Dynamic Contract Inference

Nadia Polikarpova

Software Verification
19.10.2011
Dynamic contract inference

- Location invariant – a property that always holds at a given point in the program

  ... $x := 0$ ...

- Dynamic invariant inference – detecting location invariants from values observed during execution

- Also called: invariant generation, contract inference, specification inference, assertion inference, ...

- Pioneered by Daikon

  http://groups.csail.mit.edu/pag/daikon/
Overview

- How does Daikon work?
- Inferred invariants
- Improving inferred invariants
- Contract inference in Eiffel: CITADEL and AutoInfer
Daikon architecture

Source code

Instrumenter

Instrumented code

Execution

Test suite

Declarations

Trace

Detector

Inferred invariants

Annotated code

Formatted invariants

Postprocessor (printer, annotator, etc.)

Language-dependent
Daikon architecture

- Source code
  - Instrumenter
    - Instrumented code
      - Execution
        - Test suite
        - Trace
          - Detector
            - Inferred invariants
              - Postprocessor
                - Formatted invariants
                  - Annotated code
                    - Language-dependent
          - Declarative
            - Language-dependent
              - Source code
                - Instrumented code
                  - Instrumenter
                    - Test suite
Instrumenter

- Finds **program points** of interest
  - routine enter/exit, loop condition
- Finds **variables** of interest at these program points
  - current object, formals, locals, return value, expressions composed of other variables
- Modifies the source code so that every time a program point is executed, variable values are printed to the trace file
class BANK_ACCOUNT

   ... balance: INTEGER

   deposit (amount: INTEGER)
   do

       balance := balance + amount

   end

end
Daikon architecture

- **Source code**
- **Instrumenter**
  - Instrumented code
  - Declarations
  - Trace
  - Inferred invariants
  - Annotated code
  - Formatted invariants
- **Postprocessor** (printer, annotator, etc.)
- **Execution**
  - Test suite
  - Trace
- **Detector**
  - Inferred invariants

Language-dependent
Detector

- Has a predefined set of invariant **templates**
- At each program point instantiates the templates with appropriate variables
- Checks invariants against program point **samples** (variable values in the trace)
- Reports invariants that are not falsified (and satisfy other conditions)
Detector: example

- Templates: \( x = \text{const} \quad x \geq \text{const} \quad x = y \quad \ldots \)

- Program point: `BANK_ACCOUNT.deposit:::ENTER`

- Variables: `balance, amount`: INTEGER

- Invariants:
  - `balance = 0`
  - `balance \geq 0`
  - `amount = 10`
  - `amount \geq 1`
  - `balance = amount`

- Samples:
  - `balance` 0  `amount` 10
  - `balance` 10  `amount` 20
  - `balance` 30  `amount` 1
Unary invariant templates

- Constant
  \[ x = \text{const} \]

- Bounds
  \[ x < \text{const} (\leq, >, \geq) \]

- Nonzero
  \[ x \neq 0 \]

- Modulus
  \[ x = r \mod m \]

- No duplicates
  \[ s \text{ has no duplicates} \]

- Index and element
  \[ s [i] = i (\langle, \leq, >, \geq) \]
Binary invariant templates

- **Comparisons**
  \[ x = y \ (\lt, \leq, \gt, \geq) \]

- **Linear binary**
  \[ ax + by = 0 \]

- **Squared**
  \[ x = y^2 \]

- **Divides**
  \[ x = 0 \ \text{mod} \ y \]

- **Zero track**
  \[ x = 0 \ \text{implies} \ y = 0 \]

- **Member**
  \[ x \ \text{in} \ s \]

- **Reversed**
  \[ s_1 = s_2.\text{reversed} \]

- **Subsequence and subset**
  \[ s_1 \ \text{is subsequence of} \ s_2 \quad s_1 \ \text{is subset of} \ s_2 \]
- Linear ternary
  \[ ax + by + zc = 0 \]

- Binary function
  \[ z = f(x, y) \]
  where \( f = \text{and, or, xor, min, max, gcd, pow} \)
Daikoon architecture

Source code

Instrumenter

Instrumented code

Execution

Test suite

Trace

Trace

Declarations

Detector

Inferred invariants

Language-dependent

Postprocessor (printer, annotator, etc.)

Annotated code

Formatted invariants

Test suite

Source code

Instrumented code

Declarations

Trace

Inferred invariants

Annotated code

Instrumenter

Detector

Formated invariants

Language-dependent

Test suite
Annotates code with inferred invariants

```java
class BANK_ACCOUNT
...
  balance: INTEGER

 deposit (amount: INTEGER)

  do
    balance := balance + amount
  end
end
```

BANK_ACCOUNT.deposit:::ENTER
balance >= 0
amount >= 1
...
Results depend on...

- Source code
- Invariant templates
- Variables that instrumenter finds
  - potentially all expressions that can be evaluated at a program point
  - needs to choose interesting ones
- Test suite
- Fine tuning the detector
Dynamic inference is...

