Refactoring and Static Checking: Two Applications of Dynamic Invariant Detection

Michael Ernst
MIT Lab for Computer Science
http://sdg.lcs.mit.edu/~mernst/

Joint work with
Josh Kataoka, William Griswold, David Notkin, and
Jeremy Nimmer

Outline

Dynamic invariant detection: given a program, determine a specification

Refactoring: program restructuring

- Result: identify refactoring opportunities

Static checking: verify program properties

- Result: ease annotation and guarantee soundness

Conclusion

Dynamic invariant detection

Goal: recover invariants from programs

Technique: run the program, examine values

Artifact: Daikon

`http://sdg.lcs.mit.edu/daikon`

Experiments:

- reconstructed formal specifications
- aided in a software modification task

Goal: recover invariants

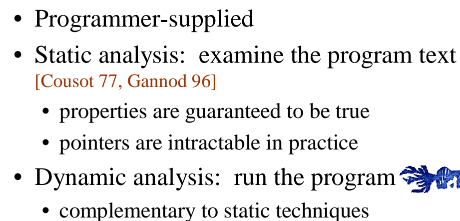
Detect invariants (as in **`assert`**s or specifications)

- `x > abs(y)`
- `x = 16*y + 4*z + 3`
- array **`a`** contains no duplicates
- for each node **`n`**, **`n = n.child.parent`**
- graph **`g`** is acyclic

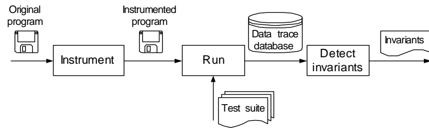
Uses for invariants

- Write better programs [Gries 81, Liskov 86]
- Document code
- Check assumptions: convert to **`assert`**
- Maintain invariants to avoid introducing bugs
- Locate unusual conditions
- Validate test suite: value coverage
- Provide hints for higher-level profile-directed compilation [Calder 98]
- Bootstrap proofs [Wegbreit 74, Bensalem 96]

Ways to obtain invariants

- Programmer-supplied
- Static analysis: examine the program text [Cousot 77, Gannod 96]
  - properties are guaranteed to be true
  - pointers are intractable in practice
- Dynamic analysis: run the program 
  - complementary to static techniques

Dynamic invariant detection



Look for patterns in values the program computes:

- Instrument the program to write data trace files
- Run the program on a test suite
- Invariant engine reads data traces, generates potential invariants, and checks them

Checking invariants

For each potential invariant:

- instantiate
  (determine constants like a and b in $y = ax + b$)
- check for each set of variable values
- stop checking when falsified

This is inexpensive: many invariants, each cheap

Improved invariant relevance

Add desired invariants:

1. Implicit values
2. Unused polymorphism

Eliminate undesired invariants
(and improve performance):

3. Unjustified properties
4. Redundant invariants
5. Incomparable variables

Techniques for pointers

Recursive data structures: linearize data structures

- Traverse pointer-directed data structures
- Present to invariant engine as sequence

Conditionals: data splitting

- Split the data into parts
- Compute invariants over each subset of data
- Compare results, produce implications

Experiment 1: Recover formal specifications

Found all formal specifications from *The Science of Programming* [Gries 81]

Also detected (erroneously) missing invariants

Likewise for other texts

Conclusion: the system is *accurate*

Experiment 2: C code lacking explicit invariants

563-line C program: regexp search & replace
[Hutchins 94, Rothermel 98]

Task: modify to add Kleene +

Use both detected invariants and traditional tools

Programmer use of invariants

Explicated data structures

- regexp compiled form (a string)

Contradicted some maintainer expectations

- anticipated $lj < j$ in `makepat`
- queried for counterexample
- avoided introducing a bug

Revealed a bug

- when $lastj = *j$ in `stclose`, array bounds error

More invariant uses

Showed procedures used in limited ways

- `makepat`: start = 0 and delim = '\0'

Demonstrated test suite inadequacy

- # calls to `in_set_2` = # calls to `stclose`

Changes in invariants validated program changes

- `stclose`: $*j = orig(*j)+1$
- `plclose`: $*j \geq orig(*j)+2$

Experiment 2 conclusions

Invariants:

- effectively summarize value data
- support programmer's own inferences
- lead programmers to think in terms of invariants
- provide serendipitous information

Conclusion: the system is *useful*

Outline

Dynamic invariant detection: given a program, determine a specification

➨ Refactoring: program restructuring

- Result: identify refactoring opportunities

Static checking: verify program properties

- Result: ease annotation and guarantee soundness

Conclusion

Refactoring

(Local) program restructuring

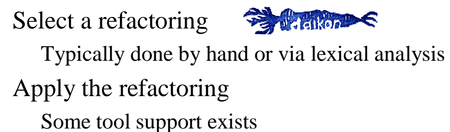
Enhance readability, performance, abstraction, maintainability, flexibility, ...

