Daikon
An Efficient Tool for Dynamic Invariant Detection

Jeff Perkins and Michael Ernst

MIT CSAIL

Program Specifications

- Specifications define various components of a program
  - Class and object invariants
    - index $\geq$ 0;
    - index < array.length
  - Method preconditions and post-conditions
    - list is sorted
    - list $\neq$ null
- Specifications are useful
  - Error detection
  - Automatic theorem proving
  - Generating test cases
  - Program understanding
- Unfortunately, specifications are often not present

Dynamic invariant detection

- Look for invariants as the program is run (dynamically)
  - Invariants are properties over the program variables that are always true.
  - Invariants form a specification
- Simple incremental algorithm
  - Track the value of each variable of interest at method entry and exit
  - Hypothesize each invariant in the grammar
    - Over each set of variables
    - At each program point
  - Check observed values for each variable (sample) at each invariant
    - Discard invariants that are falsified
  - The remaining invariants are true over the sample data
- Simple incremental algorithm scales poorly
  - The number of invariants to be checked is on the order of $(\#variables)^9$ - billions in modest test cases

Daikon - optimized dynamic invariant detector

- Produce useful and expressive program properties
  - Rich set of derived variables
    - array references: `a[i]`, `a[i..]`, `a[..i]`
    - pre-state variables: at exit, `orig(x)` stands for the value at entry
  - Rich invariant grammar
    - unary, binary, and ternary invariants
    - invariants over pointers, integers, floats, strings and arrays
- Runs on-line (in a single pass)
  - Supports pipes and sockets
  - Can handle arbitrarily large amounts of data
- Optimizations reduce the number of invariants to be checked by more than 99%

Outline

- Optimizations
  - Optimization Approach
  - Constants
  - Equality sets
  - Program point and variable hierarchy
  - Suppression
  - Optimizations are effective
  - Real programs can be processed
- Daikon Tool
- Conclusion

Optimization Approach

- Many invariants are redundant (they are implied by other invariants)
  - $(x = 5) \wedge (y = 6) \Rightarrow (x < y)$
  - $(x < y) \Rightarrow (x \leq y)$
  - $(x \geq y)$ at class `Stack` $\Rightarrow$ $(x \geq y)$ at method `Stack.top()`
- Redundant invariants are not instantiated or checked
  - Many invariants are implied by others
  - As long as the antecedents are true, the consequent need be neither instantiated nor checked
- An invariant must be created when its antecedent is falsified
  - $(x = y) \wedge \text{odd}(x) \Rightarrow \text{odd}(y)$
  - If a sample is seen where $x \neq y$, the odd(y) invariant must be created
  - The new invariant must be true over all *past* samples
  - The new invariant must be checked over future samples

Constants

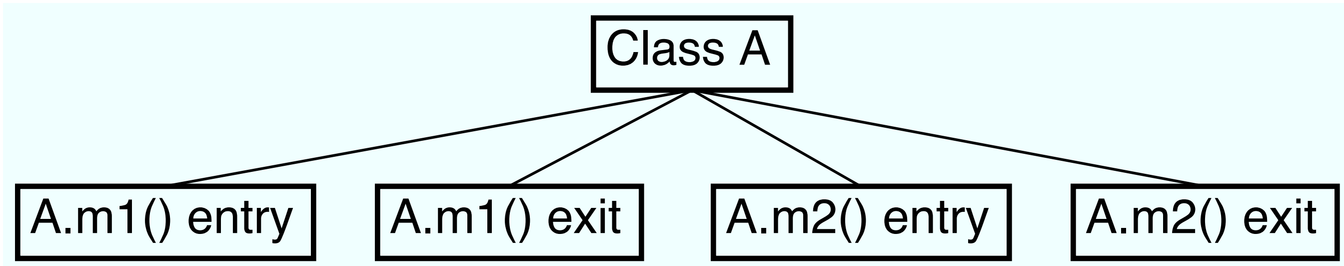
- Invariants over (only) constant variables are redundant
  - $(x = 5) \Rightarrow \text{odd}(x)$
  - $(x = 5) \wedge (y = 6) \Rightarrow x < y$
- All variables are initially constant
- Invariants are not instantiated *between* constants
- When (*var* = constant) is falsified
  - Invariants are instantiated between it and all remaining constants
  - Invariants which are not true over the constant values are discarded

