Searching for Program Invariants using Genetic Programming and Mutation Testing

Sam Ratcliff, David R. White and John A. Clark.

The University of York

SEBASE

The 13th CREST Open Workshop

Thursday 12 May 2011

Outline

Outline

What is an invariant?

Algorithm 1 Array sum program

$i, s := 0, 0;$
do $i \neq n \rightarrow$
  $i, s := i + 1, s + b[i]$
od

Precondition: $n \geq 0$
Postcondition: $s = \sum_{j=0}^{n-1} b[j]$
Loop invariant: $0 \leq i \leq n$ and $s = \sum_{j=0}^{i-1} b[j]$

What use is an invariant? What are they good for?

They can be provided as a specification for a programmer.

They can be used to derive programs or to prove them correct.

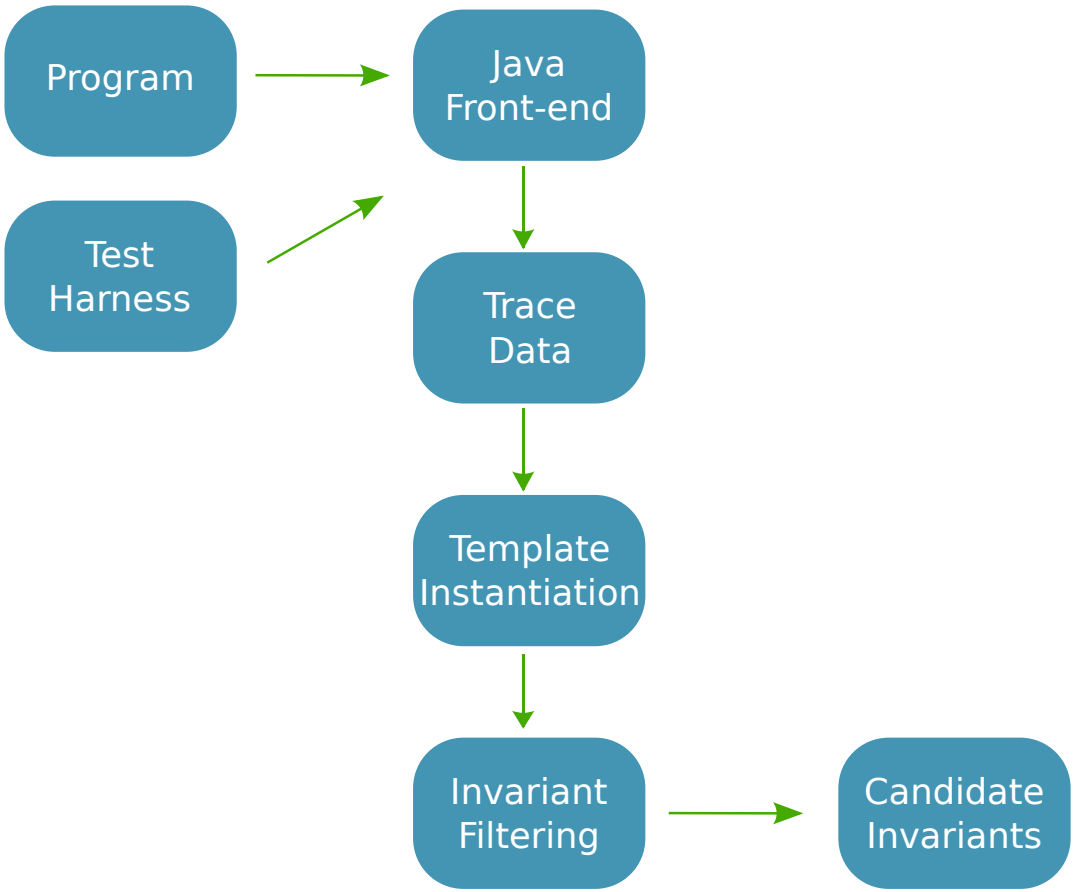
A well-known example of their use is in the design-by-contract paradigm.

