Loop Invariants

Verification

- Consider a loop-free program $P$
  - With conditionals
  - Memory references
  - Data structures
  - No function calls

- What is the computational complexity of verifying
  
  \[
  \{ \text{Precondition} \} \hspace{1mm} P \hspace{1mm} \{ \text{Postcondition} \}
  \]

Loops

- Now consider the same problem
  - Where $P$ can have one loop
  - But still no function calls

- What is the computational complexity of verifying
  \[
  \{ \text{Precondition} \} \hspace{1mm} P \hspace{1mm} \{ \text{Postcondition} \}
  \]

Verification of Loops

- Verifying properties of loops is the hard problem

- Solve this, and everything else is much easier

A Simple Example

\[
\begin{align*}
X &= 0 \\
I &= 0 \\
\text{while } I < 10 \text{ do} & \\
& \hspace{1mm} X = X + 1 \\
& \hspace{1mm} I = I + 1 \\
\text{assert}(X == 10)
\end{align*}
\]

Loop Invariants

- To verify loops, it suffices to find a sufficiently strong loop invariant

- What is a loop invariant?
  - A predicate that holds on every loop iteration
  - (at the same program point, usually at loop head)

- What is "sufficiently strong"
  - More in a minute ...
Loop Invariant (1)

\[ X = 0 \]
\[ I = 0 \]
while I < 10 do
\[
\begin{align*}
& \{ \text{true} \} \\
& X = X + 1 \\
& I = I + 1
\end{align*}
\]
assert(X == 10)

Loop Invariant (2)

\[ Z = 42 \]
\[ X = 0 \]
\[ I = 0 \]
while I < 10 do
\[
\begin{align*}
& \{ Z = 42 \} \\
& X = X + 1 \\
& I = I + 1
\end{align*}
\]
assert(X == 10)

Loop Invariant (3)

\[ Z = 42 \]
\[ X = 0 \]
\[ I = 0 \]
while I < 10 do
\[
\begin{align*}
& \{ I < 4327 \} \\
& X = X + 1 \\
& I = I + 1
\end{align*}
\]
assert(X == 10)

Loop Invariant (4)

\[ Z = 42 \]
\[ X = 0 \]
\[ I = 0 \]
while I < 10 do
\[
\begin{align*}
& \{ X < 11 \} \\
& X = X + 1 \\
& I = I + 1
\end{align*}
\]
assert(X == 10)

Loop Invariant (5)

\[ Z = 42 \]
\[ X = 0 \]
\[ I = 0 \]
while I < 10 do
\[
\begin{align*}
& \{ X = I \& I < 11 \} \\
& X = X + 1 \\
& I = I + 1
\end{align*}
\]
assert(X == 10)

Comments

- Loop invariants aren't hard to compute
  - If you don't care about quality
    - true
  - What we want is to prove the assertion
    - Need an invariant strong enough to do this
Comments

- But how can we prove the assertion?
- We need a proof strategy
  - An algorithm that we can apply to any loop

Inductive Invariants

\[ \text{while (B)} \]
\[ \{ \]
\[ \ldots \text{code} \ldots \]
\[ \} \]
\[ \text{Post} \]
\[ I \wedge \neg B \Rightarrow \]
\[ Post \]

Pre \(\Rightarrow\) I
The invariant holds initially

I \wedge B \{ \text{code} \} I
If the invariant and loop condition hold, executing the loop body re-establishes the invariant

I \wedge \neg B \Rightarrow Post
If the invariant holds and the loop terminates, then the post-condition holds

Loop Invariant (1)

\[ X = 0 \]
\[ I = 0 \]
while I < 10 do
\[ \{ \text{true} \} \]
\[ X = X + 1 \]
\[ I = I + 1 \]
assert(X == 10)

Loop Invariant (2)

Z = 42
X = 0
I = 0
while I < 10 do
\[ \{ Z = 42 \} \]
\[ X = X + 1 \]
\[ I = I + 1 \]
assert(X == 10)

Loop Invariant (3)

Z = 42
X = 0
I = 0
while I < 10 do
\[ \{ I < 4327 \} \]
\[ X = X + 1 \]
\[ I = I + 1 \]
assert(X == 10)
Loop Invariant (4)

\[
\begin{align*}
Z &= 42 \\
X &= 0 \\
I &= 0 \\
\text{while } I < 10 &\text{ do} \\
&\quad \{ X < 11 \} \\
&\quad X = X + 1 \\
&\quad I = I + 1 \\
\text{assert}(X = 10)
\end{align*}
\]

Loop Invariant (5)

\[
\begin{align*}
Z &= 42 \\
X &= 0 \\
I &= 0 \\
\text{while } I < 10 &\text{ do} \\
&\quad \{ X = I \land I < 11 \} \\
&\quad X = X + 1 \\
&\quad I = I + 1 \\
\text{assert}(X = 10)
\end{align*}
\]

Invariant Inference

- An old problem
- A different approach with two ideas:
  1. Separate invariant inference from the rest of the verification problem

Why?

