Invariant Generation

• Tools such as Dafny enable automated program verification by
  – automatically generating verification conditions and
  – automatically checking validity of the generated VCs.
• The user still needs to provide the invariants.
  – This is often the hardest part.
• Can we generate invariants automatically?
Partial Correctness of Programs

- Initial states $Q_{init}$
- Reachable states $Q_{reach}$
- Error states $Q_{err}$

State space $Q$
Partial Correctness of Programs

- Initial states: $Q_{\text{init}}$
- Reachable states: $Q_{\text{reach}}$
- Safe invariant: $Q_{I}$
- Error states: $Q_{\text{err}}$

State space $Q$
Inductive Invariants for Example Program

1: assume $y \geq z$;
2: while $x < y$ do
   $x := x + 1$;
3: assert $x \geq z$

Safe inductive invariant:

$p_c = \ell_1 \lor$
$p_c = \ell_2 \land y \geq z \lor$
$p_c = \ell_3 \land y \geq z \land x \geq y \lor$
$p_c = \ell_{exit}$
Predicate Abstraction

• construct abstraction $\alpha$ using a given set of building blocks, so-called predicates
• predicate = formula over program variables $V$
• fix finite set of predicates $Preds = \{p_1, ..., p_n\}$
• over-approximate $F$ by conjunction of predicates in $Preds$

$$\alpha(F) = \land\{ p \in Preds \mid F \models p \}$$

• computation of $\alpha(F)$ requires $n$ theorem prover calls ($n =$ number of predicates)
Predicate Abstraction

\[ p_1 \equiv x \leq 0 \quad p_2 \equiv y > 0 \quad \ldots \]

reachable states

\[ \text{reach} \]

\[ p_1 \land p_2 \land \ldots \]

x:0,y:5

x:-1,y:3

\[ \text{reach}\# \]

\[ x:0,y:3 \]

x:1,y:5

\[ \neg p_1 \land p_2 \land \ldots \]

state space \( Q \)

error states

error
Abstract Reachability Graph

$p_1 \equiv x \leq 0 \quad p_2 \equiv y > 0 \quad \ldots$

reachable states

reach

$p_1 \land p_2 \land \ldots$

x:0,y:5

invariant

reach^#

x:=x+1

x:1,y:5

$\neg p_1 \land p_2 \land \ldots$

state space $Q$

error states

error
Abstract Reachability Graph

- **Preds** = \{false, \ pc = \ell_1, ..., \ pc = \ell_{\text{err}}, \ y \geq z, \ x \leq y\}
- nodes \ F_1, ..., \ F_4 \in Q^\#_{\text{reach}}
- labeled edges \in Tree
- dotted edge: entailment relation (here: \text{post}^#(\rho_2, \ F_2) \vdash F_2

\begin{align*}
F_1: & \ pc = \ell_1 \\
F_2: & \ pc = \ell_2 \land y \geq z \\
F_3: & \ pc = \ell_3 \land y \geq z \land x \geq y \\
F_4: & \ pc = \ell_{\text{exit}} \land y \geq z \land x \geq y
\end{align*}

\begin{align*}
F_1 &= \alpha(\text{init}) \\
F_2 &= \text{post}^#(\rho_1, \ F_1) \\
\text{post}^#(\rho_2, \ F_2) &\vdash F_2 \\
F_3 &= \text{post}^#(\rho_3, \ F_2) \\
F_4 &= \text{post}^#(\rho_5, \ F_3)
\end{align*}
Abstract Reachability Graph

with $\text{Preds} = \{\text{false}, \; \text{pc} = \ell_1, \; \ldots, \; \text{pc} = \ell_{\text{err}}, \; y \geq z\}$

$F_1: \; \text{pc} = \ell_1$

$F_2: \; \text{pc} = \ell_2 \land y \geq z$

$F_3: \; \text{pc} = \ell_3 \land y \geq z$

$F_4: \; \text{pc} = \ell_{\text{err}} \land y \geq z$

$F_5: \; \text{pc} = \ell_{\text{exit}} \land y \geq z$

$F_1 = \alpha(\text{init})$

$F_2 = \text{post}^\#(\rho_1, F_1)$

$\text{post}^\#(\rho_2, F_2) \models F_2$

$F_3 = \text{post}^\#(\rho_3, F_2)$

$F_4 = \text{post}^\#(\rho_4, F_3)$

$F_5 = \text{post}^\#(\rho_5, F_3)$
Too Coarse Abstraction

<table>
<thead>
<tr>
<th>reachable states $reach$</th>
<th>invariant $reach^#$</th>
<th>state space $Q$</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>error states</td>
</tr>
<tr>
<td></td>
<td></td>
<td>error</td>
</tr>
</tbody>
</table>
Refinement of Predicate Abstraction

• given formulas $F_1, F_2, F_3, F_4$ such that

$$init \models F_1$$
$$post(\rho_1, F_1) \models F_2$$
$$post(\rho_3, F_2) \models F_3$$
$$post(\rho_4, F_3) \models F_4$$
$$F_4 \land error \models false$$

• add atoms of $F_1, ..., F_4$ to $Preds$.

