Detecting Invariants

Andreas Zeller + Andreas Gross

Lehrstuhl Softwaretechnik
Universität des Saarlandes, Saarbrücken

Exam

on Tuesday, 2003-02-18, 11:15 in lecture room 45/001 (here)

Written examination, duration: 90 minutes

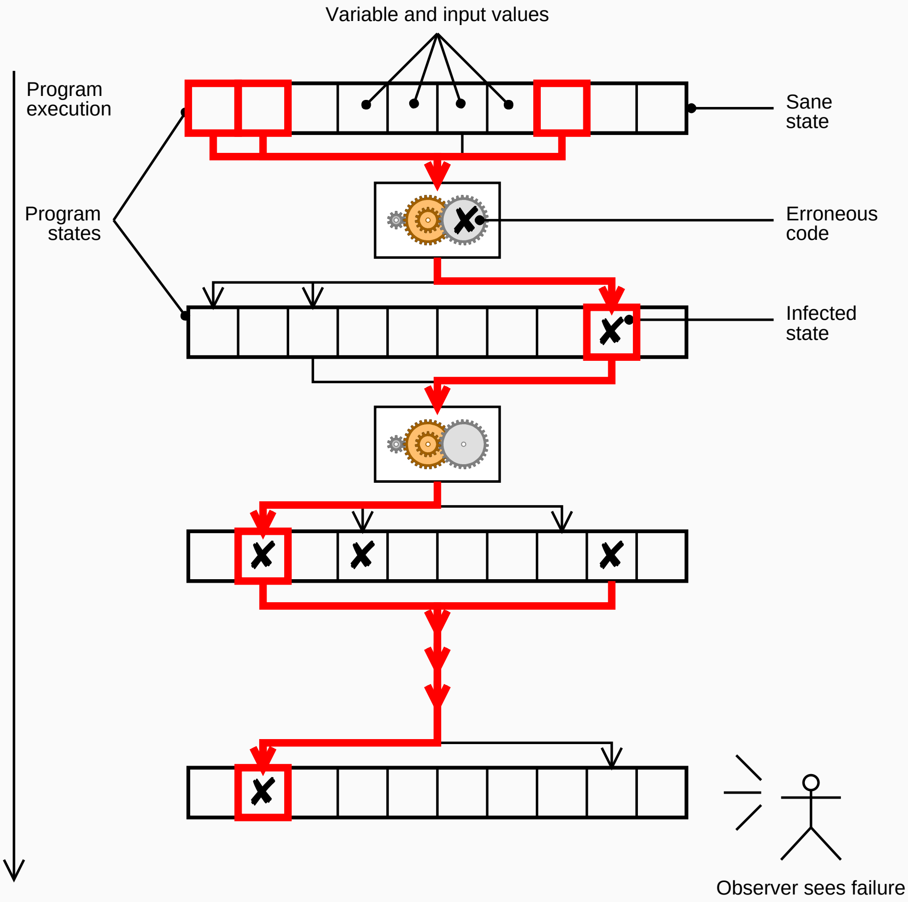
Tools: course material, books, papers; no electronic devices

Final grade will be 20% exercises, 80% examination

Q & A lab on Friday, 2003-02-14

Register by e-mail to Holger Cleve ⟨ *cleve@cs.uni-sb.de* ⟩ until Friday, 2003-02-15

Cause-Effect Chains



Reasoning Techniques

Deduction is reasoning from the general to the particular

e.g. from the program code (abstract) to the program run (concrete)

Example: static program analysis

Induction is reasoning from the particular to the general

e.g. from multiple program runs to common properties

Example: anomaly detection by coverage

Invariants

An *invariant* is a property that holds for all correct program runs.

```
int result = a / b;     // b != 0
int day_of_month;       // 1 ≤ day_of_month ≤ 12
```

Invariants...

- can be checked at run-time (*assertions*)
- can be verified statically
- are typically required for a correct execution
- are seldom explicitly specified

Invariants (2)

Possible uses of invariants:

Refactoring. Eliminate unused variables (e.g. invariants `temp == a`)

Modification. Make sure modifications do not affect the invariants.

