation}
\end{align*}

• $p$ – a proposition over state variables

• Standard negation, conjunction and disjunction

• $\Diamond \varphi$ – “next” (also denoted $X \varphi$)

• $\varphi \mathcal{U} \varphi'$ – “until”

• $\square \varphi$ – “box”, “always”, “forever” (also $G \varphi$)

• $\lozenge \varphi$ – “diamond”, “eventually”, “sometime” (also $F \varphi$)
Intuition

\[
p \rightarrow p
\]

\[
\circ \varphi \rightarrow \varphi
\]

\[
\varphi \cup \psi \rightarrow \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \psi
\]

\[
\Box \varphi \rightarrow \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi
\]

\[
\Diamond \varphi \rightarrow \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi \varphi
\]

\[
\Diamond \varphi \rightarrow \varphi
\]
Paths and Path Formulas

**Path** $\pi$: a sequence of connected states: $\pi := s_0, s_1, s_2, ...$

**Path formulae.** Let $f$ be a formula, which

- a state formula $p$ is also a path formula: $\llbracket p \rrbracket (\pi_i) := \llbracket p \rrbracket (s_i)$
- boolean operations on path formulae are path formulae
  - e.g. $\llbracket f \land g \rrbracket (\pi_i) := \llbracket f \rrbracket (\pi_i) \land \llbracket g \rrbracket (\pi_i)$
- path quantifiers
  - $\llbracket \square \rrbracket f (\pi_i) := \text{globally } f (\pi_i) = \forall k \geq i . \llbracket f \rrbracket (\pi_k)$
  - $\llbracket \diamond \rrbracket f (\pi_i) := \text{eventually } f (\pi_i) = \exists k \geq i . \llbracket f \rrbracket (\pi_k)$
  - $\llbracket \lozenge \rrbracket f (\pi_i) := \text{next } f (\pi_i) = \llbracket f \rrbracket (\pi_{i+1})$
  - $f \mathbin{U} g (\pi_i) := f \text{ until } g = \begin{align*}
  \exists k \geq i \quad \text{s.t.} \quad & \llbracket g \rrbracket (\pi_k) \quad \text{and} \quad \forall j. \quad i \leq j < k \land \llbracket f \rrbracket (\pi_j)
\end{align*}$

Given a formula $f$ and a path $\pi$ if $\llbracket f \rrbracket (\pi)$ is true, then $\pi \models f$
Liveness Vs. Safety

Two very common terms:

Safety:
• Something bad will never happen: $G \neg \text{bad}$
• If it fails to hold, it’s easy to produce a witness

Liveness:
• Something good will eventually happen: $F \text{good}$
• What does a witness for this look like? ($\neg F \text{good}$)
Box vs. Diamond

- \( \Diamond p \iff \neg \Box \neg p \)
- \( \square p \iff \neg \Diamond \neg p \)

In another notation:

- \( F p \iff \neg G \neg p \)
- \( G p \iff \neg G \neg p \)
Common Combinations

• p will hold infinitely often:
  • G (F p)

• p will continuously hold from some point on
  • F (G p)

• If p happens infinitely often, then so does q
  • (G p) \implies (G q)
LTL Examples

• If you submit your homework (submit) you eventually get a grade back (grade)

• You should get your grade before you submit the next homework

• If assignment i was submitted before drop date, you should get your grade before drop date
LTL Examples

• If you submit your homework (submit) you eventually get a grade back (grade)
  • G (submit ⇒ F grade)

• You should get your grade before you submit the next homework
  • G (submit ⇒ X ( ¬submit U grade ) )

• If assignment i was submitted before the drop date, you should get your grade before the drop date
  • ( G (submit_i ⇒ F dropDate) ) ⇒ ( G ( grade_i ⇒ F dropDate ) )
  • and G (submit_i ⇒ F grade_i)
Microwave Example

- \( S = \{ s_1, s_2, s_3, s_4 \} \)
- \( S_0 = \{ s_1 \} \)
- \( R = \{ (s_1,s_2), (s_2,s_1), (s_1,s_4), (s_4,s_2), (s_2,s_3), (s_3,s_2), (s_3,s_3) \} \)
- \( L(s_1) = \{ -\text{close}, -\text{start}, -\text{cooking} \} \)
- \( L(s_2) = \{ \text{close}, -\text{start}, -\text{cooking} \} \)
- \( L(s_3) = \{ \text{close}, \text{start}, \text{cooking} \} \)
- \( L(s_4) = \{ -\text{close}, \text{start}, -\text{cooking} \} \)

