Software Verification

Lecture 5: Assertion Inference

Carlo A. Furia
Bertrand Meyer
Proving Programs Automatically

The Program Verification problem:
- Given: a program $P$ and a specification $S = [\text{Pre}, \text{Post}]$
- Determine: if any execution of $P$, for any value of input parameters, satisfies $S$
- Equivalently: establish whether $\{\text{Pre}\} P \{\text{Post}\}$ is (totally) correct

- A general and fully automated solution to the Program Verification problem is unachievable because the problem is undecidable

- One of the consequences of this inescapable limitation is the impossibility of computing verification conditions (VC) fully automatically
  - VC: assertions used in the correctness proof
  - (It is not an obvious consequence: formally, a reduction between undecidable problems)
The Program Verification problem:

- **Given**: a program \( P \) and a specification \( S = [\text{Pre}, \text{Post}] \)
- **Determine**: if any execution of \( P \), for any value of input parameters, satisfies \( S \)
- **Equivalently**: establish whether \( \{\text{Pre}\} \ P \ {\text{Post}} \) is (totally) correct

Practically:

Proving the correctness of a computer program requires knowledge about the program that is not readily available in the program text

--- Chang & Leino

In this lecture, we survey techniques to automatically infer assertions in interesting special cases
Correctness is consistency of implementation to specification

The paradox:

if the specification is inferred from the implementation, what do we prove?
The **paradox**: if the specification is inferred from the implementation, what do we prove?

Possible **retorts**:

- The paradox only arises for correctness proofs; there are other applications (e.g. reverse-engineering legacy software)
- The result may be presented to a programmer for **assessment**
- Inferred specification may be **inconsistent**, thus denoting a problem
The Assertion Inference Paradox

The paradox:
if the specification is inferred from the implementation, what do we prove?

The paradox does not arise if we only infer VC (i.e., intermediate assertions) and not specifications (pre and postconditions)

- VC are a technical means to an end (proving correctness)
  - the tool infers loop invariants
- The specification is a formal description of what the implementation should do
  - the programmer writes the specification
Invariants

Let us consider a general (and somewhat informal) definition of invariant:

**Def. Invariant:** assertion whose truth is preserved by the execution of (parts of) a program.

\[ x: \text{INTEGER} \]
\[ \text{from } x := 1 \text{ until ... loop } x := -x \text{ end} \]

Some invariants:
- \(-1 \leq x \leq 1\)
- \(x = -1 \lor x = 0 \lor x = 1\)
- \(x \geq -10\)
Kinds of Invariants

Def. Invariant: assertion whose truth is preserved by the execution of (parts of) a program.

We can identify different families of invariants, according to what parts of the program preserve the invariant:

- Location invariant at x: assertion that holds whenever the computation reaches location x
- Program invariant: predicate that holds in any reachable state of the computation
- Class invariant: predicate that holds between (external) feature invocation
- Loop invariant: predicate that holds after every iteration of a loop body

\[
\begin{align*}
\{P\} & A \{I\} \\
\{I \land \neg c\} & B \{I\} \\
\hline
\{P\} & \text{ from } A \text{ until } c \\
\text{loop } B \text{ end} & \{I \land c\}
\end{align*}
\]
Kinds of Invariants

1: \( x: \text{INTEGER} \)
2: 
3: from \( x := 1 \)
4: until ...
5: loop \( x := -x \) end

- Location invariant at 2:
  \( x = 0 \)

- Loop invariant:
  \( x = -1 \text{ v } x = 1 \)

- Program invariant:
  \( x \geq -10 \)
Focus on Loop Invariants

The various kinds of invariants are closely related by the inference rules of Hoare logic.

- If $L x$ is a location invariant at $x$ then:
  \[ @x \Rightarrow L x \]
  is a program invariant.

- If $P$ is a program invariant then it is also a location invariant at every location $x$.

- If $I$ is a loop invariant of:
  \[ x : \text{from ... until } c \text{ loop ... end} \]
  then $I \land c$ is a location invariant at $x+1$.

- If $L$ is a location invariant at $x+1$:
  \[ x : a := b + 3 \]
  then $L \ [b + 3 / a]$ is a location invariant at $x$.