- Complementary to static analysis
  - underapproximations
  - "see through" hard analysis problems
    - functionality may be simpler than the code
- Possible to generate many, many tests

Nothing New Under the Sun

- Sounds like DAIKON?
  - Yes!
- Hypothesize (many) invariants
  - Run the program
  - Discard candidate invariants that are falsified
  - Attempt to verify the remaining candidates
A Simple Program

s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + 1;
    y := y + 1;
}

• Instrument loop head
• Collect state of program variables on each iteration

Profs. Aiken, Barrett & Dill
Lecture 12

A DAIKON-Like Approach

s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + 1;
    y := y + 1;
}

• Hypothesize
  - s=y
  - s=2y
• Data
<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Profs. Aiken, Barrett & Dill
Lecture 12

A DAIKON-Like Approach

s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + 1;
    y := y + 1;
}

• Hypothesize
  - s=y
  - s=2y
• Data
<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Profs. Aiken, Barrett & Dill
Lecture 12

Another Approach

s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + 1;
    y := y + 1;
}

• Data
<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Profs. Aiken, Barrett & Dill
Lecture 12

Arbitrary Linear Invariant

as + by = 0

• Data
<table>
<thead>
<tr>
<th>s</th>
<th>y</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

Profs. Aiken, Barrett & Dill
Lecture 12
Observation

\[ ax + by = 0 \]

<table>
<thead>
<tr>
<th>( x )</th>
<th>( y )</th>
<th>( s )</th>
<th>( a )</th>
<th>( b )</th>
<th>( t )</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>e</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

NullSpace(M)

\[ \{ w | Mw = 0 \} \]

Linear Invariants

- Construct matrix \( M \) of observations of all program variables
- Compute NullSpace(\( M \))
- All invariants are in the null space

Spurious "Invariants"

- All invariants are in the null space
  - But not all vectors in the null space are invariants
- Consider the matrix

  \[ \begin{pmatrix} s & y \\ 0 & 0 \end{pmatrix} \]

- Need a check phase
  - Verify the candidate is in fact an invariant

An Algorithm

- Check candidate invariant
  - If an invariant, done
  - If not an invariant, get counterexample
    - Counterexample can be guaranteed to satisfy all invariants
- Add new row to matrix
  - And repeat
Termination

- How many times can the solve & verify loop repeat?

- Each counterexample is linearly independent of previous entries in the matrix

- So at most $N$ iterations
  - Where $N$ is the number of columns
  - Upper bound on steps to reach a full rank matrix

Summary

- Superset of all linear invariants can be obtained by a standard matrix calculation

- Counter-example driven improvements to eliminate all but the true invariants
  - Guaranteed to terminate

What About Non-Linear Invariants?

```plaintext
s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + y;
y := y + 1;
}
```

Idea

- Collect data as before

- But add more columns to the matrix
  - For derived quantities
  - For example, $y^2$ and $s^2$

- How to limit the number of columns?
  - All monomials up to a chosen degree $d$

[Nguyen, Kapur, Weimer, Forrest 2012]

What About Non-Linear Invariants?

```plaintext
s = 0;
y = 0;
while( * )
{
    print(s,y);
    s := s + y;
y := y + 1;
}
```

Solve for the Null Space

```plaintext
a + bs + cy + ds^2 + ey^2 + fsy = 0
```

Candidate invariant: $-2s + y + y^2 = 0$
Comments

• Same issues as before
  - Must check candidate is implied by precondition, is
    inductive, and implies the postcondition on
    termination
  - Termination of invariant inference guaranteed if
    the verifier can generate counterexamples

• Experience: Solvers do well as checkers!