How do we create invariants?

They are provided by the programmer (sometimes).

We can also try to generate invariants for a given program...

The Daikon Invariant Generator



The Limitations of Daikon

Daikon is limited in two regards:

- templates are restricted in size to make instantiation tractable.
- invariants are limited to those embodied in the repository of templates.

We would like to be able to locate invariants of arbitrary size and complexity. . .

Outline

Using GP to find Invariants

Two different approaches:

Daikon

Brute force-enumeration and data-driven restriction.

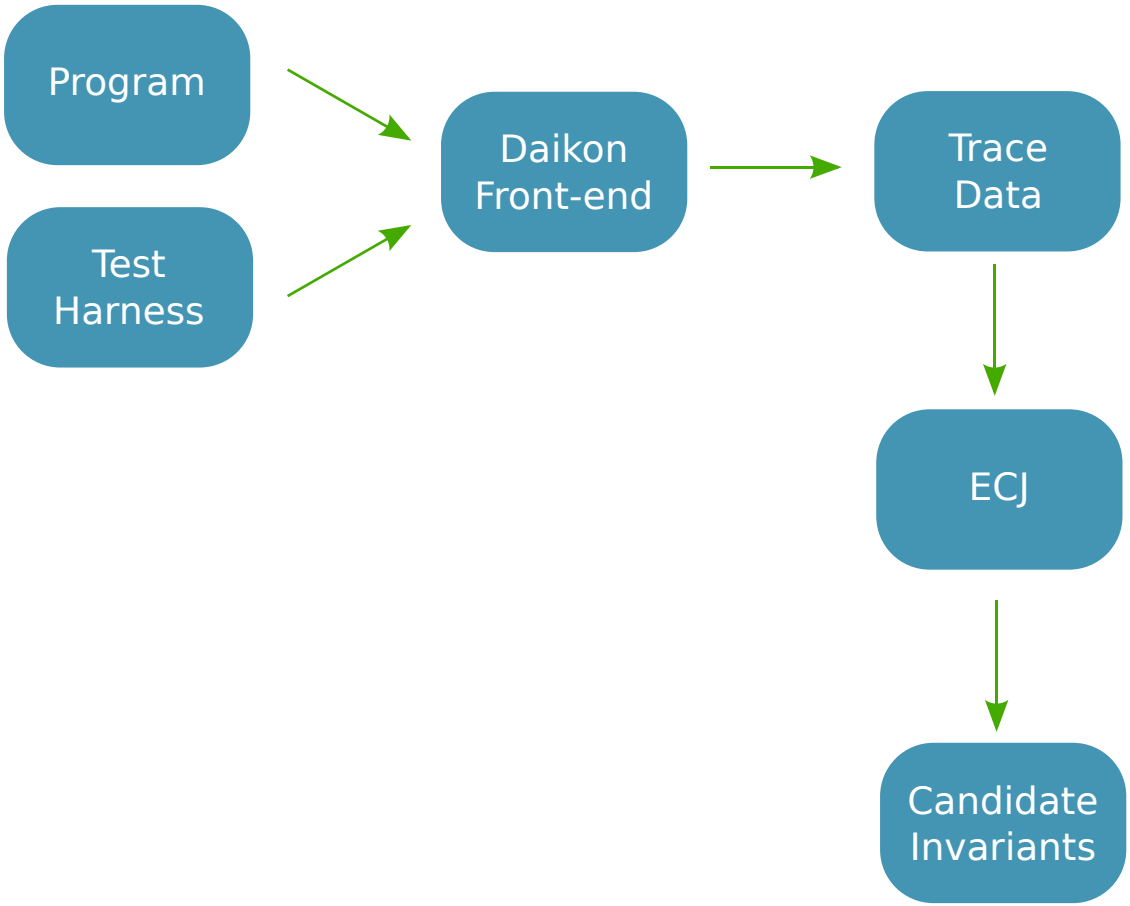
Search Approach

Construction of invariants guided by heuristics.