- Etc...
Focus on Loop Invariants

If we have loop invariants we can get (almost) everything else at little cost

- but not vice versa:
  getting loop invariants requires invention

In the following discussion we focus on loop invariants (and call them simply “invariants”)

This focus is also consistent with the Assertion Inference Paradox
Techniques for Invariant Inference

Classification of invariant inference techniques:

• **Dynamic** techniques
• **Static** techniques
  - statistical techniques
  - exact techniques

(Roughly) direction of increasing:
- soundness
- completeness
- mathematical sophistication

Classification is neither sharp nor complete, yet *useful*
Exact Static Techniques for Invariant Inference
Static Invariant Inference: classification

Static exact techniques for invariant inference are further classified in categories:

- **Direct**
- **Assertion-based**
  - postcondition weakening
- **Based on abstract interpretation**
  - forward analysis (bottom-up)
  - backward analysis (top-down)
- **Constraint-based**
  - usually, template-based

Further reading: Bradley & Manna, 2007
Exact Static Techniques for Invariant Inference:

Postcondition-weakening Approach
The Role of User-provided Contracts

• None of the other techniques for invariant inference takes advantage of other annotations in the program text, such as contracts provided by the user
  – Not every annotation can (or should, cf. Assertion Inference Paradox) be inferred automatically!

• However, there is a close connection between a loop’s invariant and its postcondition
  • The invariant is a weakened form of the postcondition
    – e.g., Post: \( x = n \) Invariant: \( x \leq n \)
  • In the “constructive approach” to programming, this observation guides the construction of a loop invariant from a postcondition before the program is written
    – Edsger Dijkstra, David Gries, Tony Hoare, …

    A program and its proof should be developed hand-in-hand, with the proof usually leading the way — Gries, 1981
Constructive Programming: Example

- **Goal:** computing the *square root* of input *a*
  - Pre: $a > 0$
  - Post: $Result \times Result = a$

- **Get the invariant from Post**
  - modify Post by *uncoupling* the two occurrences of Result:
    $$Result \times y = a \quad \text{and} \quad y = Result$$
  - Invariant: $Result \times y = a$
  - Exit condition: $y = Result$
Constructive Programming: Example

- **Pre:**  \( a > 0 \)
- **Post:**  \( \text{Result} \times \text{Result} = a \)
- **Invariant:**  \( \text{Result} \times y = a \)
- **Exit condition:**  \( y = \text{Result} \)

```plaintext
square_root (a: REAL): REAL
require Pre end
local y: REAL
do
  from -- Establish Invariant
  until -- Exit condition
  loop
    -- Bring y closer to Result
    -- Restore Invariant
  end
ensure Post end
```

```plaintext
square_root (a: REAL): REAL
require \( a > 0 \) end
local y: REAL
do
  from \( \text{Result} := a ; \ y := 1 \)
  until \( y = \text{Result} \)
  loop
    \( \text{Result} := (\text{Result} + y) / 2 \)
    \( y := a / \text{Result} \)
  end
ensure \( \text{Result} \times \text{Result} = a \) end
```
**Loops as Approximation Strategy**

- **Postcondition:** \( \text{Result} = \max \{a_1, a_2, ..., a_n\} \)

- **Initially:** \( \text{Result} = \max \{a_i\} \)

- **After i iterations:** \( \text{Result} = \max \{a_1, a_2, ..., a_i\} \)
  
  - This is the abstract loop invariant
  
  - Loop body: \( \text{Result} := \max (\text{Result}, a[i]) ; \ i := i + 1 \)
Loops as problem-solving strategy

A loop invariant is a property that:

- Is easy to **establish initially**
  - even if only to cover a **trivial** part of the data
- Is easy to **extend** to cover a bigger part
- If covering all data, gives the **desired result**
Invariants by Postcondition Weakening

- In a nutshell:

  static verification of candidate invariants obtained by weakening postconditions

1. Assume the availability of postconditions
2. Weaken postconditions according to various heuristics
   - the heuristics mirror common patterns that link postconditions to invariants
   - each weakened postcondition is a candidate invariant
3. Verify which candidates are indeed invariants
   - Typically with an automatic program prover such as Boogie
   - Checking is much simpler than inferring the invariant out of the blue
4. Retain all verified invariants

- 2009 -- C.A. Furia & B. Meyer
- Implementation: gin-pink which finds invariants in Boogie programs
Postcondition Weakening Heuristics

- **Constant relaxation**
  - replace constant by variable
    - “constant”: unchanged by the loop body
    - “variable”: changed by the loop body

- **Uncoupling**
  - replace variable appearing twice by two variables

- **Term dropping**
  - remove a term (usually a conjunct)

- **Variable aging**
  - use expression representing previous value
Maximum Value of an Array

```plaintext
max (A: ARRAY [T]; n: INTEGER): T
  require A.length = n >= 1
  local i: INTEGER
  do
    from i := 0; Result := A[1];
    until i >= n
  loop
    i := i + 1
    if Result <= A[i] then Result := A[i] end
  end
  ensure \( \forall 1 \leq j \leq n \Rightarrow A[j] \leq Result \)
```