Equality sets

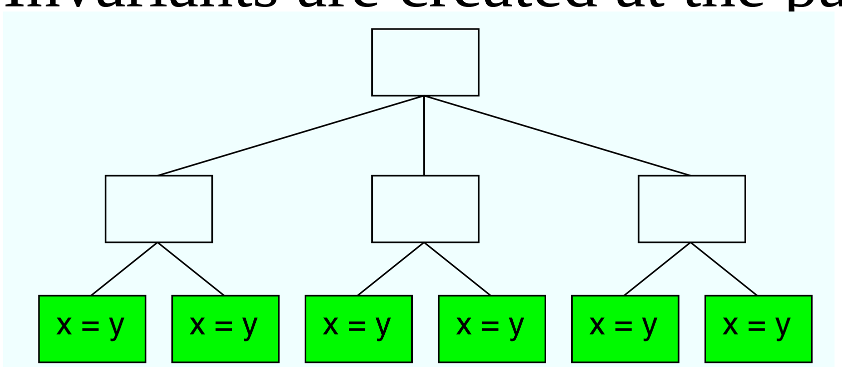
- If two or more variables are equal, any invariant true over one variable is true over all of them
  - $(x = y)$ and $f(x) \Rightarrow f(y)$
- Initially, all variables are placed in a single equality set
- One variable (the leader) represents the set
- Invariants are instantiated *only* between leaders
- When (*var1* = *var2*) is falsified
  - The set is split into two or more equality sets
  - Invariants over each old leader are copied to each new leader

Program point and variable hierarchy

- Relationship between program points



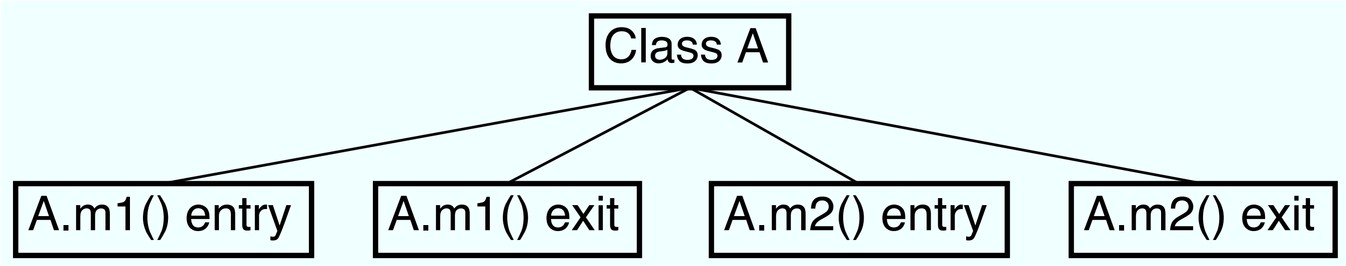
- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child



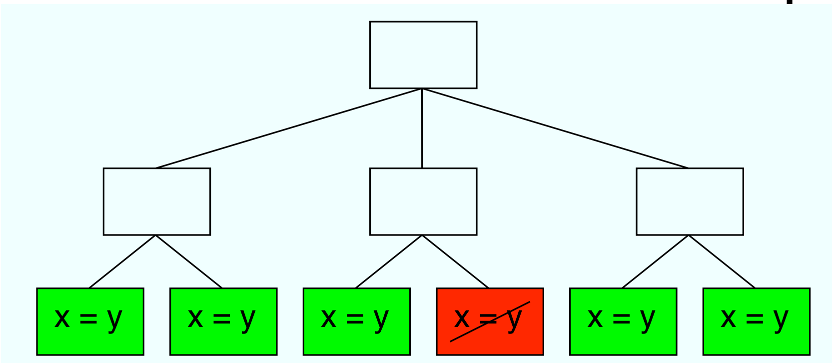
Initially each invariant (e.g., x = y) holds at each leaf

