Dynamic Contract Inference

Nadia Polikarpova

Software Verification

17.10.2012
Dynamic contract inference

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

  ...  
  \[ x := 0 \]
  ...  

  ![Diagram](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

Instrumenter

Execution

Detector

Source code

Instrumented code

Declarations

Trace

Test suite

Formatted invariants

Annotated code

Language-dependent

Postprocessor (printer, annotator, etc.)

Inferred invariants
Daikon architecture

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

der
end
end
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: \textit{balance}, \textit{amount}: INTEGER
- Invariants:
  - \underline{balance} = 0
  - \underline{balance} \geq 0
  - \underline{amount} = 10
  - \underline{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**
  \[ s[i] = i (\leq, \leq, >, \geq) \]
Binary invariant templates

- Comparisons
  $$x = y \ (<, \leq, >, \geq)$$

- Linear binary
  $$ax + by = 0$$

- Squared
  $$x = y^2$$

- Divides
  $$x = 0 \mod y$$

- Zero track
  $$x = 0 \implies y = 0$$

- Member
  $$x \in s$$

- Reversed
  $$s_1 = s_2.\text{reverse}$$

- Subsequence and subset
  $$s_1 \text{ is subsequence of } s_2 \quad \quad \quad s_1 \text{ is subset of } s_2$$
Ternary invariant templates

- Linear ternary
  \[ ax + by + zc = 0 \]
- Binary function
  \[ z = f(x, y) \]
  where \( f = \text{and, or, xor, min, max, gcd, pow} \)
Daikon architecture

Source code → Instrumenter → Instrumented code → Execution → Detector

- Declarations
- Trace

Instrumenter:
- (printer, annotator, etc.)

Postprocessor (annotated code, etc.):
- Formatted invariants
- Annotated code
- Language-dependent

Detector:
- Inferred invariants

Test suite
Annotator

- 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 and variables
  - 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
- relevant inferred invariants
- not inferred
- uninteresting
- incorrect
- 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
- Comparing several versions of a program
Improving quality

- Improving relevance
  - Statistical test
  - Redundant invariants
  - Comparability analysis

- Improving recall
  - More templates and variables
  - Conditional invariants
Statistical test

- Checking invariant
  \[ x \neq 0 \]

- 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

\begin{itemize}
\item \textbf{ensure}
\begin{align*}
& x > 0 \quad \cancel{x \neq 0} \\
& \ldots
\end{align*}
\end{itemize}

\begin{itemize}
\item Invariants that are implied by other invariants are not interesting
\item How to find them?
\begin{itemize}
\item General-purpose theorem prover
\item Daikon has built-in hierarchy of invariants (invariants know their suppressors)
\end{itemize}
\end{itemize}
Comparability analysis

class BANK_ACCOUNT

... 
invariant
    number > 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

  $x = y \quad x := y \quad x > y \quad x + y \quad ...$
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
  \((P_1 \text{ and } P_2 \ldots \text{ and } P_m) \implies Q\)
  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

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

http://se.inf.ethz.ch/people/polikarpova/citadel.html

- 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

- incorrect
- uninteresting
- new
- implied by written
- implied by inferred
- both
- not inferred
- not expressible

- inferred invariants
- perfect specification
- programmer-written invariants
## 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**
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: LIST.extend**

```
extend (v: G)
    -- Add `v' to end. Do not move cursor.

...

ensure

occurrences (v) = occurrences (v) + 1
count = old count + 1
i_th (old count + 1) = v
forall i . 1 <= i <= old count implies i_th (i) = old i_th (i)
old after implies index = old index + 1
not old after implies index = old index
last = v
forall o:G /= v . occurrences (o) = old occurrences (o)
forall o:G /= v . has (o) = old has (o)
```