Q: Can the microwave cook with the door open (-close)?
Microwave Example

Q: Can the microwave cook with the door open (-close)?
Reminder: Transition System (TS)

Describes potential system behaviors

• **TS:** Tuple \((S, \Theta, \rightarrow)\): S is set of states, \(\Theta \subseteq S\) are start states, \(\rightarrow\) is a relation of state transitions
  • \(\rightarrow \subseteq S \times S\) (we often write \(s_1 \rightarrow s_2\) for \((s_1, s_2) \in \cdot \rightarrow \cdot\))

• **Labeled TS:** \((S, \Theta, \rightarrow, \Lambda)\): \(\Lambda\) is a set of labels
  • \(\rightarrow \subseteq S \times \Lambda \times S\)
  • we often write \(s_1 \overset{\lambda}{\rightarrow} s_2\) for \((s_1, \lambda, s_2) \in \cdot \rightarrow \cdot\).

• (recall) For atomic propositions set AP, Kripke structure = \((S, S_0, R, L)\)
  • S : finite set of states; \(S_0 \subseteq S\) : set of initial states
  • \(R \subseteq S \times S\) : transition relation
  • \(L : S \rightarrow \wp (AP)\) : labels **each state** with a set of atomic propositions
Why are Kripke Structures Enough?

• Can still represent all (finite or infinite) traces
Liveness Vs. Safety

• Two terms you are likely to run into:

• Safety:
  • Something bad will never happen: $G \neg \text{bad}$
  • If it fails to hold, it’s easy to produce a witness

• Liveness:
  • Something good will eventually happen: $F \text{good}$
  • What does a witness for this look like?
Automata for LTL properties

• LTL defines properties over a trace

• Given a trace, we want to know whether it satisfies the property

• Model checking: $\text{Language(Model)} \subseteq \text{Language(Formula)}$

• Problem:
  • we need to build an automata to recognize infinite strings!
  • $\omega$ – Regular Languages
Reminder: Finite State Machine (FSM)

• A FSM is a 5-tuple \(\langle \Sigma, S, I, \delta, F \rangle\)
• \(\Sigma\) is an **alphabet**
• \(S\) is a finite set of **states**
• \(I \subseteq S\) is a set of **initial states**
• \(\delta \subseteq S \times \Sigma \times S\) is a **transition relation**
• \(F \subseteq S\) is a set of **accepting states**

Accepts the word \(w\) iff it ends in the accepting state after consuming the word
Buchi Automata

- Similar to a DFA
  - but with a stronger notion of acceptance

- In DFA, you have an accept state
  - when you reach accept state, you are done
  - this means you only accept finite strings

- In Buchi automata you also have accepting states
  but you only accept strings that visit the accept state infinitely often
(Non-deterministic) Buchi Automata

• A Buchi Automaton is a 5-tuple $\langle \Sigma, S, I, \delta, F \rangle$
  • $\Sigma$ is an alphabet
  • $S$ is a finite set of states
  • $I \subseteq S$ is a set of initial states
  • $\delta \subseteq S \times \Sigma \times S$ is a transition relation
  • $F \subseteq S$ is a set of accepting states

• Non-deterministic Buchi Automata are not equivalent to deterministic ones
Basic examples

• $p$

• $Fp$

• $Gp$

• $p U q$
Basic examples

- \( p \)

- \( F p \)

- \( p U q \)

Graphs from: http://www.lsv.fr/~gastin/ltl2ba/index.php
Example

• G F p
Example

- $G F p$
Example

- $F_p \Rightarrow G\ q$
Example

- $G \text{ rec} \Rightarrow F \text{ ack}$
CS 477: Model Checking

Sasa Misailovic

Based on previous slides by Elsa Gunter and Armando Solar-Lezama (used with permission)

University of Illinois at Urbana-Champaign
From LTL to automata

• Any LTL formula can be expressed as a non-deterministic Buchi automata (NBA)

• But the construction of the automata is complicated: exponential on the size of the formula


• To visualize the formula:
Explicit State Model checking

The Basic Strategy

Temporal Logic Formula

Buchi Automata

Product Automata

Model checker

OK

Counterexample trace

Kripke structure
A Bit About Complexity

• Satisfiability of a LTL formula: PSPACE-hard

• There is an algorithm that can solve the problem in $M \models F$ in $O(|M| \cdot 2^{|F|})$