Method

1. Loop invariants are considered by adding dummy methods (Daikon deals in method entry/exit points).
2. Daikon's front-end tools are used to generate execution traces.
3. GP is used to search the space of invariants. (Fitness is the number of samples the invariant is consistent with).
4. Fully-consistent invariants are added to an archive.
5. Syntactic equivalents subsequently punished.
6. Archive is output at the end of the search.

Method Overview



GP Search

A GP search is run for each method.

Used population sizes of $\{100,250\}$, and similar total number of generations.

Mutation ($p=0.9$) heavily favoured over crossover ($p=0.1$).

GP Search

A GP search is run for each method.

Used population sizes of $\{100,250\}$, and similar total number of generations.

Mutation (p=0.9) heavily favoured over crossover (p=0.1).

$\rightarrow$ *Emphasis on individual improvements, exploiting syntactic similarity of consistent invariants.*

Functions over Variables

Function	**Description**
$=$	Equals
$= 0$	Equals zero
$>$	Greater than
$\geq$	Greater than or equal to
≥ 0	Greater than or equal to 0
≥ 1	Greater than or equal to 1
%	Modulo
$\neq$	Is not equal to
$\neq 0$	Is not equal to zero

Function over Arrays

Function	**Description**
ArrayElement	Value at array position
ArrayLessThan	Lexical comparison of two arrays
ArrayLEQ	Lexical comparison of two arrays
ArraysEqual	Numeric comparison of two arrays
IsMemberOf	Membership of an array
LEQAllElements	Compare value to all values in array
MaxIndex	The last index of an array
NotNull	Check if array is not null
PreviousElement	Element at position prior to argument
Size	Array size
SortedArray	Is array sorted?

Functions used by Gries

Function	**Description**
AND	Logical AND
ArraySum	Array sum
GCD	Greatest common divisor of variables
IsMemberSubArray	Does the subarray contain this value?
LEQSubArray	Compare value to subarray
≤ 0	Less than or equal to zero
Negative	Multiply by -1
OR	Logical OR
PermOfFour	Permutation of four values
PermOfTwo	Permutation of two values

Example Programs

One set taken from Gries' work, used previously in Daikon:

Example	**Inputs**	**Description**
Abs	int x	x set to abs(x)
ArraySum	int[ ] b, int s	s set to sum(b)
FourTupleSort	int q0, q1, q2, q3	inputs ordered
GCD	int x, y	x set to gcd(x,y)
Max	int x, y, z	z set to max(x,y)
MinArray	int[ ] b, int x	x set to min. value in b
Perm	int x, y	x and y ordered by $<$

. . . and also Bubble Sort, Insertion Sort, Quicksort and Selection Sort.

Results (1/2)

Example	Program Point	Median Percentage Found
Abs	abs	100.00
ArraySum	arraysum	100.00
ArraySum	loop	92.86
BubbleSort	bubblesort	100.00
BubbleSort	inner loop	100.00
BubbleSort	outer loop	100.00
FourTupleSort	fourtuplesort	100.00
GCD	gcd	100.00
GCD	loop	66.67
InsertionSort	insertionsort	100.00
InsertionSort	inner loop	53.45
InsertionSort	outer loop	86.84

Results (2/2)

Example	Program Point	Median Percentage Found
Max	max	100.00
MinArray	minarray	100.00
MinArray	loop	100.00
Perm	perm	100.00
Quicksort	dummy	100.00
Quicksort	partition	63.56
Quicksort	quicksort	100.00
Quicksort	quicksortrecursive	62.50
SelectionSort	selectionsort	100.00
SelectionSort	inner loop	72.50
SelectionSort	outer loop	100.00

Outline

Uninteresting Invariants

Success rates look impressive: we have found most of the Daikon invariants.