Program point and variable hierarchy

- Relationship between program points



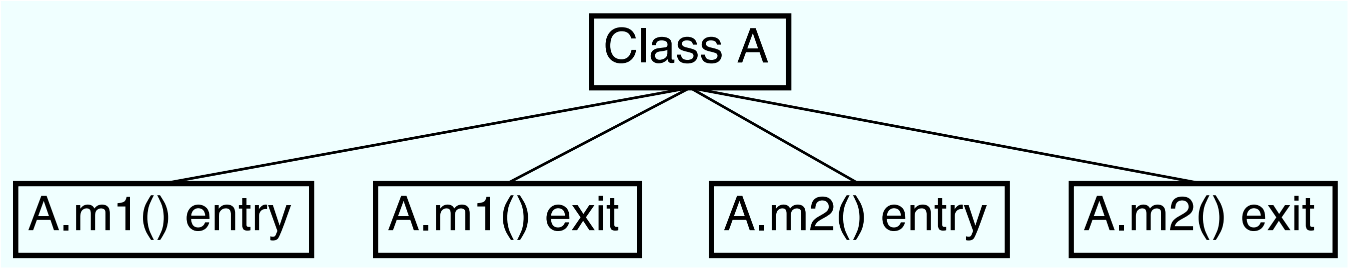
- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child



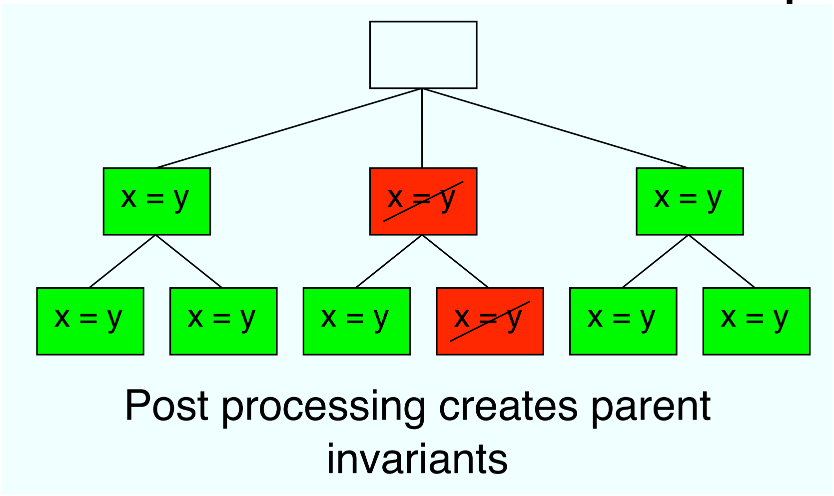
After processing the invariant was falsified at one program point (red)