- Loop modifies \( i \) ("variable")
- Loop doesn't modify \( n \) ("constant")
- Constant relaxation: replace constant \( n \) by variable \( i \) in postcondition to get candidate invariant
- Invariant: \( \forall 1 \leq j \leq i \Rightarrow A[j] \leq Result \)
Maximum Value of an Array (2\textsuperscript{nd} version)

```plaintext
max_v2 (A: ARRAY [T]; n: INTEGER): T
    require A.length = n \geq 1
    local i: INTEGER
    do
        from i := 1; Result := A[1];
        until i > n
        loop
            if Result \leq A[i] then Result := A[i] end
            i := i + 1
        end
    ensure \( \forall 1 \leq j \leq n \Rightarrow A[j] \leq Result \)
```

- **Constant relaxation**: replace constant \( n \) by variable \( i \) in postcondition: \( \forall 1 \leq j \leq i \Rightarrow A[j] \leq Result \)

- **Variable aging**: use expression representing the previous value of \( i \): \( i - 1 \)

- **Invariant**: \( \forall 1 \leq j \leq i - 1 \Rightarrow A[j] \leq Result \)
Array Partitioning

\textbf{partition} (A: ARRAY [T]; n: INTEGER; pivot: T): INTEGER

\begin{verbatim}
require A.length = n \geq 1
local l, h: INTEGER
do
  from l := 1 ; h := n until l = h
  loop
    from until l = h or A[l] > pivot loop l := l + 1 end
    from until l = h or pivot > A[h] loop h := h - 1 end
    A.swap (l, h)
  end
if pivot \leq A[l] then l := l - 1 end ; h := l ; Result := h
ensure (\forall 1 \leq k \leq Result \Rightarrow A[k] \leq pivot) \land (\forall Result < k \leq n \Rightarrow A[k] \geq pivot)
\end{verbatim}

- **Uncoupling**: replace first occurrence of \textbf{Result} by \textbf{l} and second by \textbf{h}

  \[(\forall 1 \leq k \leq l \Rightarrow A[k] \leq pivot) \land (\forall h < k \leq n \Rightarrow A[k] \geq pivot)\]

- **Variable aging**: use expression representing the previous value of \textbf{l}: \textbf{l} - 1

- **Invariant**: \[(\forall 1 \leq k \leq l - 1 \Rightarrow A[k] \leq pivot) \land (\forall h < k \leq n \Rightarrow A[k] \geq pivot)\]
**Array Partitioning**

\[
\text{partition} (A: ARRAY [T]; n: INTEGER; pivot: T): INTEGER \\
\text{require } A.length = n \geq 1 \\
\text{local } l, h: INTEGER \\
\text{do} \\
\hspace{1em} \text{from } l := 1; h := n \text{ until } l = h \\
\hspace{2em} \text{loop} \\
\hspace{3em} \text{from until } l = h \text{ or } A[l] > pivot \text{ loop } l := l + 1 \text{ end} \\
\hspace{3em} \text{from until } l = h \text{ or } pivot > A[h] \text{ loop } h := h - 1 \text{ end} \\
\hspace{2em} A.swap (l, h) \\
\hspace{1em} \text{end} \\
\hspace{1em} \text{if } pivot \leq A[l] \text{ then } l := l - 1 \text{ end}; h := l; \text{Result} := h \\
\text{ensure } (\forall 1 \leq k \leq \text{Result} \Rightarrow A[k] \leq pivot) \land (\forall \text{Result} < k \leq n \Rightarrow A[k] \geq pivot)
\]

- **Term dropping:** remove first conjunct in postcondition:
  \[
  \forall \text{Result} < k \leq n \Rightarrow A[k] \geq pivot
\]

- **Constant relaxation:** replace constant \text{Result} by variable \text{h}

- **Invariant:**
  \[
  \forall h < k \leq n \Rightarrow A[k] \geq pivot
\]
Invariant Inference: the Algorithm

- **Goal**: find invariants of loops in procedure `proc`
- For each:
  - `post`: postcondition clause of `proc`
  - `loop`: outer loop in `proc`

  compute all weakenings $W$ of `post` w.r.t. `loop`
  
  - considering postcondition clauses separately implements term dropping

- **Result**: any formula in $W$ which can be verified as invariant of any loop in `proc`
Building Weakening of Postcondition

- **Goal**: build all weakenings $W$ of postcondition post w.r.t. loop
- **Add** post (unchanged) to $W$
- For each pair (constant, variable) such that:
  - constant is any subexpression of post which is unchanged by loop
  - variable is any variable (possibly) changed by loop
- **Result**: set of computed weakenings $W$
Constant Relaxation without Uncoupling