• refinement guarantees that counterexample path $\rho_1, \rho_3, \rho_4$ is eliminated.
function AbstRefineLoop
begin
  Preds := ∅;
repeat
  (reach#, Tree) := AbstReach(Preds)
  if exists F ∈ reach# such that F ∧ error ≠ false then
    path := MakePath(F, Tree)
    if FeasiblePath(path) then
      return "counterexample path: path"
    else
      Preds := Preds ∪ RefinePath(path)
    else
      return "program is safe"
  end
end
Craig Interpolation

Given: an unsatisfiable conjunction of formulas $A \land B$

A Craig interpolant for $A \land B$ is a formula $I$ such that

- $A$ implies $I$
- $I \land B$ is unsatisfiable
- all free variables in $I$ are shared between $A$ and $B$
Interpolation-Based CEGAR

[McMillan’03, ...]

assume $0 \leq y$;

$x := 0$;

$z := n$;

while $x < y$ do

$x := x + 1$;

$z := z - 1$;

assert $z = n - y$;

Interpolant for $A \land B$

$z_1 = n_\theta - x_1 \land x_1 \leq y_\theta$

$\begin{align*}
A & : \begin{cases}
\theta \leq y_\theta \\
x_\theta = 0 \\
z_\theta = n_\theta \\
x_\theta < y_\theta \\
x_1 = x_\theta + 1 \\
z_1 = z_\theta - 1 \\
x_1 < y_\theta \\
x_2 = x_1 + 1 \\
z_2 = z_1 - 1 \\
x_2 \geq y_\theta \\
z_2 \neq n_\theta - y_\theta
\end{cases} \\
B & : \begin{cases}
\theta \leq y_\theta \\
x_\theta = 0 \\
z_\theta = n_\theta \\
x_\theta < y_\theta \\
x_1 = x_\theta + 1 \\
z_1 = z_\theta - 1 \\
x_1 < y_\theta \\
x_2 = x_1 + 1 \\
z_2 = z_1 - 1 \\
x_2 \geq y_\theta \\
z_2 \neq n_\theta - y_\theta
\end{cases}
\end{align*}$

1. iter.

2. iter.
Interpolation Procedures

- first-order predicate logic admits interpolation
- typically interested in quantifier-free interpolants
- many theories admit quantifier-free interpolation
  - linear arithmetic over $\mathbb{Z}$ and $\mathbb{Q}$
  - free function symbols with equality
  - ...
- interpolants computed from refutation proofs
- implemented in a number of Satisfiability Modulo Theory (SMT) solvers
Interpolation for Linear Arithmetic over Rationals

• Literals $L$ : (strict) linear inequalities
  \[0 \leq c_0 + c_1x_1 + \ldots + c_nx_n\]
  where
  – $c_0, \ldots, c_n$ are constants (rational numbers)
  – $x_1, \ldots, x_n$ are variables (denoting rationals)

• Clause $C$ : set of literals

• Sequents $C \models D$ where
  – $C$ and $D$ are clauses
  – $C \models D$ means $\bigwedge \{L \mid L \in C\} \models \bigvee \{L \mid L \in D\}$
A Simple Proof System

\[
\begin{align*}
\text{HYP} & \quad \frac{C \vdash L}{L \in C} \\
\text{COMB} & \quad \frac{C \vdash 0 \leq s \quad C \vdash 0 \leq t}{C \vdash 0 \leq c_1 s + c_2 t} \quad c_{1,2} > 0 \\
\text{CONTRA} & \quad \frac{L_1, \ldots, L_n \vdash 0 \leq c}{C \vdash \neg L_1, \ldots, \neg L_n} \quad c < 0 \\
\text{RES} & \quad \frac{C \vdash L, D \quad C \vdash \neg L, D'}{C \vdash D, D'}
\end{align*}
\]
Interpolants from Proofs