Debugging. Report invariant violations; detect abnormal invariants.

Sources of Invariants

Programmer. Rely on specified *assertions and comments*.

- ✔ Invariants are directly accessible
- ✘ Invariants are seldom specified

Static Analysis. Deduce invariants from *source*.

- ✔ Invariants are correct.
- ✘ All limits of static analysis: obscure code, pointers, ...

Dynamic Detection. Induce invariants from *program runs*.

- ✔ automatic, efficient, only based on observation
- ✘ invariants hold only for observed runs.

Invariant Detection Tools

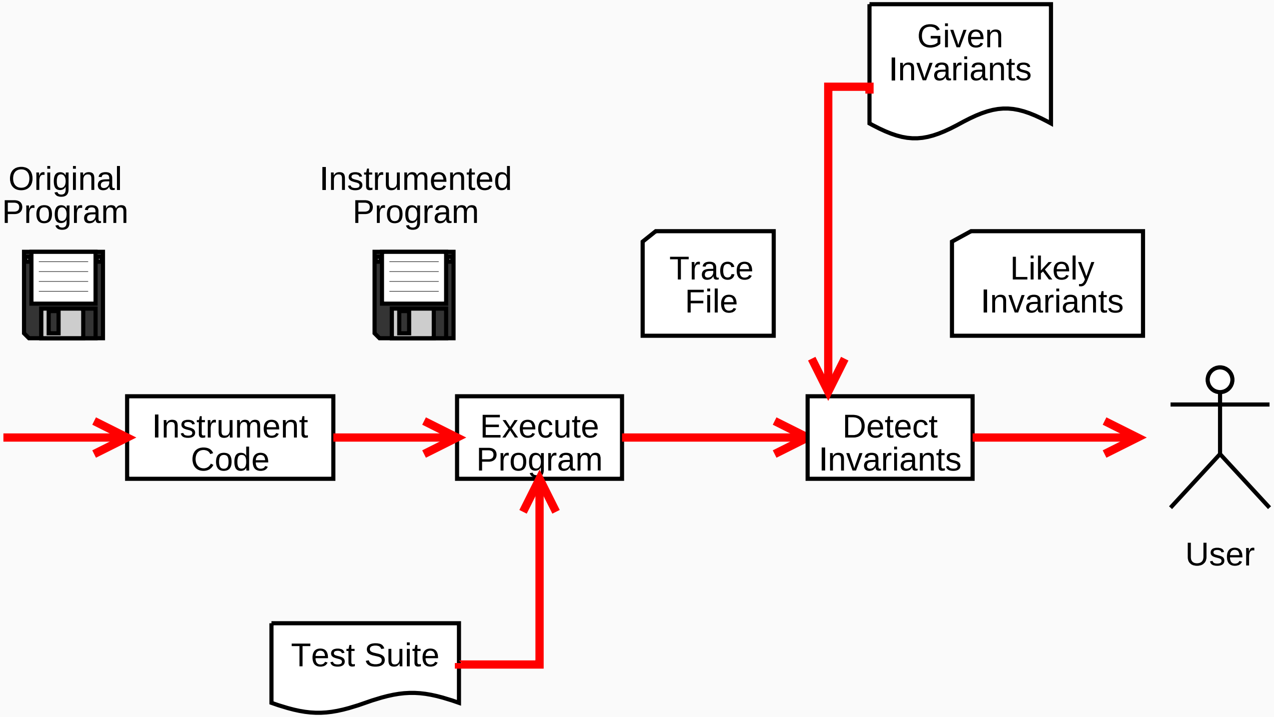
Daikon helps in refactoring, modification and debugging

- Determines *invariants*
- Written by Michael Ernst et al. (1998)
- C++, Java, Lisp and other languages
- analyzed up to $\sim$ 13.000 lines of code

Diduce (Dynamic Invariant Detection $\cup$ Checking Engine)

- Determines *invariant violations*
- Written by Sudheendra Hangal and Monica S. Lam (2001)
- Java bytecode
- analyzed $>$ 30.000 lines of code

How Daikon Works



Step 1: Instrument Code

Daikon instruments the code to *trace* and *check* variables.