• Goal: build all weakenings $W$ of postcondition post w.r.t. loop where constant is relaxed into variable

• Replace every occurrence of constant in post by variable and add the result to $W$
  • this implements constant relaxation (without uncoupling)

• Compute the previous value old of variable w.r.t. loop
  • this implements variable aging

• Replace every occurrence of constant in post by old and add the result to $W$

• Result: set of computed weakenings $W$
Constant Relaxation with Uncoupling

• **Goal**: build all weakenings $W$ of postcondition post w.r.t. loop where constant is relaxed into variable

• Replace individually each occurrence of constant in post by variable and add the results to $W$
  • this implements constant relaxation with uncoupling

• Compute the previous value old of variable w.r.t. loop
  • this implements variable aging

• Replace individually each occurrence of constant in post by old and add the results to $W$

• Result: set of computed weakenings $W$
Implementation: gin-pink

gin-pink: Generation of INvariants by PostcondItioN weakening

• written in Eiffel
• command-line tool
  - Boogie in / Boogie out
• works with any high-level language that can be translated to Boogie
• distributed under GPL
  - invariant candidate generation works on every system
  - checking of candidates calls Boogie,
    hence currently available only under Microsoft Windows
• available for download from http://se.inf.ethz.ch/people/furia/
### Experiments on Literature Examples

<table>
<thead>
<tr>
<th>Procedure name</th>
<th>LOC</th>
<th># candid.</th>
<th># invar.</th>
<th># relev.</th>
<th>% relev.</th>
<th>Time (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Array Partitioning (v1)</td>
<td>58 (22)</td>
<td>38</td>
<td>9</td>
<td>3</td>
<td>33</td>
<td>93</td>
</tr>
<tr>
<td>Array Partitioning (v2)</td>
<td>68 (40)</td>
<td>45</td>
<td>2</td>
<td>2</td>
<td>100</td>
<td>205</td>
</tr>
<tr>
<td>Array Stack Reversal</td>
<td>147 (34)</td>
<td>134</td>
<td>4</td>
<td>2</td>
<td>50</td>
<td>529</td>
</tr>
<tr>
<td>Array Stack Reversal (ann.)</td>
<td>147 (34)</td>
<td>134</td>
<td>6</td>
<td>4</td>
<td>67</td>
<td>516</td>
</tr>
<tr>
<td>Bubble Sort</td>
<td>69 (29)</td>
<td>14</td>
<td>2</td>
<td>2</td>
<td>100</td>
<td>65</td>
</tr>
<tr>
<td>Coincidence Count</td>
<td>59 (29)</td>
<td>1351</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>4304</td>
</tr>
<tr>
<td>Dutch National Flag</td>
<td>77 (43)</td>
<td>42</td>
<td>10</td>
<td>2</td>
<td>20</td>
<td>117</td>
</tr>
<tr>
<td>Dutch National Flag (ann.)</td>
<td>77 (43)</td>
<td>42</td>
<td>12</td>
<td>4</td>
<td>33</td>
<td>122</td>
</tr>
<tr>
<td>Max of Array (v1)</td>
<td>27 (17)</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>30</td>
</tr>
<tr>
<td>Max of Array (v2)</td>
<td>27 (17)</td>
<td>7</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>16</td>
</tr>
<tr>
<td>Plateau</td>
<td>53 (29)</td>
<td>31</td>
<td>6</td>
<td>3</td>
<td>50</td>
<td>666</td>
</tr>
<tr>
<td>Sequential Search (v1)</td>
<td>34 (26)</td>
<td>45</td>
<td>9</td>
<td>5</td>
<td>56</td>
<td>120</td>
</tr>
<tr>
<td>Sequential Search (v2)</td>
<td>29 (21)</td>
<td>24</td>
<td>6</td>
<td>6</td>
<td>100</td>
<td>58</td>
</tr>
<tr>
<td>Shortest Path</td>
<td>57(44)</td>
<td>23</td>
<td>2</td>
<td>2</td>
<td>100</td>
<td>53</td>
</tr>
<tr>
<td>Stack Search</td>
<td>196 (49)</td>
<td>102</td>
<td>3</td>
<td>3</td>
<td>100</td>
<td>300</td>
</tr>
<tr>
<td>Sum of Array</td>
<td>26 (15)</td>
<td>13</td>
<td>1</td>
<td>1</td>
<td>100</td>
<td>44</td>
</tr>
<tr>
<td>Welfare Crook</td>
<td>53 (21)</td>
<td>20</td>
<td>2</td>
<td>2</td>
<td>100</td>
<td>586</td>
</tr>
</tbody>
</table>
Exact Static Techniques for Invariant Inference:

Constraint-based Approach
Constraint-based Invariant Inference

• In a nutshell:
  encode semantics of iteration as constraints on a template invariant

  1. Choose a template invariant expression
     • template defines a (infinite) set of assertions
  2. Encode the loop semantics as a set of constraints on the template
     • initiation
     • consecution
  3. Solve the constraints
     • this is usually the complex part
  4. Any solution is an invariant

• 2003 -- Henny Sipma et al.
• 2004 -- Zohar Manna et al.
• 2007 -- Tom Henzinger et al.
• ...
Farkas's Lemma (1902)

Let $S$ be a system of linear inequalities over $n$ real variables:

$$
S \triangleq \begin{bmatrix}
    a_{11}x_1 + \cdots + a_{1n}x_n + b_1 & \leq & 0 \\
    \vdots & \vdots & \vdots \\
    a_{m1}x_1 + \cdots + a_{mn}x_n + b_m & \leq & 0
\end{bmatrix}
$$

and let $\Psi$ be a linear inequality:

$$
\Psi \triangleq c_1x_1 + \cdots + c_nx_n + d \leq 0
$$

Then $S \Rightarrow \Psi$ is valid iff $S$ in unsatisfiable or there exist $m+1$ real nonnegative coefficients $\lambda_0, \lambda_1, \ldots, \lambda_m$ such that:

$$
c_j = \sum_{i=1}^{m} \lambda_i a_{ij} \quad (1 \leq j \leq m) \quad d = -\lambda_0 + \sum_{i=1}^{m} \lambda_i b_i
$$
Constraint-based Inv. Inference: Example

Use Farkas's lemma to turn:

\[ T \left[ \frac{x}{x}; \frac{n}{n} \right] \land x < n \land x' = x + 1 \land n' = n \]

\[ \Rightarrow T \left[ \frac{x'}{x}; \frac{n'}{n} \right] \]

into a constraint over \( c, d, \) and \( e \) only.

\[
\begin{array}{l|ccc}
\mu_1 & cx & +dn & +e & \leq 0 \\
\lambda_1 & x & -n & +1 & \leq 0 \\
\lambda_2 & -x & +x' & -1 & \leq 0 \\
\lambda_3 & x & -x' & +1 & \leq 0 \\
\lambda_4 & -n & +n' & \leq 0 \\
\lambda_5 & n & -n' & \leq 0 \\
\hline
& cx' & +dn' & +e & \leq 0 \\
\end{array}
\]
Constraint-based Inv. Inference: Example

dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end

• Template invariant expression:

\[ T = c \cdot x + d \cdot n + e \leq 0 \]

• Constraints encoding loop semantics:

• Initiation: “T holds for the initial values of x and n”

\[ T [0/x; n_0/n] \equiv c \cdot 0 + d \cdot n_0 + e \leq 0 \equiv d \cdot n_0 + e \leq 0 \]
Constraint-based Inv. Inference: Example

```
dummy Routine (n: NATURAL)
local x: NATURAL do
    from x := 0
    until x ≥ n
    loop x := x + 1 end
end
```

- Constraints encoding loop semantics:
  - Consecution: “if T holds and one iteration of the loop is executed, T still holds”
    \[ T [x/x; n/n] \land ( \neg (x ≥ n) \land x' = x + 1 \land n' = n ) \Rightarrow T [x'/x; n'/n] \]

- Solving the constraints requires to eliminate occurrences of x, x', n, n'
  - For linear constraints we can use Farkas's Lemma
**Constraint-based Inv. Inference: Example**

<table>
<thead>
<tr>
<th></th>
<th>(\mu_1)</th>
<th>(cx)</th>
<th>(+dn)</th>
<th>(+e)</th>
<th>(\leq 0)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\lambda_1)</td>
<td>(x)</td>
<td>(-n)</td>
<td>(+1)</td>
<td>(\leq 0)</td>
<td></td>
</tr>
<tr>
<td>(\lambda_2)</td>
<td>(-x)</td>
<td>(+x')</td>
<td>(-1)</td>
<td>(\leq 0)</td>
<td></td>
</tr>
<tr>
<td>(\lambda_3)</td>
<td>(x)</td>
<td>(-x')</td>
<td>(+1)</td>
<td>(\leq 0)</td>
<td></td>
</tr>
<tr>
<td>(\lambda_4)</td>
<td>(-n)</td>
<td>(+n')</td>
<td>(\leq 0)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>(\lambda_5)</td>
<td>(n)</td>
<td>(-n')</td>
<td>(\leq 0)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

