The Daikon system for dynamic detection of likely invariants

MIT Computer Science and
Artificial Intelligence Lab.

16 January 2007

Presented by Chervet Benjamin
Dec 2008 : System Modeling and Analysis

Contents:

The Daikon system for dynamic detection of likely invariants

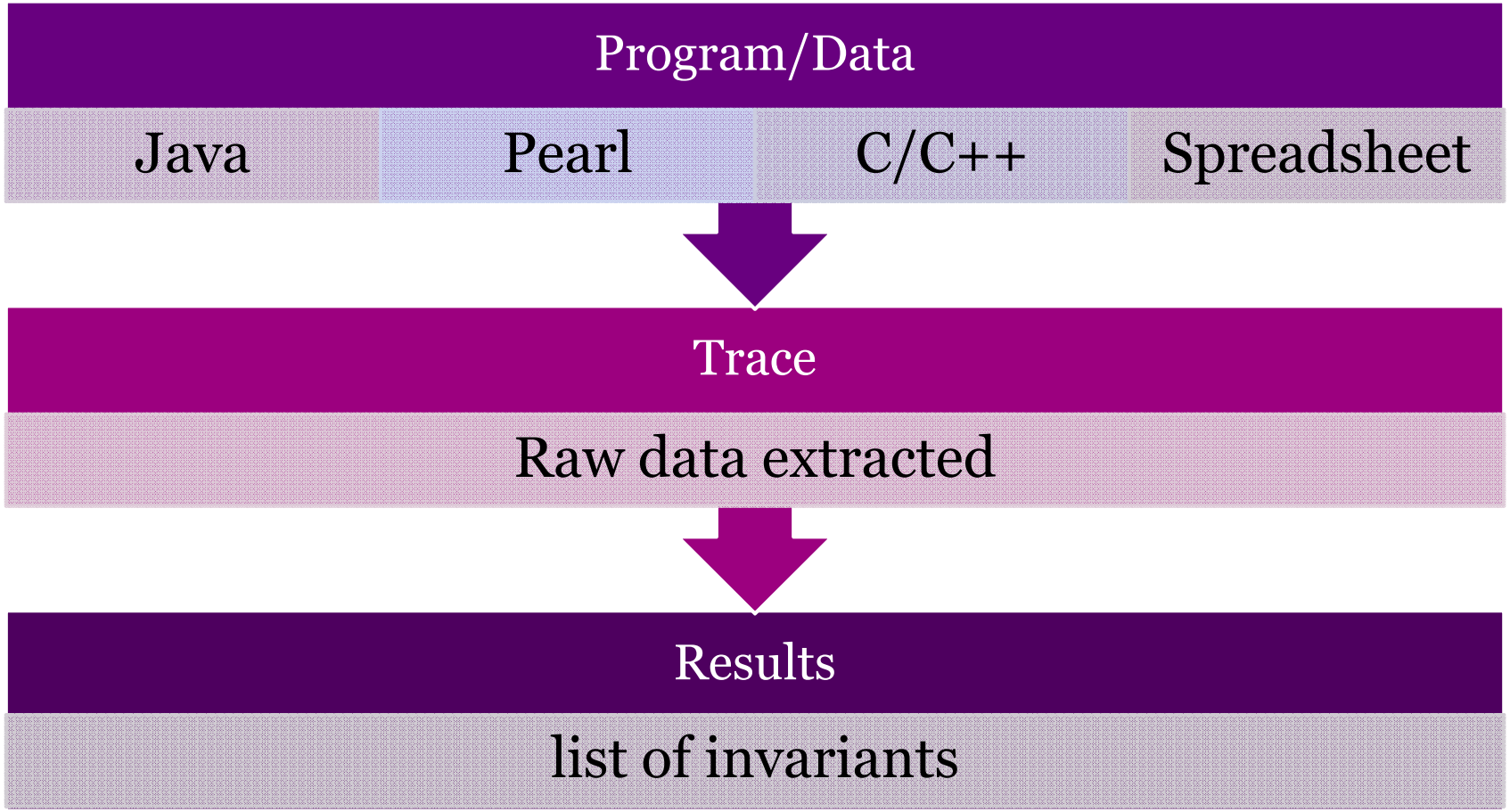
Invariants : property that holds at a certain point or points in a programm (ex: x==0, String.isObject==true, v>z…)

Dynamic detection : != static analysis : Need to run the program to generate the invariants. Will not build an automate or try all the cases.

Daikon : Chinese Radish and a software developed by the MIT, that dynamically detect likely invariants.

Daikon detects likely properties that hold at a certain point by running a programm.

How Daikon works ?