Example—`sample.c` becomes `sample.cc` (C++ code):

```
static void shell_sort(DaikonSmartPointer<int> a, int size)
{
    DaikonAddressValidator<sizeof(int)> daikon_validate_address_1(&size);
    daikon_output_to_dtrace("std.shell_sort(int *;int;)void::ENTER\n");
    daikon_output_pointer("a", a);
    daikon_output_smartpointer_ints("a[]", a);
    daikon_output_int("size", int(size));
    int i = 0;
    DaikonAddressValidator<sizeof(int)> daikon_validate_address_2(&i);
    int j = 0;
    DaikonAddressValidator<sizeof(int)> daikon_validate_address_3(&j);

    do {
      ...
    } while (h != 1);
    daikon_output_to_dtrace("std.shell_sort(int *;int;)void::EXIT1\n");
    ...
}
```

Step 2: Execute Program

We compile the instrumented program and execute it using a given large test suite:

```
$ ./sample-daikon 1 2 3
fatal: at sample.c line 17: attempted to access
       index 3 of a 3-length array
       (max legal index is 2)
$ _
```

Okay—we fix it first :-)

```
$ export DTRACEAPPEND=1
$ ./sample-daikon 1 2 3
$ ./sample-daikon 4 5 6
```

Step 3: Run Daikon

Daikon generates invariants for the sample program:

```
$ java -classpath daikon.jar \
    daikon.Daikon sample.decls sample.dtrace -o sample.inv
Daikon version 2.3.13, released July 17, 2002;
http://pag.lcs.mit.edu/daikon.
Reading declaration files .
Reading data trace files .
Read 1 declaration file, 0 spinfo files, 1 dtrace file
```

Invariants in main


```
std.main(int;char **;):::ENTER
argc == size(argv[])-1
argc == 4
size(argv[]) == 5
argv[argc..] == [null]
argv[argc..] elements == null
argv[argc+1..] == []

std.main(int;char **;):::EXIT2
argv[] == orig(argv[])
return == 0
argv[orig(argc)..] == [null]
argv[orig(argc)..] elements == null
argv[orig(argc)+1..] == []
```

Invariants in shell_sort


```
std.shell_sort(int *;int;):::ENTER
size == size(a[])
a[] one of  [1, 2, 3], [4, 5, 6]
size == 3

std.shell_sort(int *;int;):::EXIT1
a[] == orig(a[])
```

Daikon's Invariant Algorithm

The set `invariants` holds the currently valid invariants

for each *execution step*:
    for each *variable* at *execution step*:
        if ¬exist invariants[*variable*]:
            invariants[*variable*] = ⟨daikon invariants⟩
        for each *invariant* in invariants[*variable*]:
            if value(*variable*) violates *invariant*:
                invariants[*variable*] -= *invariant*

Possible Invariants

Variable Invariants compare at most three variables; like

```
x = 6;        x ∈ 2, 5, -30
x < y;        y = 5 * x + 10;
z = 4 * x + 12 * y + 3;
z = fn(x, y)
```

Sequence Invariants like `A subsequence B; A < B`

Object Invariants like

```
string.content[string.length] = '\0';
node.left.value ≤ node.right.value
this.next.last = this
```

Daikon in Action

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

Precondition: $n \geq 0$
Postcondition: $s = (\sum j : 0 \leq j < n : b[j])$
Loop invariant: $0 \leq i \leq n$ and $s = (\sum j : 0 \leq j < i :$