\[
\Phi \triangleq \exists \mu_1, \lambda_0, \ldots, \lambda_5
\]

\[
\begin{bmatrix}
\mu_1, \lambda_0, \ldots, \lambda_5 & \geq 0 \\
\mu_1 c + \lambda_1 - \lambda_2 + \lambda_3 & = 0 \\
\mu_1 d - \lambda_1 - \lambda_4 + \lambda_5 & = 0 \\
\lambda_2 - \lambda_3 & = c \\
\lambda_4 - \lambda_5 & = d \\
-\lambda_0 + \mu_1 e + \lambda_1 - \lambda_2 + \lambda_3 & = e
\end{bmatrix}
\]
Constraint-based Inv. Inference: Example

\[ \Phi \triangleq \exists \mu_1, \lambda_0, \ldots, \lambda_5 \begin{bmatrix} \mu_1, \lambda_0, \ldots, \lambda_5 \geq 0 \\ \mu_1 c + \lambda_1 - \lambda_2 + \lambda_3 = 0 \\ \mu_1 d - \lambda_1 - \lambda_4 + \lambda_5 = 0 \\ \lambda_2 - \lambda_3 = c \\ \lambda_4 - \lambda_5 = d \\ -\lambda_0 + \mu_1 e + \lambda_1 - \lambda_2 + \lambda_3 = e \end{bmatrix} \]

Finally, eliminate existential quantifiers from \( \Phi \) to get the constraint:

\[ c \leq 0 \lor (c + d = 0 \land e \leq 0) \]

- (Quantifier elimination is also quite technical)
Constraint-based Inv. Inference: Example

```
dummy_routine(n: NATURAL)
local x: NATURAL do
    from x := 0
    until x ≥ n
    loop x := x + 1 end
end
```

- Any solution \([c, d, e] \) to:
  - Initiation and Consecution:
    
    \[
    (d \cdot n_0 + e ≤ 0) \land (c ≤ 0 \lor (c + d = 0 \land e ≤ 0))
    \]
    
    determines an invariant of the loop.

- Examples:
  - \([0, -1, 0]\) \quad \longrightarrow \quad n ≥ 0
  - \([1, 0, 0]\) \quad \longrightarrow \quad x ≥ 0
  - \([1, -1, 0]\) \quad \longrightarrow \quad x - n ≤ 0
  - \([0, -1, n_0]\) \quad \longrightarrow \quad n = n_0
Constraint-based Inv. Inference: Summary

• Main issues:
  - choice of invariant templates for which effective decision procedures exist
    - interesting research topic per se, on the brink of undecidability
  - heuristics to extract the “best” invariants from the set of solutions

• Advantages:
  - sound
  - complete (w.r.t. the template)
  - exploit heterogeneous decision procedures syncretically
  - fully automated (possibly except for providing the template)
    - providing the template introduces a “natural” form of interaction with the user

• Disadvantages:
  - suitable mathematical decision theories are usually quite sophisticated
    - hence, hard to extend and customize
  - exact constraint solving is usually quite expensive
  - mostly suitable for “algebraic” invariants
    - requires integration with other techniques to achieve full functional correctness proofs
Dynamic Techniques for Invariant Inference
Dynamic Invariant Inference

• In a nutshell:
  testing of candidate invariants

  1. Choose a set of test cases
  2. Perform runtime monitoring of candidate invariants
  3. If some test run violates a candidate, discard the candidate
  4. The surviving candidates are guessed invariant

• Daikon tool, 1999 -- Mike Ernst et al.
• CITADEL: Daikon for Eiffel, 2008 -- Nadia Polikarpova
• ...
Dynamic Invariant Inference: Example

\textbf{dummy\_routine} (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end

- Test cases: \{ n = k \mid 0 ≤ k ≤ 1000 \}
- Candidate invariants:
  - \{ x ≥ c \mid -1000 ≤ c ≤ 1000 \},
  \{ n ≥ c \mid -1000 ≤ c ≤ 1000 \}
  - \{ x = c \cdot n + d \mid -500 ≤ c, d ≤ 500 \}
  - \{ x < n, x ≤ n, x = n, x ≠ n, x ≥ n, x > n \}
  - \{ x ± n ≥ c \mid -500 ≤ c ≤ 500 \}
  - ...
Dynamic Invariant Inference: Example

dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end

• Survivors (after loop iterations):
  - \{ x ≥ -c  |  0 ≤ c ≤ 1000 \},
    \{ n ≥ -c  |  0 ≤ c ≤ 1000 \}
  - x ≤ n
  - \{ x + n ≥ c  |  -500 ≤ c ≤ 500 \}
  - ...