From the program to the trace

Java Code :

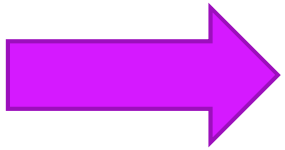
```
package perso;
public class test2{

	public static int l;
	public static int i;
	public static int useless;

public static void main(String args[]){
		test2.l=0;
		test2.i=0;
		while(10 >= test2.l){
			test2.incr();

		}
	}

	public static void incr(){
		test2.l++;
		test2.i++;
	}
}
```



Part of the trace :

```
perso.test2.incr():::ENTER
this_invocation_nonce
4
perso.test2.l
3
1
perso.test2.i
3
1
perso.test2.useless
0
1

perso.test2.incr():::EXIT26
this_invocation_nonce
4
perso.test2.l
4
1
perso.test2.i
4
1
perso.test2.useless
0
1
```

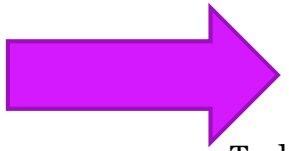
Introduction -> Short example.

From the trace to the invariants

Trace :

```
perso.test2.incr():::ENTER
this_invocation_nonce
4
perso.test2.l
3
1
perso.test2.i
3
1
perso.test2.useless
0
1

perso.test2.incr():::EXIT26
this_invocation_nonce
4
perso.test2.l
4
1
perso.test2.i
4
1
perso.test2.useless
0
1
```



Results :

To doDaikon version 4.5.1, released November 3, 2008;
Processing trace data; reading 1 dtrace file:
[20:52:35]: Finished reading test2.dtrace.gz

```
========================================
perso.test2:::CLASS
perso.test2.l == perso.test2.i
perso.test2.l >= 0
perso.test2.useless == 0
perso.test2.l >= perso.test2.useless
========================================
perso.test2.incr():::EXIT
perso.test2.useless == orig(perso.test2.useless)
perso.test2.l > perso.test2.useless
perso.test2.l - orig(perso.test2.l) - 1 == 0
perso.test2.useless <= orig(perso.test2.l)
========================================
perso.test2.main(java.lang.String[]):::ENTER
perso.test2.l == perso.test2.useless
perso.test2.l == size(arg0[])
arg0 has only one value
arg0.getClass() == java.lang.String[].class
arg0[] == []
arg0[].toString == []
```

Key features of Daikon

- Different input
  - Many programming languages
    - Java
    - C/C++
    - Pearl
  - Spreadsheet
    - CVS

Key features of Daikon

- Rich output
  - Grammar of Properties
    - 75 different invariants checks
    - Implications checking (a<3 => v > w)
  - Grammar of Variables
    - Parameters and return values checking
    - Pre state values (orig in short examples)
    - Global Variable, Pre State values, Fields
    - Can find asserts on variable that don't appear in the program
  - Multiple Program points checkings

Key features of Daikon

- Scalability
  - Used by the NASA on over 1 million lines of C/C++
- Invariant filtering
  - Redundants invariants (a – 1 >= 3 and a a-1 >2)
  - Do not display invariants between unrelated variables
- Portable
  - Run with Linux and Windows