Program point and variable hierarchy

- Relationship between program points

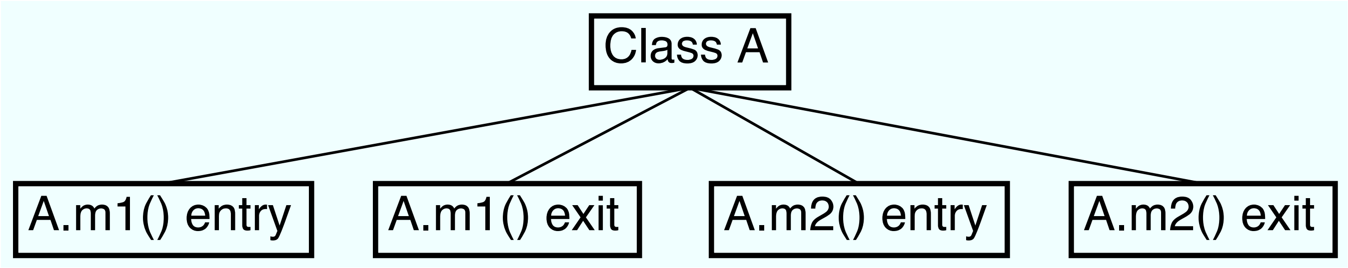


- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child

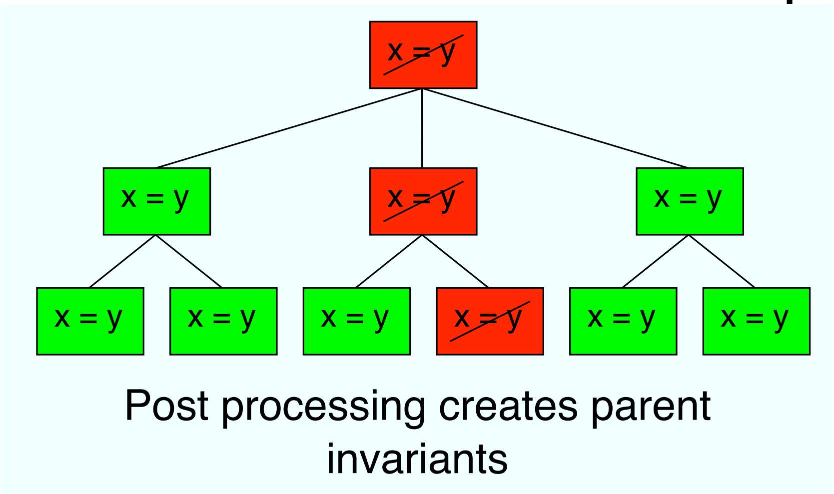


Program point and variable hierarchy

- Relationship between program points



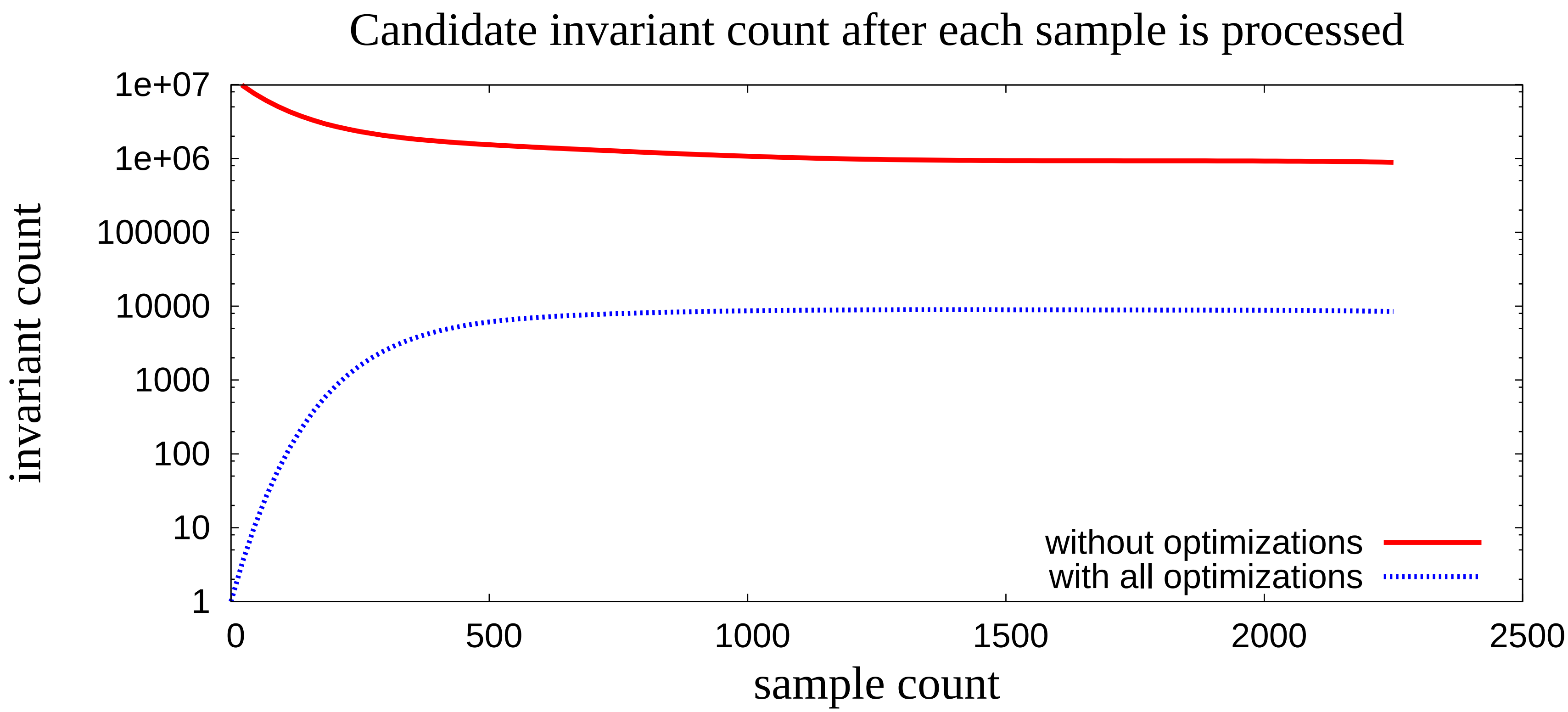
- Samples are only processed at the leaves of the hierarchy
- Invariants are created at the parent *iff* it is true at each child