• Interpolated Sequent \((A,B) \vdash C[I]\) where
  
  – \(A,B,C\) are clauses and \(I\) is a formula
  – \(A \vdash I\)
  – \(B,I \vdash C\)
  – \(fv(I) \subseteq fv(A) \cap fv(B) \cup fv(C)\)
Interpolating Proof System

HYP-A \[ (A,B) \vdash L [L] \quad L \in A \]

HYP-B \[ (A,B) \vdash L [\top] \quad L \in B \]

COMB \[ (A,B) \vdash 0 \leq s [0 \leq s'] \quad (A,B) \vdash 0 \leq t [0 \leq t'] \quad c_{1,2} > 0 \]
\[ (A,B) \vdash 0 \leq c_1 s + c_2 t [0 \leq c_1 s' + c_2 t'] \]
Interpolating Proof System

**CONTRA**

\[
\frac{(\{a_1, \ldots, a_n\}, \{b_1, \ldots, b_m\}) \vdash 0 \leq c \; [I]}{(A, B) \vdash \lnot a_1, \ldots, \lnot a_n, \lnot b_1, \ldots, \lnot b_m \; [I \lor \lnot a_1 \lor \ldots \lor \lnot a_n]} \quad c < 0
\]

**RES-A**

\[
\frac{(A, B) \vdash L, D \; [I] \quad (A, B) \vdash \lnot L, D' \; [I']}{(A, B) \vdash D, D' \; [I \lor I']} \quad L \text{ not occurs in } B
\]

**RES-B**

\[
\frac{(A, B) \vdash L, D \; [I] \quad (A, B) \vdash \lnot L, D' \; [I']}{(A, B) \vdash D, D' \; [I \land I']} \quad L \text{ occurs in } B
\]
Example

• Computing an interpolant for \((A,B)\) where
  \[ A \equiv 0 \leq y - x \land 0 \leq z - y \]
  \[ B \equiv 0 \leq x - z - 1 \]
Automated Debugging
Faulty Shell Sort

Program

- takes a sequence of integers as input
- returns the sorted sequence.

On the input sequence 11, 14 the program returns 0, 11 instead of 11,14.
Error Trace

0 int i, j, a[];
1 int size=3;
2 int h=1;
3 h = h*3+1;
4 assume !(h<=size);
5 h/=3;
6 i=h;
7 assume (i<size);
8 v=a[i];
9 j=i;
10 assume !(j>=h && a[j-h]>v);
11 i++;
12 assume (i<size);
13 v=a[i];
14 j=i;
15 assume (j>=h && a[j-h]>v);
16 a[j]=a[j-h];
17 j-=h;
18 assume (j>=h && a[j-h]>v);
19 a[j]=a[j-h];
20 j-=h;
21 assume !(j>=h && a[j-h]>v);
22 assume (i!=j);
23 a[j]=v;
24 i++;
25 assume !(i<size);
26 assume (h==1);
27 assert a[0] == 11 && a[1] == 14;
Idea of our Approach

• Consider reachability in a finite automaton

  \[ q_0 \xrightarrow{a} q_1 \xrightarrow{a} q_4 \xrightarrow{a} q_f \]

  \[ q_2 \xrightarrow{b} q_1 \xrightarrow{d} q_4 \]

  \[ q_3 \xrightarrow{c} q_2 \]

• A word that witnesses the reachability of \( q_f \):
  \[ a \ b \ c \ d \ a \ b \ b \ b \ a \]

• We can eliminate loops to obtain a simpler witness:
  \[ a \ a \ a \]
Fault Localization in Programs

Apply this idea to programs:

• error trace = finite word of program statements

• program = automaton that accepts error traces (state/transition graph)

• fault localization = eliminate loops in the error trace
State/Transition Graph

- **Nodes**: states
- **Edges**: transitions

Program states

$x := x + y$
Predicate Abstraction

\[ \varphi: x=0 \land \ldots \]
\[ x:0,y:5 \bullet \]
\[ x:0,y:6 \bullet \]

\[ \psi: x\neq 0 \land \ldots \]
\[ x:1,y:5 \bullet \]

program states
Predicate Abstraction

$$\varphi: x=0 \land \ldots$$

$$x:0, y:5$$

$$x:=x+1$$

$$x:1, y:5$$

$$\psi: x \neq 0 \land \ldots$$
Predicate Abstraction

\( \varphi: x = 0 \land \ldots \)

\( x:0, y:5 \)

\( x:0, y:6 \)

\( x = 0 \Rightarrow \ldots \)

\( y := y + 1 \)

\( \psi: x \neq 0 \land \ldots \)

\( x := x + 1 \)

\( y := y + 1 \)
Abstraction-Based Fault Localization

reachable states

program states

error states

a
b
c
d

a
b

a
Abstraction-Based Fault Localization

Need a suitable notion of state equivalence:

Two states are equivalent if, from both states, the trace can reach an error state “for the same reason”.
Error Invariants

An error invariant $I$ for a position $i$ in an error trace $\tau$ is a formula over program variables s.t.