• LTL and a fragment of FOL can express the same class of languages (infinite word languages)
  • But the ways of expressing the properties are different
  • For the detailed treatment, see Mahesh’s notes: https://courses.engr.illinois.edu/cs498mv/fa2018/LTL.pdf
Proof System (Informational)

• First: Extend all rules of Propositional Logic to LTL

• Second Step: Add one more rule

\[ \frac{G \phi}{\phi} \quad \text{Gen} \]

• Third Step: Add a collection of axioms (a sufficient set of 8 exists)

A1: \( G \phi \iff \neg(F(\neg\phi)) \)
A2: \( G(\phi \Rightarrow \psi) \Rightarrow (G\phi \Rightarrow G\psi) \)
A3: \( G\phi \Rightarrow (\phi \land X G \phi) \)
A4: \( X \neg\phi \iff \neg X\phi \)
A5: \( X(\phi \Rightarrow \psi) \Rightarrow (X\phi \Rightarrow X\psi) \)
A6: \( G(\phi \Rightarrow X\phi) \Rightarrow (\phi \Rightarrow G\phi) \)
A7: \( \phi U \psi \iff (\phi \land \psi) \lor (\phi \land X(\phi V \psi)) \)
A8: \( \phi U \psi \Rightarrow F \psi \)

• Result: a sound and relatively complete proof system
Buchi Automaton from Kripke Structure

• Given a Kripke structure:
  - $M = (S, S_0, R, L)$

• Construct a Buchi Automaton
  - $(\Sigma', S \cup \{\text{Init}\}, \{\text{Init}\}, T^*, S \cup \{\text{Init}\})$

- $T^*$ is defined s.t.
  - $T^*(s, \sigma, s')$ iff $R(s, s')$ and $\sigma \in L(s')$
  - $T^*(\text{Init}, \sigma, s)$ iff $s \in S_0$ and $\sigma \in L(s)$
Buchi Automaton from Kripke Structure

- \((\Sigma, S \cup \{\text{Init}\}, \{\text{Init}\}, T, S \cup \{\text{Init}\})\)
- T is defined s.t.
  - \(T(s, \sigma, s') \text{ iff } R(s, s') \text{ and } \sigma \in L(s')\)
  - \(T(\text{Init}, \sigma, s) \text{ iff } s \in S_0 \text{ and } \sigma \in L(s)\)
Negated Property

• Given a good property $P$, you can define a bad property $P'$
• If the system has a trace that satisfies $P'$, then it is buggy.

• Example
  - Good property: $G(\text{req} \Rightarrow F\text{ack})$
  - Bad property: $F(\text{req} \land (G \neg\text{ack}))$

• We are going to ask whether $M$ satisfies $P'$
  - If it does, then we found a bug
Computing the Product Automaton

• Given Buchi automata A and B’
  - $A = (\Sigma, S_A, T_A, \{\text{Init}_A\}, S_A)$
  - $B' = (\Sigma, S_B, T_B, \{\text{Init}_B\}, F')$
  - $A \times B' = (\Sigma, S_A \times S_B, T, \{(\text{Init}_A, \text{Init}_B)\}, F)$

• Where
  - $T((s_1, s_2), \sigma, (s_1', s_2'))$ iff $T_A(s_1, \sigma, s_1')$ and $T_B(s_2, \sigma, s_2')$
  - $(s_1, s_2) \in F$ iff $s_2 \in F'$
Check if a state is visited infinitely often

• Check for a cycle with an accepting state

• Cycle must be reachable from the initial state

**Simple algorithm**

• Do a depth-first search (DFS) to find an accepting state

• Do a DFS from that accepting state to see if it can reach itself
From Programs to Models

• Recall operational semantics

• Programs may have an infinite set of states (loops, recursion)

• To get a finite model, bound the number of iterations
Next...

• Continue the discussion from programs to models

• Following Jhala and Majumdar, “Software Model Checking” survey (most of text and examples from there)
Model Checking Concurrent Systems

• Based on Baier and Katoen Book

• Also find more on concurrent processes & SPIN in Moshe Vardi’s online chapter:
  https://cnx.org/contents/zVdF_TJw@3.4:13_CJz0Y@12/Concurrent-Programming-and-Verification-Outline
Reminder: Transition System (TS)

Describes potential system behaviors

- **TS**: Tuple \((S, \Theta, \rightarrow)\): \(S\) is set of states, \(\Theta \subseteq S\) are start states, \(\rightarrow\) is a relation of state transitions
  - \(\rightarrow \subseteq S \times S\) (we often write \(s_1 \rightarrow s_2\) for \((s_1, s_2) \in \rightarrow\))