Uses

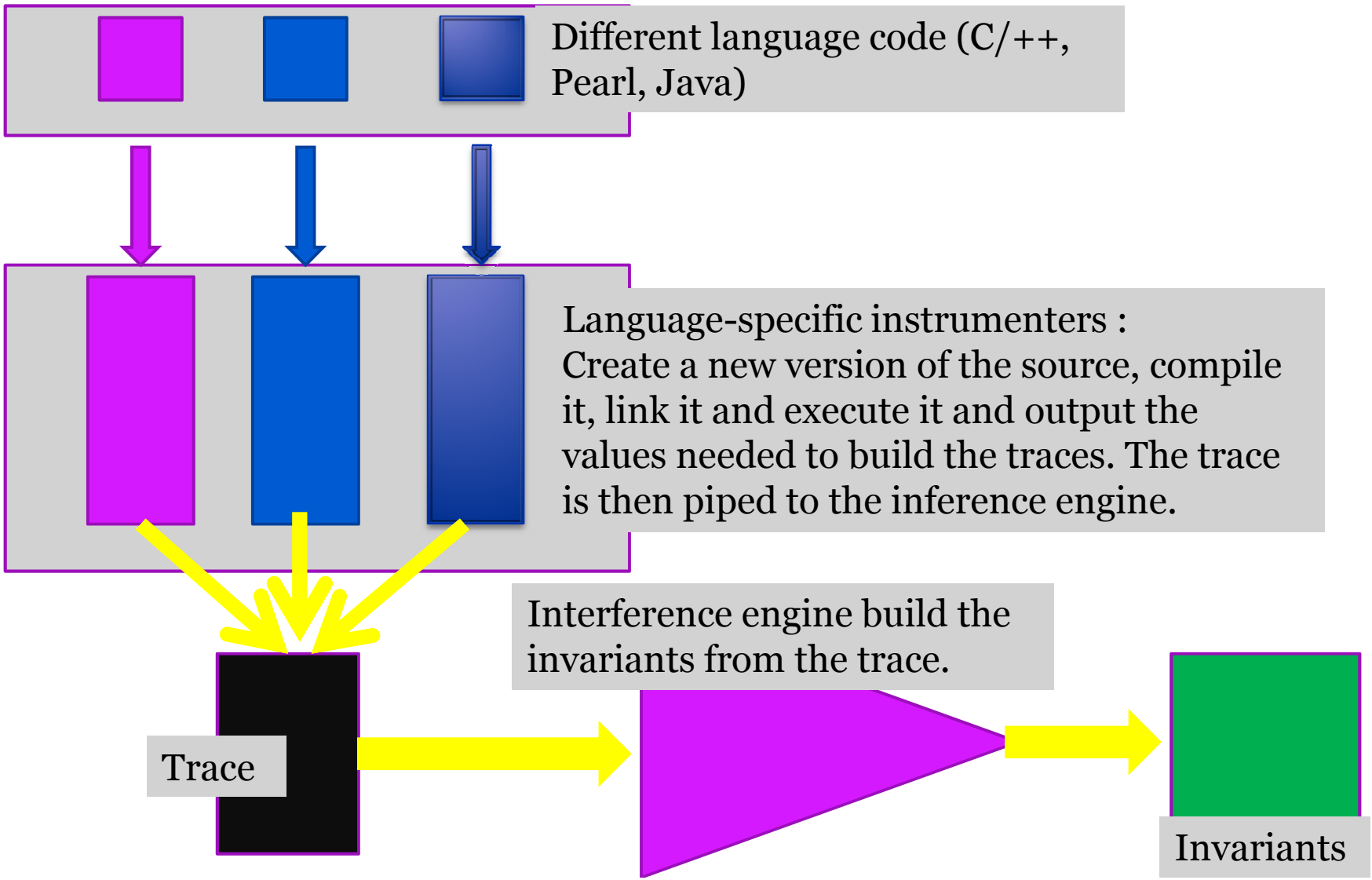
- Understanding an algorithm or the implementation
- Documentation
  - Always up to date
- Avoiding bugs
  - Check if the invariant stay true after a modification of the code
- Debugging
- Testing

Uses

- Verification
  - Are the specification true ?
  - 95% of the properties were provable with Java.
- Data structure and control repair.

II.Daikon in deep -> Uses.

Daikon architecture



Interference Engine

- Assume that all potential invariants are true
- Test each one against each sample
- Discard contradicted invariant
- A few optimizations needed to scale better.

Optimization : reduce the calculus amount by 99%.

- Equal variables
  - If an invariant is true for one variable, then it will be true if another variable is equal.
- Dynamically constant variables
  - If a variable has the same value at each observed sample, we do not need to perform the same test at each step. (Ex: if v = 2, in 3 steps, we do not need to make the test v >=0 for the 3 steps).
- Variable hierarchy:
  - Some variables contribute to invariant at multiple program points. If the same samples appears at two points of a programm, then the invariant at the first point and at the second point are the same.
- Suppression of weak invariants:
  - X > Y implies X >= Y

III.Synthesis Exemple

Simple stack with a fixed-maximum size:

```
Fields:
Object[] Array; // Array containing stacks elements
Int topOfStack; // top of the stack , -1 if the stack is empty.

Methods:

Void push(Object x) ; //Add an object at the top of the stack
Void pop(); // Remove the object at the top of the stack
Object top(); // Return (without removing) the objet at the top
Object topandpop(); //Remove and return the object at the top
Boolean isEmpty(); //
Boolean isFull();
Void makeEmpty(); // Clear the stacks.
```

III.Synthesis Exemple

Exemple of outputs and how it can be used :

Object invariants for the Class:

```
This.theArray != null
```

The array is never null. Methods can thus safely reference it without checking for null.

```
This.topOfStack >= -1
```

The topOfStack is never smaller than -1. It works as the programmer intended to make it work.

```
This.theArray[this.topOfStack+1..] elements = null
```

The Array is empty for the case above the top Of the case. It infers that the pop method nulls out the entry after returning it.

Post conditions for the StackAr constructor :

```
Orig(capacity) = this.theArray.length
```

Respect the specification. The capacity of the array is always equal to designed capacity.

Summarize :

- Invariants are very useful, but hard to find.
- Daikon detects the invariants dynamicaly
  - Do not check for every possible values.
- May gives not true invariants :
  - True for a particular test suits but not for another.
- But : Easier to reject false invariants that to guess the true ones.