Automated Support for Program Refactoring Using Invariants

Yoshio Kataoka (Toshiba)

Michael Ernst (MIT)

William Griswold (UCSD)

David Notkin (UW)

Goal:
Automatically identify refactoring candidates

Refactoring

(Local) program restructuring

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

Beloved of Extreme Programming

Example: Extract Method

- find repeated code
- replace each occurrence 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

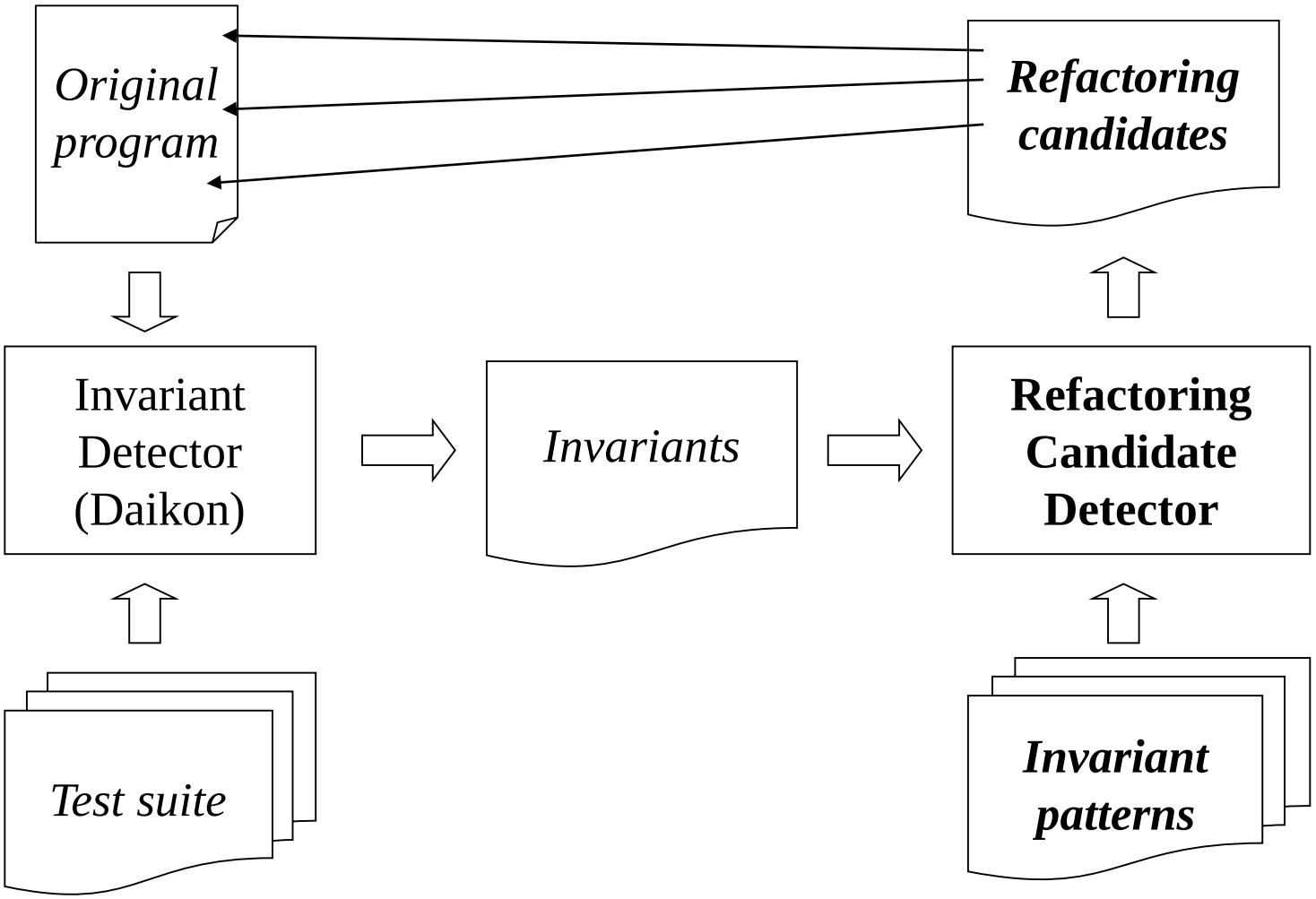
Pattern of invariants ⇒ refactoring is applicable

An invariant is a program property (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
- if **`ptr ≠ null`** then **`*ptr`** > **`i`**

Invariants are rarely present in practice

Tool architecture



Dynamic invariant detection

Goal: recover invariants from programs

Technique: run the program, examine values

- postulate potential invariants
- check for each set of variable values

Results are *likely* invariants

Dynamic invariant detection

Implementation: Daikon 

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

Experiments indicate *accuracy* and *usefulness*

- Recover/prove formal specs, aid programmers

Dynamically detected invariants may identify more refactoring opportunities

- Static analysis fails for pointers

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	8	9	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

Conclusions

Program invariants effectively identify refactoring candidates

Automatic technique

Justified in terms of run-time properties

Programmer assessment demonstrates utility and ease of use

Questions?