Predicate Abstraction and Counterexample-Guided Abstraction Refinement

Işıl Dillig

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 $\mathcal{P} = \{p_1, \ldots, p_n\}$, 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 $\{0, 1, *\}$ where $*$ indicates unknown
- In other words, we have an abstract domain where each element is a formula $\bigwedge_i l_i$ (sometimes called a cube)

Predicate Abstraction Lattice

- Given predicates $\mathcal{P}$, $(Cubes(\mathcal{P}), \Rightarrow)$ forms a complete lattice
  - $Cubes(\mathcal{P})$ is any formula $\bigwedge_i p_i$ where p_i is a predicate or the negation of a predicate in $\mathcal{P}$
- In other words, we have $\varphi_1 \sqsubseteq \varphi_2$ iff $\varphi_1 \Rightarrow \varphi_2$
  - e.g., $p_1 \land p_2 \sqsubseteq p_1$
- Top element is *true*, and bottom is *false*
- Question: How do we compute $\varphi_1 \sqcup \varphi_2$?
- What is $(p_1 \land p_2) \sqcup p_1$?
- What is $(p_1 \land p_2) \sqcup \neg p_1$?

Abstract Transformers

- We defined our abstract domain, but still need abstract transformers
- Given a statement S and cube φ, define abstract transformer $post^\#(S, \varphi)$ to be the strongest cube φ' over $\mathcal{P}$ such that:

$$sp(S, \varphi) \Rightarrow \varphi'$$

  where sp is the strongest post-condition of S wrt to φ
- Example: Suppose $\mathcal{P} = \{x = y, x \neq y, x \geq y\}$. What is $post^\#(x := x + 1, x = y)$?

Example

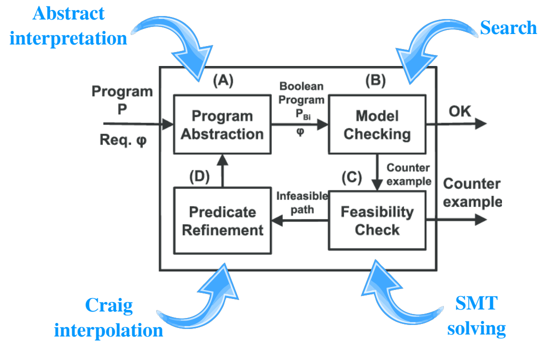
- Consider the program shown on the right
- And the predicate set $\mathcal{P} = \{x \leq 100, x = y, y = 100\}$
- Compute the abstraction of this program wrt to $\mathcal{P}$

```
x:=0; y:= 0;
while(x<100)
{
   x := x+1;
   y := y+1;
}
assert(y = 100);
```

Motivation for CEGAR

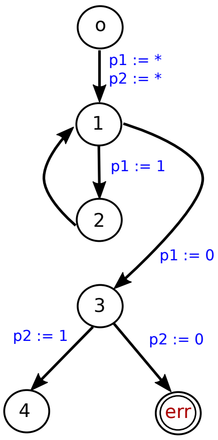
- Predicate abstraction is very sensitive to the set of predicates
- If you choose the right set, verification succeeds; otherwise, it fails
- The CEGAR paradigm allows automatically and iteratively discovering the right set of predicates

CEGAR



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

Motivation for Boolean Programs

- In general, the STG for a program is infinite
- ... but a model checker will only terminate if the STG is finite
- To ensure that STG is finite, construct a so-called boolean program via predicate abstraction
- All variables are booleans, so for a program with k locations and n variables, the size of the STG is at most $k \times 2^n$

Generating Boolean Programs

- Given a set of predicates $\mathcal{P}$ and program S, we want to generate a boolean program S' that has $|\mathcal{P}|$ 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 $wp(s, \neg p)$ 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 $\mathcal{P}$ such that $P_1 \Rightarrow wp(s, p)$ and $P_2 \Rightarrow wp(s, \neg p)$
- Thus model statement s as:

```
if(P1) b := true
else if(P2) b := false
else b := *
```

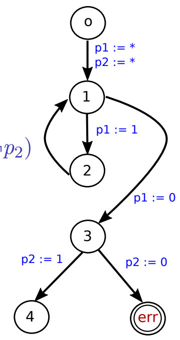

Example

- 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

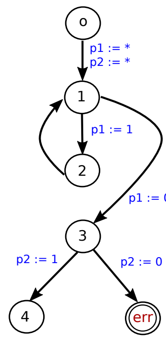
- Now that we have boolean programs, we can construct a finite STG
- For this program, there are four initial states:

  $(0, p_1, p_2), (0, p_1, \neg p_2), (0, \neg p_1, p_2), (0, \neg p_1, \neg p_2)$
- There is a transition from $(l, b_1, \ldots, b_n)$ to $(l', b'_1, \ldots, b'_n)$ iff:
  - There must be a transition from l to l' labeled with S
  - The formula $sp(S, \bigwedge_i b_i) \wedge \bigwedge_i b'_i$ must be satisfiable (query SAT solver!)



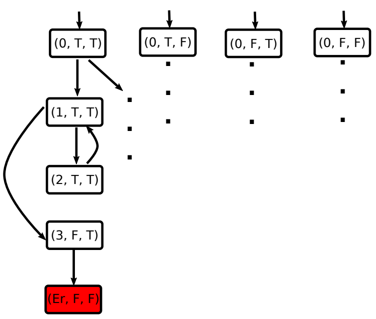
STG for Our Boolean Program, cont.