There's something I didn't mention - for one experiment, we found ...

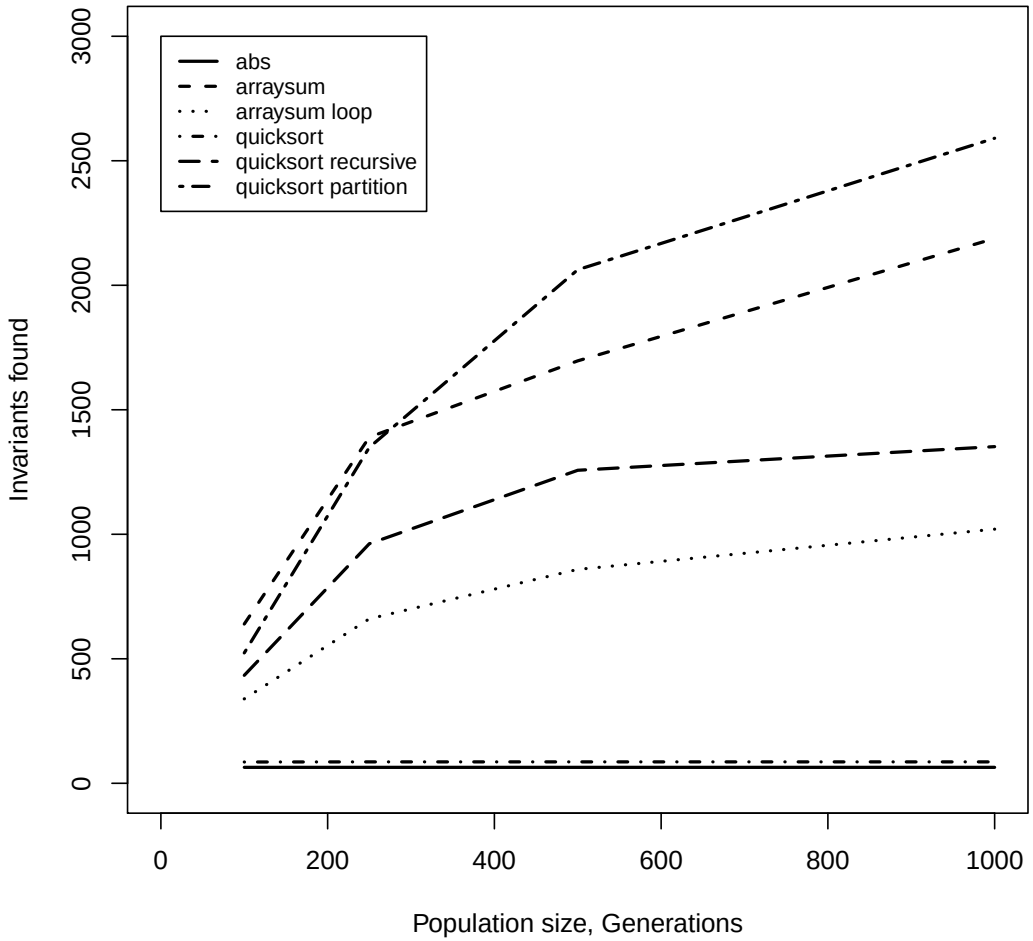
Uninteresting Invariants

Success rates look impressive: we have found most of the Daikon invariants.

There's something I didn't mention - for one experiment, we found ...

45 997 invariants!

That's a lot of invariants



What are these invariants?

Some of them are:

- Tautologies.
- Syntactic equivalents of the Daikon invariants.

What are these invariants?

Some of them are:

- Tautologies.
- Syntactic equivalents of the Daikon invariants.

. . . but some of them are just plain obvious, irrelevant or uninteresting. How can we get rid of them?

