Overview

- Previous lectures: deductive verification + automated invariant inference
- If technique says “verified”, we are good; but if it says “not verified”, we don’t know anything
  - there might be a bug or maybe our loop invariants are not good enough...
- This lecture: Software model checking based on Counterexample-Guided Abstraction Refinement (CEGAR)
  - Can verify but also give counterexamples (i.e., witnesses to property violation)

Predicate Abstraction

- To understand CEGAR, good starting point is predicate abstraction
- Given a set of predicates \( P = \{p_1, \ldots, p_n\} \), predicate abstraction computes for every program location, an abstract value \([b_1, \ldots, b_n]\) where:
  - \( b_i \) indicates whether \( p_i \) holds or not at that location
  - values of \( b_i \) drawn from the set \( \{0, 1, \ast\} \) where \( \ast \) indicates unknown
- In other words, we have an abstract domain where each element is a formula \( \wedge_{i=1}^n b_i \) (sometimes called a cube)

Abstract Transformers

- We defined our abstract domain, but still need abstract transformers
- Given a statement \( S \) and cube \( \varphi \), define abstract transformer \( post^\#(S, \varphi) \) to be the strongest cube \( \varphi’ \) over \( P \) such that:
  \[ sp(S, \varphi) \Rightarrow \varphi’ \]
  where \( sp \) is the strongest post-condition of \( S \) wrt to \( \varphi \)
- Example: Suppose \( P = \{x = y, x \neq y, x \geq y\} \). What is \( post^\#(x := x + 1, x = y) \)?

Example

- Consider the program shown on the right
- And the predicate set \( P = \{x \leq 100, x = y, y = 100\} \)
- Compute the abstraction of this program wrt to \( P \)

```plaintext
x := 0; y := 0; while(x<100)
{
   x := x+1;
   y := y+1;
}
assert(y = 100);
```
**Motivation for CEGAR**

- Predicate abstraction is very sensitive to the set of predicates
- If you choose the right set, verification succeeds; otherwise, it fails
- The CEGAR paradigm allows automatically and iteratively discovering the right set of predicates

**Model Checking Basics**

- Intuitively, model checker explores all states program can be in
- Operates over control-flow automaton (CFA)
  - Like CFG but nodes/edges are flipped + explicit error locations
- Model checker performs exploration using a so-called state transition graph (STG)

**State Space**

- Given a program \( P \), define its state to be a tuple \((l, v_1, \ldots, v_n)\) where \( l \) is the control location and \( v_i \) denotes the value of \( i \)'th variable
- The state space of the program is all the states it can be in
- Model checker constructs a state transition graph (STG), where nodes are program states and an edge \((s, s')\) indicates that state \( s \) can transition to state \( s' \)
- Program has a bug if the error state is reachable

**Generating Boolean Programs**

- Given a set of predicates \( P \) and program \( S \), we want to generate a boolean program \( S' \) that has \(|P|\) boolean variables
- In particular, \( S' \) is the same as \( S \) except that each assignment is replaced with assignments to the boolean variables
- **Key question:** How do we translate an assignment to our boolean abstraction?
Modeling Statements in Boolean Program

- Consider stmt s and boolean b representing predicate p
- If wp(s, p) is true before s, then p should be assigned to true
- If wp(s, ¬p) is true before s, then p should be assigned to false (if neither, don’t know)
- But the exact wp may not be in our abstraction, so instead of the exact wp, compute the weakest cubes $P_1, P_2$ over $P$ such that $P_1 \Rightarrow wp(s, p)$ and $P_2 \Rightarrow wp(s, ¬p)$
- Thus model statement s as:
  
  $\begin{align*}
  &\text{if}(P_1) \ b := \text{true} \\
  &\text{else if}(P_2) \ b := \text{false} \\
  &\text{else} \ b := *
  \end{align*}$

Example

- Consider the predicates $\{x > 5, x < 5, y = 5\}$
- How do you model the statement $x := y$ in the boolean program with variables $b_1, b_2, b_3$?

Back to Model Checking

- Now that we have boolean programs, we can construct a finite STG
- For this program, there are four initial states:
  
  $\{(0, p_1, p_2), (0, p_1, ¬p_2), (0, ¬p_1, p_2), (0, ¬p_1, ¬p_2)\}$
- There is a transition from $(l, b_1, \ldots, b_n)$ to $(l', b'_1, \ldots, b'_n)$ iff:
  
  $\begin{align*}
  &\text{There must be a transition from } l \text{ to } l' \\
  &\text{labeled with } S
  \end{align*}$
  
  The formula $wp(S, A_i, h_i) \land A_j, h'_j$ must be satisfiable (query SAT solver!)

STG for Our Boolean Program, cont.

- Partial STG for our program:

```
(0, T, T) ▸ (0, T, F) ▸ (0, F, T) ▸ (0, F, F)
(1, T, T) ▸ (1, T, F) ▸ (1, F, T) ▸ (1, F, F)
(2, T, T) ▸ (2, T, F) ▸ (2, F, T) ▸ (2, F, F)
(3, T, T) ▸ (3, T, F) ▸ (3, F, T) ▸ (3, F, F)
```

- Verification fails because error state is reachable!

STG for Our Boolean Program, cont.

- Which of these transitions exist in the state transition graph?
  