- Which of these transition exist in the state transition graph?
  - $(1, p_1, p_2)$ to $(3, \neg p_1, p_2)$
  - $(1, p_1, p_2)$ to $(3, p_1, p_2)$
  - $(3, \neg p_1, p_2)$ to $(err, \neg p_1, \neg p_2)$



STG for Our Boolean Program, cont.

- Partial STG for our program:



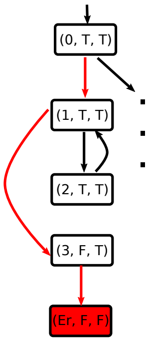
- Verification fails because error state is reachable!

Need for Refinement

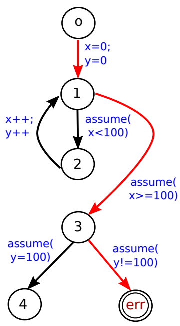
- To make the STG finite, we worked with boolean programs
- But if the error state is reachable, this could be due to imprecision in the abstraction
  - i.e., current set of predicates may not be fine-grain enough
- To decide how to proceed, we need to check if the property is **actually** violated
-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

Back to Running Example

STG:



CFA:



Cex trace:

```
x := 0; y:=0;
assume(x>=100);
assume(y!=100);
```

Clearly spurious because the trace formula is UNSAT:

$x = 0 \wedge y = 0 \wedge$
$x \geq 100 \wedge y \neq 100$

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 $0 \to 1 \to 3$
  - 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!

Eliminating Counterexamples using SP

- Let $l_0 \to^{s_1} l_1 \to^{s_2} \ldots \to^{s_n} l_n$ be a spurious cex trace
- Let p_0 be $true$, 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:

$$(l_0, \varphi_0) \to^{s_1} (l_1, \varphi_1) \to^{s_2} \ldots \to^{s_n} (l_n, \varphi_n)$$

- 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?

Proof

For any STG path $(l_0, \varphi_0) \to^{s_1} (l_1, \varphi_1) \to^{s_2} \ldots \to^{s_n} (l_n, \varphi_n)$**, we have** $\varphi_i \Rightarrow p_i$

- Base case: Trivial since p_0 is true
- Induction: By the IH, we have $\varphi_{i-1} \Rightarrow p_{i-1}$
- By STG construction, $(l_{i-1}, \varphi_{i-1}) \to^{s_i} (l_i, \varphi_i)$ exists if:

  $SAT(sp(s_i, \varphi_{i-1}) \wedge \varphi_i)$ which implies $SAT(sp(s_i, p_{i-1}) \wedge \varphi_i)$
- Furthermore, we have either $\varphi_i \Rightarrow p_i$ or $\varphi_i \Rightarrow \neg p_i$ – why?
- But if $\varphi_i \Rightarrow \neg p_i$, we'd have $UNSAT(sp(s_i, p_{i-1}) \wedge \varphi_i)$
- Thus, $\varphi_i \Rightarrow p_i$

Back to the Big Picture

- Using sp's in the cex trace removes the current counterexample
- ... but only removes this counterexample trace
- Ideally, we want to learn predicates that allow us to remove multiple spurious traces
- **Trick:** We can learn more general predicates using a technique called **Craig interpolation**

Craig Interpolant

Given an unsatisfiable formula $\varphi_1 \wedge \varphi_2$, a Craig interpolant is a formula ψ such that:

1. $\varphi_1 \Rightarrow \psi$
2. UNSAT$(\varphi_2 \wedge \psi)$
3. ψ is over the common variables of φ_1, φ_2

Interpolant Examples

Consider the following formulas:

$$\begin{aligned} \varphi_1 &\equiv x \leq w \land y \geq w \land z = x \\ \varphi_2 &\equiv y < t \land t = z \end{aligned}$$

- 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. $y > z$

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 φ_1, φ_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)

Why Are Interpolants Useful for Abstraction Refinement?

- Consider a spurious counterexample trace:

$$l_0 \to^{s_1} l_1 \to^{s_2} \ldots \to^{s_n} l_n$$

- For simplicity, suppose the trace is in SSA form and suppose $enc(s_i)$ gives logical encoding of s_i's semantics
- Then, we know that the following formula is UNSAT:

$$enc(s_1) \land enc(s_2) \land \ldots enc(s_i)$$

- Now let φ_i^- denote the trace formula up to statement i and φ_i^+ denote the formula after i
- Then, for each location l_i, we have $UNSAT(\varphi_i^- \land \varphi^+)$ and the **interpolant gives predicates that are useful to track at l_i!**

Example

- Consider the following counterexample trace that corresponds to executing loop body once:

```
x0  := 0; y0:=0;
assume(x0<100);
x1  := x0 + 1;
y1  := y0 + 1;
assume(x1>=100);
assume(y1!=100);
```

$$\boxed{\begin{aligned} &x_0 = 0 \land y_0 = 0 \land x_0 < 100 \\ &\land\ x_1 = x_0 + 1 \land y_1 = y_0 + 1 \end{aligned}} \quad \varphi_1$$

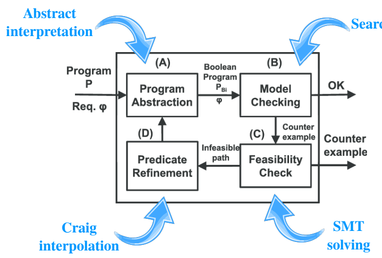
$$\boxed{x_1 \geq 100 \land y_1 \neq 100} \quad \varphi_2$$

- **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...