(from *The Science of Programming*)

```
15.1.1:::ENTER                          100 samples
  N = size(B)                             (7 values)
  N in [7..13]                            (7 values)
  B                                       (100 values)
    All elements in [-100..100]           (200 values)

15.1.1:::EXIT                           100 samples
  N = I = orig(N) = size(B)               (7 values)
  B = orig(B)                             (100 values)
  S = sum(B)                              (96 values)
  N in [7..13]                            (7 values)
  B                                       (100 values)
    All elements in [-100..100]           (200 values)

15.1.1:::LOOP                           1107 samples
  N = size(B)                             (7 values)
  S = sum(B[0..I-1])                      (452 values)
  N in [7..13]                            (7 values)
  I in [0..13]                            (14 values)
  I <= N                                  (77 values)
  B                                       (100 values)
    All elements in [-100..100]           (200 values)
  B[0..I-1]                               (985 values)
    All elements in [-100..100]           (200 values)
```

Enhancing Relevance

How can we make the invariants as relevant as possible?

- Dealing with Polymorphism
- Derived Values
- Eliminate Redundant Invariants
- Trustable Invariants
- Verifying Correctness

Dealing with Polymorphism

Problem: Comparing polymorphic variables (e.g. superclasses)

Solution: Let x be a polymorphic variable, e.g. `object x`

1. Find invariant for type of x,
   e.g. `x != null => x.type == int`
2. If invariant holds, replace `object x` by `int x`
3. Search invariants

Effect: More invariants are found.

Derived Values

Problem: Some relevant values are not found in variables:

- the size of an array, `size(a)`
- borderline values, `a[0]`, `a[size(a) - 1]`

Solution: Insert new variables when instrumenting code

- `int size_a = size(a);`
- `int extremals_a = {a[0], a[size(a) - 1]}`

Effect: More invariants are found.

Derived Values (2)

Derived values created by Daikon include:

for a sequence S: `size(S), S[0], S[1],`
`S[size(S) - 1], S[size(S) - 2]`

for a numeric sequence S: `sum(S), min(S), max(S)`

for a sequence S and an integer i:
`S[i], S[i - 1], S[0..i], S[0..i - 1]`

for methods: number of method calls

Eliminate Redundant Invariants

Problem: Let A, B be invariants. If $A \Rightarrow B$ holds, we don't have to know about B:

- A: $4 \leq x \leq 15$
- B: $x \neq 0$

Solution 1: Check for redundancies before output

Solution 2: Do not create redundant derived values like

- `first_element(a[0..12]) = first_element(a[0..5])`

Effect: Less invariants.

Trustable Invariants

Problem: Found invariant $-15 \leq x \leq 15, x \neq 0$

1000 test runs, but statement was executed only 4 times

Is $x \neq 0$ just a random effect?

Solution: Determine probability of non-random event:

$$1 - \left(1 - \frac{1}{|(-15) - 15|}\right)^4 \approx 0.13$$

If probability is greater than threshold ⇒ show invariant

Also: always show the number of events that *support* the invariant (4)

Effect: Higher trust in invariants.

Verifying Correctness

Problem: Finite number of test runs $\Rightarrow$ invariants are not proven to hold for *all* runs

Solution: Verify invariants with static analysis

Effect: Provably correct invariants

Daikon's Efficiency

Daikon's run time costs depend on

- i given invariants—$O(i)$
- v variables per execution step—up to a triple per invariant—$O(v^3)$
- t test cases (program runs)—$O(t)$
- p places in the program to be instrumented—$O(p)$

Overall run time: $O(i \times v^3 \times t \times p)$

This limits the size of programs to be analyzed!

Invariants as Anomalies

Basic approach:

- Determine invariants for a set of *passing* runs
- Determine *invariant violations* for a set of *failing runs*
- Focus on violations when searching for failure causes.

Example:

- `size = size(argc) - 1` holds in all passing runs, but
- `size = size(argc)` holds in all failing runs

⇒ focus on `size` as a possible infection!

Checking Invariants with Diduce

Diduce = Dynamic Invariant Detection ∪ Checking Engine

- works during the execution of the program
- determines invariants on the fly
- detects invariant violations
- adapts invariants automatically
- built for efficiency

Training and Checking

Diduce works in two modes:

Training mode.