Dynamic Invariant Inference: Summary

- **Main issues:**
  - choose suitable test cases
  - handle huge sets of candidate invariants (runtime overhead)
  - estimate soundness/quality of survivor predicates
  - select heuristically the “best” survivor predicates

- **Advantages:**
  - straightforward to implement (at least compared to other techniques)
  - guessing is often rather accurate in practice (possibly with some heuristics)
  - customizable and rather flexible:
    - in principle, whatever you can test you can check for invariance

- **Disadvantages:**
  - unsound (educated guessing)
  - without heuristics, large amount of useless, redundant predicates
  - sensitive to choice of test cases
  - some complex candidate invariants are difficult to implement efficiently
APPENDIX – Additional Material
Exact Static Techniques
for Invariant Inference:

Direct Approach
Direct Static Invariant Inference

• In a nutshell:

  solve the fixpoint equations underlying the program

1. $v(i)$: value of variable $v$ at step $i$ of the computation

2. Encode the semantics of loops explicitly and directly as
   recurrence equations over $v(i)$

3. Solve recurrence equations

4. Eliminate step parameter $i$ to obtain invariant

• 1973 -- Shmuel Kats & Zohar Manna

• ...
Direct Static Invariant Inference: Example

dummy_routine (n: NATURAL)
local x: NATURAL do
  from x := 0
  until x ≥ n
  loop x := x + 1 end
end

• \(x(i), n(i)\)

• Recurrence relations:

\[
x(i) = \begin{cases} 
0 & i = 0 \\
x(i - 1) + 1 & 0 < i \leq n_0 \\
x(i - i) & i > n_0 
\end{cases} \quad n(i) = \begin{cases} 
n_0 \geq 0 & i = 0 \\
n(i - i) & i > 0 
\end{cases}
\]
Direct Static Invariant Inference: Example

\[ x(i) = \begin{cases} 
  0 & i = 0 \\
  x(i-1) + 1 & 0 < i \leq n_0 \\
  x(i - i) & i > n_0
\end{cases} \]

\[ n(i) = \begin{cases} 
  n_0 & i = 0 \\
  n(i - i) & i > 0
\end{cases} \]

- Solving recurrence relations:
  - \( x(i) = \min(n_0, i) \geq 0 \)
  - \( n(i) = n_0 \)

- Eliminating step parameter \( i \):
  - \( x(i) - n(i) = \min(n_0, i) - n_0 \leq 0 \), hence:
  - \( x - n \leq 0 \), hence:
  - \( 0 \leq x \leq n \)
Direct Static Invariant Inference: Summary

- **Main issues:**
  - more a set of guidelines than a technique
  - step parameter elimination is tricky

- **Advantages:**
  - since semantics is represented explicitly, obtained invariants are often powerful
  - benefits from the programmer's ingenuity
  - additional information about the program can be “plugged in”

- **Disadvantages:**
  - solving recurrence equations can be very difficult (when possible at all)
  - unsuitable technique to automate
Exact Static Techniques for Invariant Inference:

Approach Based on Abstract Interpretation
Abstract Interpretation for Invariants

• In a nutshell:
  symbolic execution over an abstract domain
  with guarantee of termination

  1. Consider the over-approximation of the value of variables over some coarse abstract domain (instead of their exact values)
  2. Symbolically execute the program over the abstract domain
  3. Iterate loops until termination
     • termination guaranteed by the nature of the abstract domain
       or by heuristic cut-offs (widening)
  4. The final expression is an invariant

• 1976 -- Michael Karr
• 1977, 1978 -- Patrick & Radhia Cousot, Nicolas Halbwachs
• ...
Abstract Interpretation for Inv.: Example

dummy_routine (n: NATURAL)
  local x: NATURAL do
    from x := 0
    until x ≥ n
    loop x := x + 1 end
  end

- Abstract interval domain:
  conjunction of inequalities in the form
  \[ v ≤ c \text{ or } c ≤ v \]
  for any program variable \( v \) and integer constant \( c \)
- Initially: \( S(0) ≜ \{ 0 ≤ x, x ≤ 0, 0 ≤ n \} \)
- After one loop iteration: \( S(1) ≜ \{ 1 ≤ x, x ≤ 1, 0 ≤ n \} \)
- Set of abstract states reached in at most one loop iteration:
  \( S(0) \lor S(1) = \{ 0 ≤ x, x ≤ 1, 0 ≤ n \} \)
Abstract Interpretation for Inv.: Example

dummyRoutine (n: NATURAL)
local x: NATURAL do
from x := 0
until x $\geq$ n
loop x := x + 1 end
end