Beloved of Extreme Programming

Example: Extract Method

- find repeated code
- replace each instance by call to a new method

Refactoring steps

Select a refactoring 

- Typically done by hand or via lexical analysis

Apply the refactoring

- Some tool support exists

Identifying refactoring opportunities

Pattern of invariants ⇒ refactoring is applicable

Implemented invariant pattern matcher

Case study: 7000-line program

Its maintainer evaluated the results

Dynamically detected invariants may identify more refactoring opportunities

Refactorings examined

- Remove Parameter
- Eliminate Return Value
- Separate Query from Modifier
- Encapsulate Downcast
- Replace Temporary Variable by Query

Refactoring catalogs [Opdyke 92, Fowler 99] focus on simple lexical transformations

Remove Parameter

Applicable when parameter is constant or unused

- param = *constant*, or
- param = *f*(a, b, ...), where a, b, ... are in scope

Examples:

- height = width for all icons
- isAutomaticAspect = true in `Aspect` constructor
- `SetFirstItemFlag` called with constant argument

Eliminate Return Value

Applicable if return value is constant or unused

- return = *constant*, or
- return = *f*(a, b, ...), where a, b, ... are in scope

Example:

- return = true in `MakeObjectObey`

Separate Query from Modifier

Applicable when a method returns a value and has a side effect

- return ≠ *constant*, and
- a ≠ orig(a) for some a in scope

Example:

- mCurrentIndex = orig(mCurrentIndex) +1 in `CursorHistory.GetNextItem`

Encapsulate Downcast

Applicable when return value needs to be downcasted by the caller

- *LUB*(return.class) ≠ *declaredtype*(return)

Approximation:

- return.class = *constant*, and
- return.class ≠ *declaredtype*(return)

Example:

- comboBoxItems.class = AspectTraverseListItem[] in `AspectTraverseComboBox`

Replace Temp. Var. by Query

Applicable when a temporary variable holds the value of an expression
- temp = orig(temp), and
- a = orig(a) for all vars `a` in initializer of temp

Examples found after adding wrapper functions

Case study: Nebulous

A component of Aspect Browser [Griswold 01]

Visualizes cross-cutting aspects of a program

Manages changes to such aspects

Uses pattern matching and the map metaphor

78 files, 7000 non-comment non-blank lines

Case study methodology

Wrote a Perl script to identify invariant patterns in Daikon output

Ran Daikon over Nebulous executions

Ran script to identify refactoring opportunities

Nebulous programmer evaluated the recommendations

Programmer assessment

	yes	maybe	no	total
Remove Parameter	6	4	5	15
Eliminate Return Value	1	2	4	7
Sep. Query from Modifier	0	2	0	2
Encapsulate Downcast	1	1	0	2
Total	9	8	9	26

Remove Parameter: singletons, flags (another refactoring)

Eliminate Return Value: test suite, convenience

Separate Query from Modifier: style

Encapsulate Downcast: static count

Evaluation

Tool suggestions revealed architectural flaws, prompted redesign and code simplification

Easy to filter out poor suggestions
- No set of rules is right for all users and tasks
- Some are a matter of degree or of style

Maintainer had not previously identified these refactoring opportunities
- Suggestions orthogonal to clone detection tool

Future work

Add patterns for more refactorings

Perform more case studies

Combine with static analysis
- Static analysis better for "large method", "variable never used"
- Refactorings requiring static and dynamic info
- Compare dynamic and static counts

Combine with tool for applying refactorings

Outline

Dynamic invariant detection: given a program, determine a specification

Refactoring: program restructuring

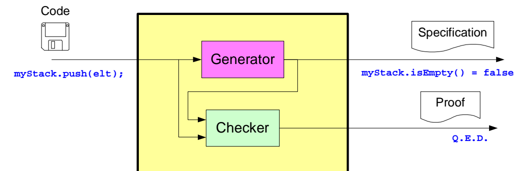
- Result: identify refactoring opportunities

➨ Static checking: verify program properties

- Result: ease annotation and guarantee soundness

Conclusion

Goal: Program specifications



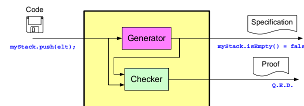
Previous approaches

Generation:

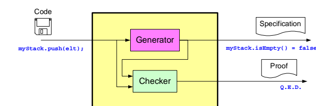
- By hand
- Static analysis

Checking

- By hand
- Non-executable models



Approach



Dynamic detection proposes *likely* invariants

- Not guaranteed to hold for all executions

Static checking verifies properties

- Difficult and tedious to annotate programs

Combining the techniques overcomes the weaknesses of each

- Ease annotation and guarantee soundness

Invariant checking

ESC/Java: Extended Static Checker for Java
[Detlefs 96, Leino 98]

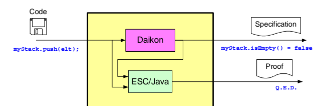
Lightweight technology: intermediate between type-checker and theorem-prover

Intended to detect array bounds and null dereference errors, and annotation violations

Modular: checks, and relies on, specifications

```
/*@  requires x >= 0                    */
/*@  ensures \result * \result == x     */
double sqrt(double x);
```


Integration approach



Run Daikon over target program

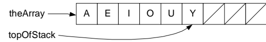
Insert results into program as annotations

Run ESC/Java on the annotated program

(All steps are automatic.)

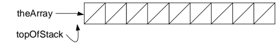
Stack object invariants

```
public class StackAr {
  Object[] theArray;
  int topOfStack;
}
```



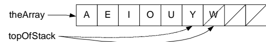

```
theArray != null
\typeof(theArray) == \type(java.lang.Object[])
topOfStack >= -1
topOfStack <= theArray.length - 1
theArray[0..topOfStack] != null
theArray[topOfStack+1..theArray.length-1] == null
```


StackAr constructor




```
public StackAr(int capacity) {
  /*@ requires capacity >= 0 */
  /*@ ensures capacity == theArray.length */
  /*@ ensures topOfStack == -1 */
  /*@ ensures
     theArray[0..theArray.length-1] == null */
  theArray = new Object[capacity];
  topOfStack = -1;
}
```


StackAr.push




```
public void push( Object x ) {
/*@ requires x != null */
/*@ requires topOfStack < theArray.length - 1 */
/*@ modifies topOfStack, theArray[*] */
/*@ ensures topOfStack == \old(topOfStack) + 1 */
/*@ ensures x == theArray[topOfStack] */
/*@ ensures theArray[0..\old(topOfStack)]
         == \old(theArray[0..topOfStack]) */
  ...
}
```


StackAr results

ESC/Java verified all Daikon invariants

No runtime errors if callers satisfy preconditions

Implementation meets generated specification

Reveal properties of the implementation (e.g., garbage collection of popped elements)

Detected a bug

Other experiments

Programs from textbooks, class assignments, and Java standard library

Contain interesting, nontrivial rep. invariants

Detected invariants did not always fully verify

Numeric results

Average values (10 programs)

- 280 lines of code
- Verified invariants: 38
- Unverified invariants: 2
- Inexpressible invariants: 9
- Redundant invariants: 11
- Missing invariants: 3
- Precision: .95 = verifiable / (verifiable + unverifiable)
- Recall: .94 = verifiable / (verifiable + missing)

Challenges to verification

ESC limitations
- Syntax, semantics, proving
- Not designed for proving program properties

Some properties are hard to detect and prove
- gcd, initializers that read from strings

Program bugs may prevent detection or proving
- `StackAr.makeEmpty` failed to clear array

Earlier Daikon enhancements helped

Test suites

Unverifiable properties can indicate:
- Usage properties of the program or environment
- Test suite deficiencies (missing special cases)

Constructing good test suites is easy
- The invariant detector helps

Solutions:
- Use system, not unit, tests
- Adjust statistical tests, make fewer approximations

Future work

Assess usefulness to programmers

Scale to larger programs

Improve generation and handling of test suites

Integrate with static analysis

Recover from failed verification attempts

Outline

Dynamic invariant detection: given a program, determine a specification

Refactoring: program restructuring
- Result: identify refactoring opportunities

Static checking: verify program properties
- Result: ease annotation and guarantee soundness

➨ Conclusion

Other current work

Determine relationship between test suite quality criteria and invariant detection

Detect temporal invariants (if car approaches intersection, light eventually turns green)

Integrate with IOA language and LP theorem-prover

Improve performance, scaling, conditional invariants, handling of C++, UI

Gather experience from users and experiments
- CMU, MIT (also 6.170), OSU, Raytheon, Toshiba, UCSD, UVa, UW

Conclusion

Dynamically discovered invariants can be used as input to tools

Refactoring opportunities aid code improvement

Static verification
- aids program annotation and checking
- makes dynamic invariant detection sound
- indicates current unsoundness is minor

Further evidence of the accuracy and usefulness of dynamic invariant detection