Goal: find possible invariants

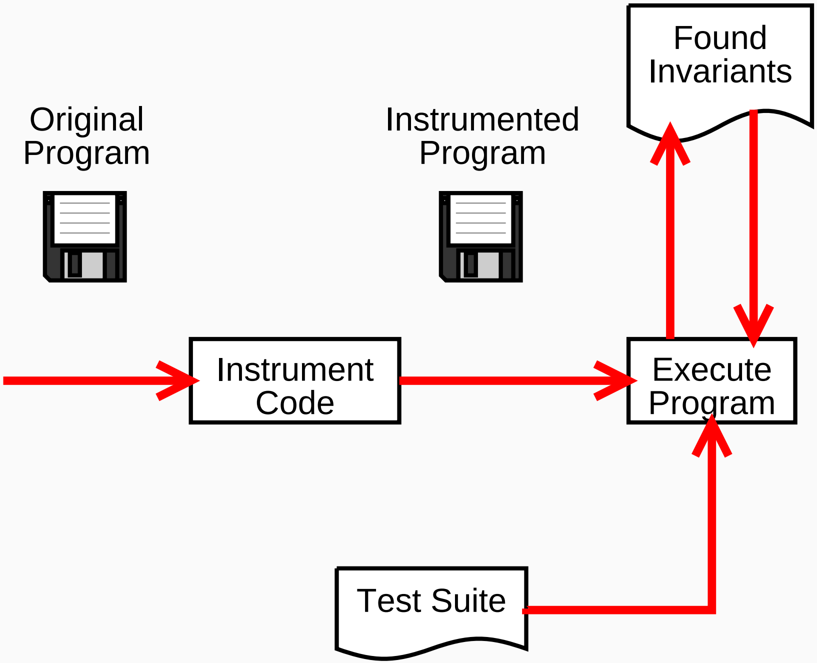
Requires a test suite that is known to work

Checking mode.

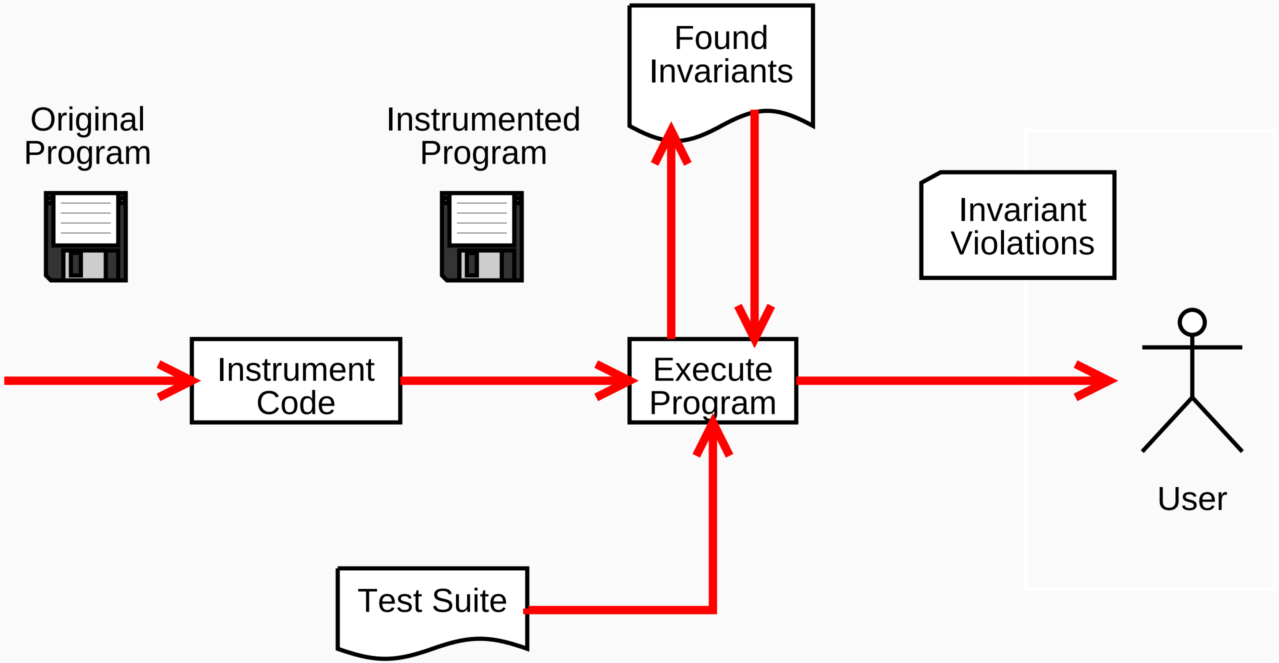
Goal: find possible violations of invariants