- Not **sound**
  - Sound over the test suite, but not potential runs
- Not **complete**
  - Restricted to the set of templates
  - Heuristics for eliminating irrelevant invariants might remove relevant ones
- Even if it was, it reports properties of the code, not the developers intent
Classification

- inferred invariants
- incorrect
- uninteresting
- relevant inferred invariants
- not inferred
- perfect specification
Quality measures

- **Correctness** – percentage of correct inferred invariants (true code properties)

- **Relevance** (precision) – percentage of relevant inferred invariants

- **Recall** – percentage of true invariants that were inferred
Using inferred invariants

- As a specification (after human inspection)
  - Strengthening and correcting human-written specifications
  - Inferring loop invariants that are difficult to construct manually
- Finding bugs
- Evaluating and improving test suites
Improving quality

- Improving relevance
  - Statistical test
  - Redundant invariants
  - Comparability analysis
- Improving recall
  - More templates and variables
  - Conditional invariants
Statistical test

- Checking invariant
  \[ x /= 0 \]

- Let samples of \( x \) be nonzero, distributed in \([-5, 5]\)
  - With 3 samples:
    \[ p_{by\_chance} = (1 - 1/11)^3 \approx 0.75 \]
  - With 100 samples:
    \[ p_{by\_chance} = (1 - 1/11)^{100} \approx 0.00007 \]

- Each invariant calculates probability in its own way
- Threshold is defined by the user (usually \(< 0.01\)
Redundant invariants

\[ x > 0 \]
\[ x \neq 0 \]

- Invariants that are implied by other invariants are not interesting

- How to find them?
  - General-purpose theorem prover
  - Daikon has built-in hierarchy of invariants (invariants know their suppressors)
Comparability analysis

class BANK_ACCOUNT

...  

invariant

\[ \text{number} > \text{owner.birth\_year} \]

end

- Using the same syntactic type (INTEGER) to represent multiple semantic types
- Semantics types can be recovered by static analysis
- Variables \( x \) and \( y \) are considered comparable if they appear in constructs like

\[
x = y \quad x := y \quad x > y \quad x + y \quad \ldots
\]
Improving recall

It is easy:

- add more invariant templates
- add more variables of interest

However that increases the search space and

- either makes inference intractable
- or decreases relevance

Choose templates and variables in a smart way

e.g. at the entry to `withdraw (amount: INTEGER)`

`is_amount_available (amount)` is a good choice but

`is_amount_available (5)` is not
Conditional invariants

- Invariants of the form
  \[(P_1 \text{ and } P_2 \ldots \text{ and } P_m) \text{ implies } Q\]
  are hard to infer with the basic technique: it has to try all combinations of \(P_i\) and \(Q\).

- An efficient way: Decision Tree Learning

\[
\begin{align*}
\text{old after} & \\
\text{False} & \Rightarrow \text{index} = \text{old index} \\
\text{True} & \Rightarrow \text{index} = \text{old index} + 1
\end{align*}
\]
Contract Inference Tool Applying Daikon to Eiffel Language

- Infers only contracts expressible in Eiffel
  - no invariants over sequences
- Uses zero-argument functions as variables
  - Eiffel functions are pure
  - user-supplied preconditions are used to check whether a function can be called
- Infers loop invariants
Experiment

- Comparing programmer-written contracts with inferred ones
- **Scope**: 25 classes (89–1501 lines of code)
  - 15 from industrial-grade libraries
  - 4 from an application used in teaching CS at ETH
  - 6 from student projects
- **Tests suite**: 50 calls to every method, random inputs + partition testing
- **Contract clauses total**:
  - programmer-written: 831
  - inferred: 9’349
Classification

Programmer-written

not expressible

not inferred

implied by inferred

Inferred

implied by programmer-written

both

new

uninteresting

incorrect
## Results

<table>
<thead>
<tr>
<th>Measure</th>
<th>Description</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Correctness</td>
<td>correct IC / IC</td>
<td>90%</td>
</tr>
<tr>
<td>Relevance</td>
<td>relevant IC / IC</td>
<td>64%</td>
</tr>
<tr>
<td>Expressibility</td>
<td>PC expressible in Daikon / PC</td>
<td>86%</td>
</tr>
<tr>
<td>Recall</td>
<td>inferred PC / PC</td>
<td>59%</td>
</tr>
<tr>
<td>Strengthening factor</td>
<td>PC + relevant IC / PC</td>
<td>5.1</td>
</tr>
</tbody>
</table>

IC = Inferred contract Clauses
PC = Programmer-written contract Clauses
DEMO
AutoInfer

http://se.inf.ethz.ch/research/autoinfer

- Does not use Daikon
- Uses AutoTest to generate the test suite
- Infers universally quantified expressions and implications
- Uses functions with arguments as variables
- Only infers postconditions of commands
Example: \texttt{LIST.extend}

\texttt{extend} \ (v: G)

\begin{verbatim}
  -- Add `v' to end. Do not move cursor.
...
\end{verbatim}

\texttt{ensure}

\begin{align*}
\text{occurrences} \ (v) &= \text{occurrences} \ (v) + 1 \\
\text{count} &= \text{old count} + 1 \\
i_{th} \ (\text{old count} + 1) &= v \\
\forall i. \ 1 \leq i \leq \text{old count} \implies i_{th} \ (i) &= \text{old} \ i_{th} \ (i) \\
\text{old after} \implies \text{index} &= \text{old index} + 1 \\
\text{not old after} \implies \text{index} &= \text{old index} \\
\text{last} &= v \\
\forall o:G \neq v. \ \text{occurrences} \ (o) &= \text{old occurrences} \ (o) \\
\forall o:G \neq v. \ \text{has} \ (o) &= \text{old has} \ (o)
\end{align*}