Using Mutation Testing

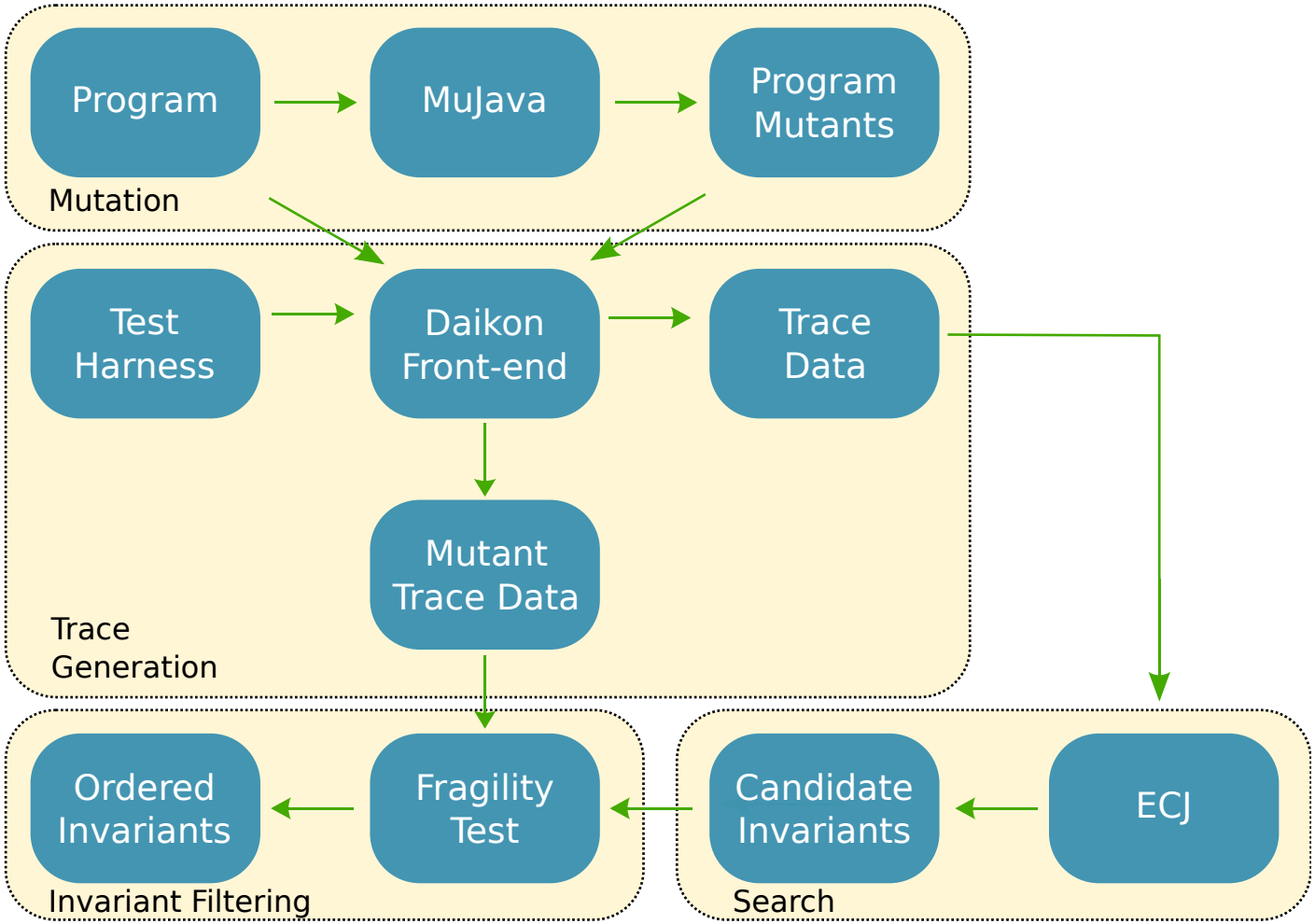
Mutation Testing

Good test data can identify small syntactic errors.

Mutation Fragility Test

Useful invariants are those general enough to be consistent with the traces of the program yet specific enough to be inconsistent with the trace data of (first-order) mutants.