Summary to This Point

• Algorithm for algebraic invariants
  - Up to a given degree

• Guess and Check
  - Hard part is inference done by matrix solve
  - Check part done by standard SMT solver
  - Much simpler and faster than previous approaches

What About Disjunctive Invariants?

• Disjunctions are expensive
  - In addition to conjunctions

• Existing techniques severely restrict
disjunctions
  - E.g., to a template

What About Non-Numeric Invariants?

• Arrays?
• Lists?
• Other data structures?

• Invariant inference techniques are very
  specialized

A Search-Based Approach

• All methods for finding invariants are
  heuristics
  - Can never be complete

• So why not use general but incomplete
  techniques?
**MCMC**

- Markov Chain Monte Carlo sampling
- The only known tractable solution method for high dimensional irregular search spaces

**MCMC Overview**

**MCMC Sampling Algorithm for Invariants**

1. Select an initial candidate
2. Repeat (millions of times)
   - Propose a random modification and evaluate cost
   - If (cost decreased)
     - accept
   - If (cost increased)
     - with some probability accept anyway

**Recall**

\[ \text{Pre} \Rightarrow I \]

\[ I(s) \Rightarrow I(t) \text{ if } s \{\text{body}\} t \]

\[ I \land \neg B \Rightarrow \text{Post} \]

**Data**

- Good states \( G \)
  - Reachable states
- Pairs \( Z \)
  - States \((s,t)\) such that starting the loop body \( S \) in state \( s \) terminates in state \( t \).
- Bad states \( B \)
  - States that lead to an assertion violation

**Cost Function (Roughly)**

- Penalize a candidate invariant \( C \)
  - 1 for each good state \( g \) in \( G \) where \( C(g) \) is false.
  - 1 for each bad state \( b \) in \( B \) where \( C(b) \) is true
  - 1 for each pair \((s,t)\) in \( Z \) where \( C(s) \) and not \( C(t) \)
- The cost of \( C \) is the sum of the penalties
**Overall Algorithm**

- Run search until a 0-cost candidate $C$ is found
- Use a decision procedure to verify that $C$ is an invariant
  - If yes, done
  - If no, get a counterexample
    - A good state, bad state, or pair
    - Add to the data
    - Repeat

**MCMC Sampling Algorithm for Invariants**

1. Select an initial candidate
2. Repeat (millions of times)
   - Propose a random modification and evaluate cost
   - If (cost decreased)
     - accept
   - If (cost increased)
     - with some probability accept anyway

**Numerical Invariants**

- Find invariants of the form

\[
\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \sum_{k=1}^{n} w_k^{(i,j)} x_k \leq d^{(i,j)}
\]

**Moves**

- Replace a coefficient
- Replace a constant on the rhs
- Replace all coefficients and the constant in a single inequality

\[
\bigvee_{i=1}^{\alpha} \bigwedge_{j=1}^{\beta} \sum_{k=1}^{n} w_k^{(i,j)} x_k \leq d^{(i,j)}
\]

**Arrays**

- Use the fluid updates abstraction
- Reduce to search for numerical predicate $T$
  - But now involves universal quantifier
  - $f,g$ are array variables

\[
\forall u,v,T(x_1, x_2, \ldots, x_n, u, v) \Rightarrow f[u] = g[v]
\]
A Problem with Arrays

- Decision procedures for arrays cannot give us counterexamples
- Instead use executions to generate data
  - Including bad states and pairs

Generating Data

- Pick a number \( k \)
- At the loop head
  - Assign all numeric variables a value \( \leq k \)
  - Assign all arrays a size \( \leq k \)
  - Assign all elements of arrays a value \( \leq k \)
- For experiments, we used \( k = 4 \)