- all states reachable by executing the prefix of $\tau$ up to position $i$ satisfy $I$

- all executions of the suffix of $\tau$ that start from $i$ in a state that satisfies $I$, still lead to the error.
Error Invariants

Execution of trace

- $x = 0 \land y = 0 \land a = -1$
- $x = 0, y = 0, a = -1$
- $x = 1, y = 0, a = -1$
- $x = 1, y = 1, a = -1$
- $x = 0, y = 1, a = -1$

Information provided by the error invariants:

- Statement $y := y + 1$ is irrelevant
- Variable $y$ is irrelevant
- Variable $a$ is irrelevant after position 2
Error invariants are not unique

Execution of trace

x=0 ∧ y=0 ∧ a=-1

x=0, y=0, a=-1
0: x := x + 1;

x=1, y=0, a=-1
1: y := y +1;

x=1, y=1, a=-1
2: x := x + a;

x=0, y=1, a=-1
x > 0

We are interested in **inductive** error invariants!
Checking Error Invariants

Input Values  \rightarrow\ Control-Flow Path  \rightarrow\ Expected Outcome

Precondition  \land\ Path Formula  \land\ Postcondition

\$\land\ \land\ st_0 \land \ldots \land st_i \land \ldots \land st_n\$
Error Trace Formula

Example

Initial state:
\[ x=0 \land y=0 \land a=-1 \]

Operations:
1. \[ x := x + 1; \]
2. \[ y := y + 1; \]
3. \[ x := x + a; \]

Result:
\[ x > 0 \]

Transformed state:
\[ x_0=0 \land y_0=0 \land a_0=-1 \]

Operations:
1. \[ x_1 = x_0 + 1 \land \]
2. \[ y_1 = y_0 + 1 \land \]
3. \[ x_2 = x_1 + a_0 \land \]

Result:
\[ x_2 > 0 \]
Checking Error Invariants

Input Values

Control-Flow Path

Expected Outcome

Precondition

Path Formula
\( st_0 \land ... \land st_i \land ... \land st_n \)

Postcondition

\( I \) is an error invariant for position \( i \) iff

\[ A \models I \quad \text{and} \quad I \land B \models \bot \]
Craig Interpolants are Error Invariants

Craig interpolant for $A \land B$ is an error invariant for position $i$

$\Rightarrow$ use Craig interpolation to compute candidates for inductive error invariants.
Computing Abstract Error Traces

Basic Algorithm:
1. Compute the error trace formula from the error trace.
2. Compute a Craig interpolant $I_i$ for each position $i$ in the error trace.
3. Compute the error invariant matrix:
   - for each $I_i$ and $j$, check whether $I_i$ is an error invariant for $j$.
4. Choose minimal covering of error trace with inductive error invariants.
5. Output abstract error trace.
Error Invariant Matrix for Faulty Shell Sort

| \( \setminus \text{st} \) | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | 27 |
|-------------------------|---|---|---|---|---|---|---|---|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 0                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 1                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 2                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 3                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 4                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 5                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 6                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 7                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 8                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 9                       |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 10                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 11                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 12                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 13                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 14                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 15                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 16                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 17                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 18                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 19                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 20                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 21                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 22                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 23                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 24                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 25                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 26                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
| 27                      |   |   |   |   |   |   |   |   |   |   |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |    |
Abstract Error Trace for Faulty Shell Sort

0 int i,j, a[];
1 int size=3;
2 int h=1;
3 h = h*3+1;
4 assume !(h<=size);
5 h/=3;
6 i=h;
7 assume (i<size);
8 v=a[i];
9 j=i;
10 assume !(j>=h && a[j-h]>v);
11 i++;
12 assume (i<size);
13 v=a[i];
14 j=i;
15 assume (j>=h && a[j-h]>v);
16 a[j]=a[j-h];
17 j=h;
18 assume (j>=h && a[j-h]>v);
19 a[j]=a[j-h];
20 j=h;
21 assume !(j>=h && a[j-h]>v);
22 assume (i!=j);
23 a[j]=v;
24 i++;
25 assume !(i<size);
26 assume (h==1);
27 assert a[0] == 11 && a[1] == 14;