- **Labeled TS***: \((S, A, \Theta, \rightarrow, \Lambda)\): \(A\) is a set of actions, \(\Lambda\) is a set of node labels
  - \(\rightarrow \subseteq S \times A \times S\)
  - we often write \(s_1 \xrightarrow{a} s_2\) for \((s_1, a, s_2) \in \rightarrow\)

*one of the versions*
Parallel Systems

\[ TS = TS1 \big|\big| TS2 \]

- Operator \( \big|\big| \) denotes that the two transition systems execute in parallel.
Concurrency and Interleaving

• Interleaving – widely adopted model for concurrent systems
• Create the global system state that comprises of individual parallel components
• Pure interleaving: non-deterministic choice between the transitions of concurrent processes
• E.g., for two processes P, Q some legal interleavings:

  P Q P Q P Q P Q P Q P Q P Q P Q P Q P Q P Q P Q ...
  P P Q P Q P P Q P Q Q P Q P Q ...
  Q P Q P P P P P P Q Q Q Q Q P ...
  Q Q Q Q Q Q Q Q Q Q Q Q Q Q Q Q ...

Scheduler

• Decides on the interleavings
• The interleaving model of concurrency abstracts over the choice of real-world scheduler or performance of the processes
• It models all possible interactions
• The number of possible interleavings often leads to a combinatorial explosion
Interleaving of Transition System

• Let \( TS_1 = (S_1, A_1, \Theta_1, \rightarrow_1, \Lambda_1) \) and \( TS_2 = (S_2, A_2, \Theta_2, \rightarrow_2, \Lambda_2) \) be two labeled transition systems

• The interleaving transition system \( TS = TS_1 \parallel TS_2 \)

\[
TS = (S_1 \times S_2, A_1 \cup A_2, \Theta_1 \times \Theta_2, \rightarrow, \Lambda)
\]

where

\[
\begin{align*}
S_1 \rightarrow_1 S_1' \\
\frac{a}{a}
\end{align*}
\]

\[
(s_1, s_2) \rightarrow (s_1', s_2)
\]

\[
\begin{align*}
S_2 \rightarrow_2 S_2' \\
\frac{a}{a}
\end{align*}
\]

\[
(s_2, s_2) \rightarrow (s_1, s_2')
\]

and \( \Lambda((s_1, s_2)) = \Lambda_1(s_1) \cup \Lambda_2(s_2) \)
Example: Traffic Lights
How about programs? No sharing data!

- $x:=x+1 \ |\ |\ y:=y-2$

The result of concurrently executing actions is identical to executing them in sequence!
Communicate via Shared Variables

- \( x := x + 1 \) || \( x := x - 2 \)

The result of concurrently executing actions is identical to executing them in sequence!
Communicate via Shared Variables

- Instead of defining $|\,|$ on transition systems, define it on the control flow graphs, $G_1$ and $G_2$
- The two graphs have a set of shared variables $X'$
  The other variables are private to $G_1$ and $G_2$
- Then obtain the underlying transition system $TS(G_1 | | G_2)$
Communicate via Shared Variables

- Instead of defining $\parallel$ on transition systems, define it on the control flow graphs, $G_1$ and $G_2$
- The two graphs have a set of shared variables $X'$
  - The other variables are private to $G_1$ and $G_2$
- Then obtain the underlying transition system $TS(G_1 \parallel G_2)$
Communicate via Channels

• Asynchronous communication
• Extend the language with
  send (Expr, ProcessID) and x:=recv(ProcessID)
• Example:
  
  \[
  [ \text{send} (1+1, \#1); x:=\text{recv}(\#1) ]_{\#0} \\
  || [ x:=\text{recv}(\#0); x:=x+1; \text{send}(x, \#0) ]_{\#1}
  \]

• Each process state has a buffer (FIFO queue) with incoming messages from other messages
• To build the transition system, take the || of CFGs (since the channels are analogous to shared variables between processes) but all other variables are local
• And then convert to its corresponding transition system.
Deadlock: Dining Philosophers

• A state of the system in which it cannot make a transition to perform a useful action
• Typically occurs when processes mutually wait for the others to proceed (e.g., release or acquire resources)
• Freedom of deadlocks is a weak form of liveness property
• If deadlocked, the program cannot reach the accepting state
Abstraction Refinement*

• When abstract analysis produces a counterexample, it can be
  • genuine, i.e., can be reproduced on the concrete program,
  • spurious, i.e., does not correspond to a real computation but arises due to imprecisions in the analysis