  $\begin{align*}
  &\{(1, p_1, p_2) \text{ to } (3, ¬p_1, p_2)\} \\
  &\{(1, p_1, p_2) \text{ to } (3, p_1, p_2)\} \\
  &\{(3, ¬p_1, p_2) \text{ to } (err, ¬p_1, ¬p_2)\}
  \end{align*}$

Need for Refinement

- To make the STG finite, we worked with boolean programs
- But if the error state is reachable, this could be due to imprecision in the abstraction
  
  $\begin{align*}
  &\text{i.e., current set of predicates may not be fine-grain enough}
  \end{align*}$
- To decide how to proceed, we need to check if the property is actually violated
  
  Fortunately, the model checker can provide a counterexample in the form of a program trace!
Back to Running Example

Eliminating Counterexamples using SP

- Let \( l_0 \rightarrow s_1 \rightarrow s_2 \rightarrow \cdots \rightarrow s_n \) be a spurious cex trace
- Let \( p_0 \) be \texttt{true}, and define \( p_i \) as \( sp(s_i, p_{i-1}) \)
- Claim: Adding \( p_1, \ldots, p_n \) to \( \mathcal{P} \) will rule out this cex!
- Why is this true? Consider any potential path in the STG: 
  \( (l_0, \varphi_0) \rightarrow s_1 (l_1, \varphi_1) \rightarrow s_2 \rightarrow \cdots \rightarrow s_n (l_n, \varphi_n) \)
- Will show by induction on \( n \) that \( \varphi_i \Rightarrow p_i \)
- Why does this imply that such a path cannot exist in the STG?

Goal of Refinement

- The goal of refinement is to prevent the model checker from giving the same counterexample trace as before
  - In our example, the cex trace is \( 0 \rightarrow 1 \rightarrow 3 \)
  - This corresponds to executing the loop zero times
- How do we find predicates that will rule out this spurious trace?
- Most basic idea: Compute strongest postcondition for each statement in the cex trace; add these to set of predicates!

Craig Interpolant

Given an unsatisfiable formula \( \varphi_1 \land \varphi_2 \), a Craig interpolant is a formula \( \psi \) such that:
1. \( \varphi_1 \Rightarrow \psi \)
2. \( \text{UNSAT}(\varphi_2 \land \psi) \)
3. \( \psi \) is over the common variables of \( \varphi_1, \varphi_2 \)
Interpolant Examples

Consider the following formulas:
\[ \varphi_1 \equiv x \leq w \land y \geq w \land z = x \]
\[ \varphi_2 \equiv y < t \land t = z \]

- Which of the following formulas are interpolants for \( \varphi_1 \land \varphi_2 \)?
  1. \( y \geq z \)
  2. \( y \geq x \land z = x \)
  3. \( y > z \)

Why Are Interpolants Useful for Abstraction Refinement?

- Consider a spurious counterexample trace:
  \( l_0 \rightarrow s_1 \rightarrow l_1 \rightarrow s_2 \cdots \rightarrow s_n \rightarrow l_0 \)
- For simplicity, suppose the trace is in SSA form and suppose \( \text{enc}(s_i) \) gives logical encoding of \( s_i \)'s semantics
- Then, we know that the following formula is UNSAT:
  \[ \text{enc}(s_1) \land \text{enc}(s_2) \land \cdots \land \text{enc}(s_n) \]
- Now let \( \varphi_i^- \) denote the trace formula up to statement \( i \) and \( \varphi_i^+ \) denote the formula after \( i \)
- Then, for each location \( l_i \), we have \( \text{UNSAT}(\varphi_i^- \land \varphi_i^+) \) and the interpolant gives predicates that are useful to track at \( l_i \)

Do Interpolants Always Exist?

- **Result from William Craig**: For first-order formulas, such an interpolant always exists (1957).
- Furthermore, this result extends to first-order theories :)  
  - However, even if \( \varphi_1, \varphi_2 \) are quantifier-free, the interpolant may use quantifiers  
  - But for some theories (e.g., LRA, LIA), the interpolant is always quantifier-free if original formula is quantifier-free
- Some SAT and SMT solvers can give you interpolants of unsatisfiable formulas (beyond scope for this class)

Example

- Consider the following counterexample trace that corresponds to executing loop body once:
  \[
  \begin{align*}
  x0 := 0; & \quad y0:=0; \\
  \text{assume}(x0<100); & \quad \text{assume}(x0\geq100); \\
  x1 := x0 + 1; & \quad y1 := y0 + 1; \\
  \text{assume}(x1\geq100); & \quad \text{assume}(y1!=100); \\
  \end{align*}
  \]
  \[ \varphi_1 \equiv x_0 = 0 \land y_0 = 0 \land x_0 < 100 \land x_1 = x_0 + 1 \land y_1 = y_0 + 1 \]
  \[ \varphi_2 \equiv x_1 \geq 100 \land y_1 \neq 100 \]
- **Interpolant**: \( x_1 = y_1 \land x_1 \leq 100 \)
- Using the predicates in the interpolant, we can now verify the correctness of this program!

Per-location Abstraction

- In the basic form of predicate abstraction, we have a global set of predicates that we "track" everywhere
  - But not all predicates are useful everywhere...
- **Observation**: The interpolant tells us which predicates are useful where!
- Thus, rather than having a global set of predicates, we can have a different predicate set for each different location
  - Since the model checker is very sensitive to the number of predicates, this is really important for scalability

CEGAR Summary

- Can both verify and give counterexamples, but no termination guarantees...