Using Daikon to automatically estimate the number of executed instructions - Internal preliminary Report

Victor Braberman [2] Diego Garbervetsky [1] Sergio Yovine [3]

Abstract

We present a proof of concept that combines static and dynamic analysis in order to obtain symbolic expression that over-approximates the number of times an statement is executed in object oriented languajes like Java. The tool leverages existing dynamic analysis tools like Daikon, but guides it (via programm instrumentation) in order to obtain linear invariants that, after some tranformations, are useful to generate the symbolic expressions. We show how this tool can be used to over-estimate memory consumption in Java applications.

Key words: Program guidance, byte instrumentation,
Java,Run-time analysis.

1 Introduction

We present a proof of concept that combines static and dynamic analysis in order to obtain symbolic expression that over-approximates the number of times an statement is executed in object oriented languajes like Java. The tool leverages existing dynamic analysis tools like Daikon, but guides it (via programm instrumentation) in order to obtain linear invariants that, after some tranformations, are useful to generate the symbolic expressions.

We use a technique based on computing linear invariants that relate program variables to the number of times a statement is executed. Roughly speaking, this is the number of integer points that satisfy the invariant. This number is given in a parametric form as a polynomial where unknonws are

[1] *School of Computer Science, Universidad de Buenos Aires, Argentina.* E-mail: `diegog@dc.uba.ar`. Partially supported by projects ANCyT grant PICT 11738 and IBM Eclipse Innovation Grants.

[2] *School of Computer Science, Universidad de Buenos Aires, Argentina.* E-mail: `diegog@dc.uba.ar`. Partially supported by projects ANCyT grant PICT 11738 and IBM Eclipse Innovation Grants.

[3] *VERIMAG, France.* E-mail: `sergio.yovine@imag.fr`. Partially supported by projects DYNAMO (Min. Research, France) and MADEJA (Rhône-Alpes, France).

method input parameters. Our method does not require annotating the program in any form and does produce non-linear parametric upper-bounds. The polynomials are to be evaluated on program (or method) inputs to obtain the actual bound.

The approach combines techniques used for performance analysis [11], cache analysis [4], data locality [17], worst case execution time analysis [16], and memory optimization [14,25].

2 Preliminaries

2.1 Notation for Programs

We define a program to be a set $\{C_0, C_1, \ldots\}$ of classes. Each class C has a set $\{C.m_0, C.m_1, \ldots\}$ of $Methods$. A method has a list P_m of parameters and a sequence of statements denoted by $Body_m$. Each statement in a program is identified with a $Label =_{def} Method \times I\!N$ which uniquely characterizes its location by means of the stm mapping ($stm : Label \rightarrow Statement$). When $stm(l)$ is a call statement $args_l$ will denote the list of the call arguments.

A *Call Graph* of a method m is a directed graph $CG_m =< N, E >$ where $N = Methods$ represents the program methods and $E = (Methods \times Label \times Methods)$ represents the call relation. $(c, l, m) \in E$ means that the method c, at location l, calls method m. For simplycity, we will assume that we can determine, by static analysis, for each call, exactly which method will be invoked, not being able to have more than one possible invocable method.

When CG_m is a DAG, a finite *Call Tree* $CT_m =< N, E >$ can be obtained by unfolding the call graph. This unfolding is done by cloning the nodes that have more than one parent. $N = Methods_{CT} = Label^+ \times Method$ represents the path from the root node and $E = (Methods_{CT} \times Label \times Methods_{CT})$

Example
In Figure 1 we present one motivating example. It is a (very simple) implementation of a dynamic array using a list of fixed sized nodes. We are interested in the allocation statement located at newBlock.7. The number of times this statement is executed when execution start by method `addAll` depends on the size of the collection passed as a parameter. The execution of this statement takes place in the method where a new block of memory is request because the previous block is full. The Call Graph and Call Tree for method $m0$ are depicted in Figure 2. □

$CallChain_m$ denote the set of possible paths trough the call graph starting from method m (i.e, following its egdes).