• If analysis imprecise, use the counterexample to refine it (CEGAR): use a new abstract domain that can represent strictly more sets of concrete program states [Ball and Rajamani 2000b; Clarke et al. 2000; Saidi 2000]

*from Jhala and Majumdar, Software Model Checking, 2009
Example Program

0:   LOCK = 0;
1:   do {
      LOCK = 1;
      old = new;
2:   if (*) {
3:     LOCK = 0;
      new++;  }  
4:   } while (new != old);
5:   if (LOCK==0)
6:   error();

LOCK = 0;  

Abstract Counterexample

- “The abstraction of the set of reachable set of states:
  
  $\text{ LOCK } = 1 \land \text{ new } = \text{ old } \lor \text{ LOCK } = 0 \land \text{ new } \neq \text{ old }$
  
captures the intuition that at line 4, either the lock is acquired and new is equal to old, or the lock is not acquired and the value of new is different from old (in fact, new = old + 1)”

- Abstraction insufficient if it tracks only the predicates LOCK = 0 and LOCK = 1
Example Program

```
0:  LOCK = 0;
1:  do {
    LOCK = 1;
    old = new;
2:    if (*) {
3:      LOCK = 0;
        new++;  
    }
4:  } while (new != old);
5:  if (LOCK==0)
6:      error();
    LOCK = 0;
```

Fig. 7. Program

```
Transitions  Constraints
LOCK = 0          LOCK_{\text{f}} = 0
LOCK = 1          LOCK_{\text{f}} = 1
old = new

[True]

LOCK = 0
new = new + 1
[new = old]

[LOCK == 0]          LOCK_{\text{f}} = 0
```

Fig. 8. Abstract Counterexample

• “The abstraction of the set of reachable set of states:
  \[(\text{LOCK} = 1 \land \text{new} = \text{old}) \lor (\text{LOCK} = 0 \land \text{new} \neq \text{old})\]
captures the intuition that at line 4, either the lock is acquired and \text{new} is equal to \text{old}, or the lock is not acquired and the value of \text{new} is different from \text{old} (in fact, \text{new} = \text{old} + 1)”

• Abstraction insufficient if it tracks only the predicates \text{LOCK} = 0 and \text{LOCK} = 1
CEGAR Overview

• Input to CEGAR algorithm: a path in the control flow graph that represents a possible counterexample produced by abstract reachability analysis.

• Step #1: constructs the trace formula from the path, such that the formula is satisfiable iff the path is executable by the concrete program.

• Step #2: a solver checks for satisfiability of the trace formula:
  • Yes -- the path is reported as a concrete counterexample to the property.
  • No -- the proof of unsatisfiability is mined for new predicates that can rule out the current counterexample when the abstract domain is augmented with these predicates.

• The CEGAR loop makes progress by eliminating at least one counterexample in each step.
Step 1: Trace Formula

- To convert an abstract counterexample into a trace formula, rename the state variables at each transition of the counterexample and conjoin the resulting transition to get

\[ \bigwedge_{i=0}^{n-1} \rho_i(X_i, X_{i+1}) \]

- Equivalent to the bounded model checking formula for the unrolled version of the program corresponding to the path, and also to the weakest precondition \( \text{wp}(\text{unrolled\_program}, \text{true}) \)

- To avoid constraints of the form \( x_{i+1} = x_i \) for each \( x \) not modified by an operation, convert the path to static single-assignment (SSA) form
  - Thus: \( \text{LOCK}_1, \text{LOCK}_2, \text{new}_1, \text{new}_2 \)
Step 2: Refinement (Syntax Based)

- **Strategy:** find an unsatisfiable core of atomic predicates appearing in the formula, whose conjunction is inconsistent.

- The trace formula is unsatisfiable as it contains the conjunction of 
  \( \text{old}_2 = \text{new}_0, \text{new}_4 = \text{new}_0 + 1, \text{new}_4 = \text{old}_2 \)

- **Refinement:** drop the subscripts [Henzinger et al. 2002] and the new predicates are 
  \( \text{old} = \text{new}, \text{new} = \text{new} + 1, \text{new} = \text{old} \)

  - hen this predicate is added to the set of predicates, the resulting set  
    \( \{\text{LOCK} = 0, \text{LOCK} = 1, \text{new} = \text{old}\} \) 

suffices to refute the counterexample, i.e. the path is not a counterexample in the abstract model generated from these predicates.