• Initially: $S(0) \triangleq \{0 \leq x, x \leq 0, 0 \leq n\}$

• Set of abstract states reached in at most one loop iteration:
  $S(0) \lor S(1) = \{0 \leq x, x \leq 1, 0 \leq n\}$

• $S(0) \lor S(1)$ does not subsume $S(0)$
  • no fixpoint, keep on iterating

• Abstract states after at most $k$ loop iterations:
  $S(0) \lor \ldots \lor S(k) = \{0 \leq x, x \leq k, 0 \leq n\}$

• No convergence as: $S(0) \lor \ldots \lor S(k)$ does not subsume $S(0) \lor \ldots \lor S(k-1)$
Abstract Interpretation for Inv.: Example

\hspace{10mm} \textit{dummy\_routine} \ (n: NATURAL)
\hspace{10mm} \textit{local} \ x: NATURAL \ \textit{do}
\hspace{20mm} \textit{from} \ x := 0
\hspace{20mm} \textit{until} \ x \geq n
\hspace{20mm} \textit{loop} \ x := x + 1 \ \textit{end}
\hspace{10mm} \textit{end}

\begin{itemize}
  \item Abstract states after at most \( k \) loop iterations:
    \[ S(0) \lor \ldots \lor S(k) = \{ 0 \leq x, x \leq k, 0 \leq n \} \]
  \item Apply \textit{heuristic over-approximation}:
    \begin{itemize}
      \item \textit{relax} \ S(0) \lor \ldots \lor S(k) \textit{by dropping inequalities with growing bounds:}
        \[ S' = \text{widen} \{ 0 \leq x, x \leq k, 0 \leq n \} \Rightarrow \{ 0 \leq x, 0 \leq n \} \]
      \item \( S' \) is a \textit{fixpoint} of the loop iteration
    \end{itemize}
  \item \( 0 \leq x \land 0 \leq n \) is a \textit{loop invariant}
  \begin{itemize}
    \item \textit{a very weak one, but more sophisticated choices of abstract domain and/or heuristic over-approximation would yield the “desired”} \( 0 \leq x \leq n \)
  \end{itemize}
\end{itemize}
Abstract Interpretation for Inv.: Summary

• Main issues:
  - effective choice of abstract domain
    * trade off: accuracy vs. computational efficiency
  - smart choice of heuristic widening

• Advantages:
  - the abstract interpretation framework is quite general and customizable to many different program properties
  - fully automated
  - sound
  - scalable: efficient implementations are possible

• Disadvantages:
  - incompleteness, from two sources:
    * invariants can be inexpressible in the abstract domain
    * even if they are expressible, heuristic widening loses completeness
  - requires integration with other techniques to achieve full functional correctness proofs
Statistical Static Techniques for Invariant Inference
Statistical Static Invariant Inference

The goal of the analysis is usually different than for other classes of invariant inference techniques:

- inferring likely specific behavioral specification (e.g., temporal properties)
- inferring likely violations of invariance behavioral properties
- no functional invariants

Examples of inferred behavioral specs:
- Location $x$ performs allocation ($A$) of some resource
- Location $y$ performs deallocation ($D$) of some resource
- Every allocated resource is eventually deallocated
Statistical Static Invariant Inference

- In a nutshell:
  - learning of a statistical model

1. **Classify** all possible behaviors (w.r.t. goal properties) of code
2. Assign **prior probabilities** to different behaviors
   - possibly including additional knowledge
3. Compute **cumulative probabilities** of code
   - e.g., through simple symbolic execution
4. Report **likely inferred specifications** or likely invariance violations
   - those with the highest cumulative probabilities

- 2002 -- James Larus et al.
- ...
Statistical Invariant Inference: Example

- **Prior probabilities:**
  - \( \Pr[\text{green}] = 0.9 \) (probability that the sequence is ok)
  - \( \Pr[\text{red}] = 0.1 \) (probability that the sequence gives a bug)

- **Cumulative probabilities, e.g.:**
  - \( \Pr[A, \neg D, D] = \prod f(A, \neg D, D) = 0.9 \times \ldots \times \ldots \)
    - \( f \) are the various elements of knowledge
Statistical Invariant Inference: Summary

- **Main issues:**
  - choose suitable behavioral properties
  - compute efficiently complex products of probabilities for a huge number of candidate behaviors
  - classify possible behaviors by their probabilities
  - introduce effective *ad hoc* factors

- **Advantages:**
  - reasonably robust w.r.t. the choice of prior probabilities
  - customizable: can incorporate very specific probability factors
  - can handle large "real" programs

- **Disadvantages:**
  - unsound
  - mostly limited to simple behavioral properties
  - best suited for "systems" code
    (where full functional correctness is usually not a concern)