Requires a failing test suite or a random test suite

Training Mode



Checking Mode



Instrumenting Code

Instrument execution steps.

- Read/write accesses on object
- Read/write accesses on static variable
- Method calls

Add code.

- Test invatiant with current values
- Report invariant violation
- Adapt invariant

Diduce's Invariant Algorithm

The set `invariants` holds the currently valid invariants

for each *execution step*:
    for each *variable* at *execution step*:
        if ¬exist invariants[*variable*]:
            invariants[*variable*] = ⟨constant⟩
        else:
            if value(*variable*) violates *invariants*[*variable*]:
                adjust invariants[*variable*]

Diduce data structure

For each instrumented place in the program, store

- the number of times the place was executed, and
- the found *value* of the variable
- the *difference* between the old value and the new value

Values and differences are stored as pairs `(int U, int M)`

- U is the initial value found (convert if necessary)
- `M` is a bit vector; ith bit is 0 if a difference was found in the ith bit

Diduce Example

Code	i	Value U	Value M	Difference U	Difference M	Invariant
`i = 10;`	1010	1010	...11111	0	...11111	$i = 10$
`i += 1;`	1011	1010	...11110	1	...11110	$10 \le i \le 11 \wedge \lvert i' - i \rvert \le 1$
`i += 1;`	1100	1010	...11000	1	...11110	$8 \le i \le 15 \wedge \lvert i' - i \rvert \le 1$
`i += 1;`	1101	1010	...11000	1	...11110	$8 \le i \le 15 \wedge \lvert i' - i \rvert \le 1$
`i += 2;`	1111	1010	...11000	1	...11100	$8 \le i \le 15 \wedge \lvert i' - i \rvert \le 3$

Diduce: Possible Invariants

Values.

- $M = \ldots 1111 \Rightarrow$ variable is constant (or reference points to same type)
- $U - \overline{M} \leq x \leq U + \overline{M}$
- If $M = \ldots 1 \Rightarrow x$ is even

Differences.

- $M = \ldots 1111 \Rightarrow$ variable is constant
- $\overline{M} \Rightarrow$ maximum difference
- Which bits are constant?

Diduce: Costs

Diduce's run time costs depend on

- v variables written per execution step $O(v)$
- t test cases (program runs)—$O(t)$
- p places in the program to be instrumented—$O(p)$

Overall run time: $O(v \times t \times p)$—a small constant overhead for each writing operation

Space requirements: *3 words per expression*

- 1 word per number of calls
- 2 words for variable value and difference

Diduce vs. Daikon

- ✔ efficient
- ✔ invariants are computed during execution (integration in debugging tool)
- ✘ smaller set of invariants (ranges and values)
- ✘ less precise invariants

Concepts

- Given a sufficient large number of passing test runs, one can effectively determine invariants that hold for all observed test cases
- Checking failing test cases against trained invariants of passing test cases can lead to data likely to induce a failure.
- Technique is easy to use; results are quite easy to interpret
- Increased precision (Daikon vs. Diduce) comes with higher costs for execution and space
- The determined invariants hold for the observed test cases only—not necessarily for *all* test cases.

References

- Michael Ernst et al., *The Daikon invariant detector*, http://pag.lcs.mit.edu/daikon/
- J. Sudheendra Hangal, Monica S. Lam, *Tracking Down Software Bugs using Automatic Anomaly Detection*, Proc. International Conference on Software Engineering, 2002. http://suif.stanford.edu/papers/