Mutation Fragility Test



Mutation Fragility Test

Each invariant i is therefore assigned a priority score, $p(i)$:

$$p(i) = \frac{1}{|M|} \sum_{m \in M} \frac{1}{|S|} \sum_{s \in S} c(i, s) \tag{1}$$

M is the set of relevant mutants, S the set of sample data points for a mutant, $c(i, s)$ is 1 if the invariant is consistent with the mutant datapoint.

We can then order the invariant list by this value.

Results: the Brief Highlights

Algorithm 1 Array sum program

$i, s := 0, 0;$
do $i \neq n \rightarrow$
  $i, s := i + 1, s + b[i]$
od

Precondition: $n \geq 0$
Postcondition: $s = \sum_{j=0}^{n-1} b[j]$
Loop invariant: $0 \leq i \leq n$ and $s = \sum_{j=0}^{i-1} b[j]$

Results: the Brief Highlights

For the ArraySum method and loop program points, the system generated 1837 and 789 invariants respectively. Hence a programmer must examine 2626 invariants.

Results: the Brief Highlights

For the ArraySum method and loop program points, the system generated 1837 and 789 invariants respectively. Hence a programmer must examine 2626 invariants.

When the mutant fragility metric is used to order them, only 17 must be examined.

Results: the Brief Highlights

Similarly, the GCD example from Gries includes an invariant that is ranked 1 out of 3114 invariants!

Results: Summary

Method	Invariant	Depth	Total
arraysum	$orig(n) \geq 0$	1837	1837
arraysum	$s = \sum_{k=0}^{n-1} b[k]$	**4**	1837
arraysum.loop	$i \geq 0$	340	789
arraysum.loop	$n \geq i$	289	789
arraysum.loop	$s = \sum_{k=0}^{i-1} b[k]$	**13**	789
fourtuplesort	$q0 \leq q1$	123	557
fourtuplesort	$q1 \leq q2$	136	557
fourtuplesort	$q2 \leq q3$	142	557
gcd	$orig(x) \geq 1$	1685	1685
gcd	$orig(y) \geq 1$	1685	1685
gcd	$x = gcd(orig(x), orig(y)$	**15**	1685
gcd	$x \geq 1$	1228	1685
gcd	$x = y$	135	1685
gcd	$gcd(x, y) = gcd(orig(x), orig(y))$	243	1685
gcd.loop	$x \geq 1$	358	3114
gcd.loop	$gcd(x, y) = gcd(orig(x), orig(y))$	691	3114
gcd.loop	$x = gcd(orig(x), orig(y)$	**1**	3114

Results: Summary II

Method	Invariant	Depth	Total
max	$z \geq x$	11055	11055
max	$z \geq y$	555	11055
max	$(z = x) \vee z = y)$	1172	11055
minarray	$n \geq i$	1506	1506
minarray	$\forall k, 0 \geq k \leq i - 1, x \leq b[k]$	**22**	1506
minarray	$x \in b$	**24**	1506
minarray.loop	$i \geq 1$	571	2173
minarray.loop	$n \geq i$	571	2173
minarray.loop	$\forall k, 0 \geq k \leq i - 1, x \leq b[k]$	**38**	2173
minarray.loop	$x \in b$	**18**	2173
perm	$x \leq y$	**40**	243
perm	$\{x, y\}$ is perm. of $\{y, x\}$	**8**	243

Outline

Summary

Contributions:

- A new way of finding invariants.
- A new way of prioritising invariants.

Future Work:

- Improving search guidance.
- Applying mutation testing to Daikon output.

More at GECCO 2011...

For further details, see our paper (to appear at GECCO 2011):

"Searching for Invariants using Genetic Programming and Mutation Testing."

Sam Ratcliffe, David R. White and John A. Clark.

A paper is in preparation on using mutation testing to prioritise invariants produced by Daikon.