Results on Arrays

<table>
<thead>
<tr>
<th>Program</th>
<th>[?]</th>
<th>23-R</th>
<th>AMC</th>
<th>Dual</th>
<th>Past</th>
<th>NMC</th>
<th>Temp</th>
</tr>
</thead>
<tbody>
<tr>
<td>test-m</td>
<td>0.02</td>
<td>0.06</td>
<td>0.15</td>
<td>0.12</td>
<td>0.16</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>test-p</td>
<td>0.03</td>
<td>0.07</td>
<td>0.15</td>
<td>0.14</td>
<td>0.16</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>test-s</td>
<td>0.04</td>
<td>0.10</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>copy</td>
<td>0.02</td>
<td>0.08</td>
<td>0.16</td>
<td>0.20</td>
<td>0.22</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>copy-p</td>
<td>0.04</td>
<td>0.10</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>reverse</td>
<td>0.03</td>
<td>0.08</td>
<td>0.16</td>
<td>0.20</td>
<td>0.22</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>swap</td>
<td>0.12</td>
<td>0.15</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>d-loop</td>
<td>0.12</td>
<td>0.15</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>copy</td>
<td>0.07</td>
<td>0.09</td>
<td>0.16</td>
<td>0.20</td>
<td>0.22</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>stories</td>
<td>0.02</td>
<td>0.08</td>
<td>0.16</td>
<td>0.20</td>
<td>0.22</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>memory</td>
<td>0.04</td>
<td>0.10</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>find-s</td>
<td>0.02</td>
<td>0.08</td>
<td>0.16</td>
<td>0.20</td>
<td>0.22</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>find-s</td>
<td>0.05</td>
<td>0.10</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>append</td>
<td>0.07</td>
<td>0.10</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>range</td>
<td>0.09</td>
<td>0.12</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>alloc-r</td>
<td>0.02</td>
<td>0.08</td>
<td>0.16</td>
<td>0.20</td>
<td>0.22</td>
<td>0.02</td>
<td>0.01</td>
</tr>
<tr>
<td>alloc-r</td>
<td>0.05</td>
<td>0.10</td>
<td>0.20</td>
<td>0.26</td>
<td>0.28</td>
<td>0.02</td>
<td>0.01</td>
</tr>
</tbody>
</table>

Strings

- Search space is
  - Boolean combinations of predicates \( P \)
  - \( P \) consists of constants and predicates in the program

```plaintext
i:=0; x:="a*;
while(non_det())
  i++; x:='("x*x")';
assert(x.length == 2*i+1);
if(i>0) assert(x.contains("(a*)");
```

String Results

<table>
<thead>
<tr>
<th>Pure</th>
<th>replace</th>
<th>index</th>
<th>substring</th>
</tr>
</thead>
<tbody>
<tr>
<td>342.50</td>
<td>0.01</td>
<td>0.06</td>
<td>0.53</td>
</tr>
<tr>
<td>NMC</td>
<td>0.82</td>
<td>0.02</td>
<td>0.06</td>
</tr>
<tr>
<td>Z3-STR</td>
<td>0.03</td>
<td>TO</td>
<td>114.55</td>
</tr>
</tbody>
</table>

Lists

- Search space
  - Boolean combinations of atoms
  - Atoms are relations \( R(x_1, \ldots, x_n) \)
- Moves
  - Replace one argument of a relation
  - Replace an entire relation
  - Flip polarity of an atom
Lists

- Use one reachability relation

\[ n(x,y) = y \text{ is reachable from } x \text{ in 0 or more pointer dereferences} \]

List Results

<table>
<thead>
<tr>
<th>Program</th>
<th>#G</th>
<th>#B</th>
<th>Search</th>
<th>Valid</th>
<th>Prop.</th>
<th>Accept.</th>
</tr>
</thead>
<tbody>
<tr>
<td>delete</td>
<td>50</td>
<td>2</td>
<td>0.20</td>
<td>0.04</td>
<td>4437</td>
<td>9949</td>
</tr>
<tr>
<td>delete-all</td>
<td>20</td>
<td>7</td>
<td>1.03</td>
<td>0.13</td>
<td>8482</td>
<td>7225</td>
</tr>
<tr>
<td>find</td>
<td>50</td>
<td>9</td>
<td>0.42</td>
<td>0.04</td>
<td>6681</td>
<td>5560</td>
</tr>
<tr>
<td>filter</td>
<td>50</td>
<td>26</td>
<td>10.41</td>
<td>0.11</td>
<td>160489</td>
<td>126389</td>
</tr>
<tr>
<td>last</td>
<td>50</td>
<td>3</td>
<td>0.90</td>
<td>0.04</td>
<td>98064</td>
<td>87446</td>
</tr>
<tr>
<td>reverse</td>
<td>20</td>
<td>54</td>
<td>55.11</td>
<td>0.08</td>
<td>582665</td>
<td>484208</td>
</tr>
</tbody>
</table>

Summary

- Invariant inference is a hard problem, made easier by looking at data from executions
  - Because the executions satisfy all the invariants

- Search-based techniques can work
  - Competitive with other methods
  - Easier to retarget to new domains

- Still limited by decision procedures
  - But not by their ability to do inference