Suppression

- An invariant can be suppressed if it is logically implied by some set of other invariants. For example:
  - $(x = y) \wedge (z = 1) \;\Rightarrow\; x = y \cdot z$
  - $(x = 0) \wedge (y = 0) \;\Rightarrow\; x = y \,\&\, z$
- Other optimizations are special cases of suppression
- Goals
  - Instantiate/check only non-redundant invariants
  - Use *no* storage for a non-instantiated invariants
- When an antecedent is falsified
  - Each invariant that might be suppressed is checked
  - If a suppression held before the antecedent was falsified, but no suppression holds after, the invariant is instantiated

Optimizations are effective



- 100 times fewer invariants with the optimizations

Real programs can be processed

- Flex lexical analyzer generator
  - 391 program points averaging 275 variables each
  - 232,000 samples (9.2 Gbytes of data)
  - Processing time of 4 hours
  - Max memory use of 750 Mbytes
- Daikon utilities
  - 1593 program points averaging 60 variables each
  - 26 million samples (11.5 Gbytes of data)
  - Processing time of 1.5 hours
  - Max memory use of 150 Mbytes
- Comp package of javac
  - 12,506 lines of code

Outline

- Optimizations
- Daikon Tool
  - Tool Overview
  - Daikon supports different languages and platforms
  - Daikon output formats
  - Daikon Extensibility
- Conclusion

Tool Overview

- Maintained by the Program Analysis Group, monthly releases
- Freely available
- Extensive documentation
  - 156 page user manual
  - 129 page developer manual
- Daikon has been extensively as a tool in other research (52 published papers) and as a test subject (9 published papers)
- Active mailing list
- Ongoing enhancements

Daikon supports different languages and platforms

- The inference engine is separated from the front end
- The front end is responsible for value profiling -- collecting the value of each variable at each method entry/exit
- Front ends are available for
  - Java on all platforms
  - C/C++ on Linux using the Valgrind instrumentation package
  - C/C++ on Windows using source translation and Purify
  - Perl
  - Spreadsheet data

Daikon output formats

- DBC - Design by contract format for Parasoft's Jtest tool
- ESC - Extended static checker format for use by ESC/Java
- JML - Java Modeling Language format used by many tools
- Simplify - Format of the Simplify automated theorem prover
- Java - Java expressions that can be used as assertions
- Daikon - default format with quantifiers and other features
- Annotate tool allows DBC, ESC, Java, or JML format invariants to be inserted directly into source code as annotations

Daikon Extensibility

- Instrumentors can be added for new languages or other data
- New invariants can be added by extending existing classes
- New derived variables can be added by extending existing classes
- Output formats can be added
- Additional suppressions can be added

Outline

- Optimizations
- Daikon Tool
- Conclusion
    - Current and future work

Current and future work

- Improving performance on larger programs
- Abstract types
  - dynamic inference of which variables contain related information
  - Only invariants over related variables are interesting
  - Reduces Daikon run-time and improves its results
- Instrumentation of windows binaries without source information
- Improved ability to add resulting specifications to programs