The $CallChain$ sets for our example are the following:

```
public class ArrayDim {
  Vector list; int len;
  final static int BSIZE = 5;
  public ArrayDim() {
1:  list= new Vector();
2:  len = 0;    }
  public void add(Object o) {
1:  Object[] block;
2:  if (len % BSIZE == 0)
3:    block = newBlock(BSIZE);
    else
4:    block=(Object [])
            list.lastElement();
5:  block[len % BSIZE] = o;
6:  len++;
}
  Object[] newBlock(int how) {
7: Object[] block=new Object[how];
8: list.add(block);
9: return block;
  }
  void addAll(Collection c) {
10: for(Iterator it=c.iterator();
            it.hasNext();)  {
11:    add(it.next());
       }
  }
}
```

Fig. 1. Motivating example

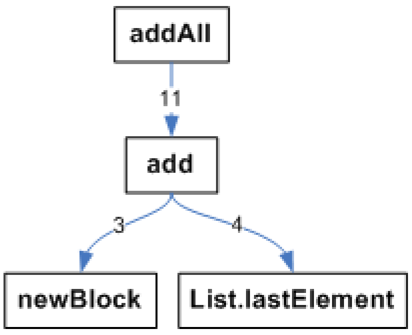

Fig. 2. Call Graph for method *ArrayDim.addAll* of the proposed example

$$CallChain_{addAll} = \{addAll.11, addAll.11.add.3, addAll.11.add.4\}$$

$$CallChain_{add} = \{add.3, add.4\}$$

$$CallChain_{newBlock} = \{\}$$

$$CallChain_{List.lastElement} = \{\}$$

A *control flow graph* (CFG) is a directed graph $G =< N, E, entry, exit >$ where N is the set of nodes and E is the set of edges. *entry* and *exit* are specials nodes indicating unique start and ending points. Given a method m, G_m is the CFG of m which includes transitively the CFG of every method that m calls. Each node $n \in N$ corresponds to one statement and has a label $l \in Label^+$. Notice that, since a called method is macro-expanded in the control flow graph each time it is invoked, labels are composed by the corresponding path in CG_m and its relative location.

We call *Instrumentation Site* every place (defined by its *Label*) of the program we want to analyze. We will define IS as the set of Instrumentation Sites.

IS_m denotes the set of instrumentation sites reachable from the entry point of the method m control flow graph.

$$IS_m = \{l \in \mathit{Label}^+ . l = cc.is \wedge cc \in \mathit{CallChain}_m \wedge is \in IS\}$$

Assuming that $IS = \{\mathit{newBlock}.7\}$:

$$IS_{\mathit{addAll}} = \{\mathit{addAll}.11.\mathit{add}.3.\mathit{newBlock}.7\}$$

$$IS_{\mathit{add}} = \{\mathit{add}.3.\mathit{newBlock}.7\}$$

$$IS_{\mathit{newBlock}} = \{\mathit{newBlock}.7\}$$

2.2 Symbolic analysis

An *invariant* for a program location is an assertion over the program variables that holds whenever this location is reached in every program run. For our work we consider an invariant $\mathcal{I}$ over the set of variables V as a set of linear or non-linear constraints over V.

Given a method m and a program label l, $\mathcal{I}_l^m$ denotes an invariant predicate over the program variables, for the control flow graph of the method m, at the node labelled l.

There is a lot of research on symbolic analysis techniques can be used to compute invariants. For example, [7,6] propose an approach based on abstract interpretation [5], where the invariants are convex polyhedrons; [18] proposes the use non-linear constraint solving for obtaining linear invariants; [12] proposes methods based on symbolic evaluation that can handle some non-linear constraints. For java, languages for describing assertions are becoming popular (e.g. JML [15]) and several method for synthesizing invariants were proposed (for instance, [19],[10] and [13]).

In this paper, invariants will be generated using `Daikon` [10]. The middle column of Table 1 shows some global invariants for some instrumentation points reachable from method *addAll*.

is	$\mathcal{I}_{\mathbf{cs}}^{\mathbf{m0}}$	$\mathcal{C}(\mathcal{I}_{\mathbf{cs}}^{\mathbf{m0}}, \mathbf{P}_{\mathbf{m0}})$
`addAll.11`	$\{it.counter >= 0, it.counter < size(c), this.len = it.counter\}$	$size(c)$
`addAll.11.add.5`	$\{it.counter >= 0, it.counter < size(c), this.len = it.counter\}$	$size(c)$
`addAll.11.add.3.newBlock.7`	$\{it.counter >= 0, it.counter < size(c), this.len = it.counter, this.len \% BSIZE = 0\}$	$\frac{1}{5}size_c + (per(size_c, [0, \frac{4}{5}, \frac{3}{5}, \frac{2}{5}, \frac{1}{5}])$

Table 1
Some invariants and Ehrhart polynomials for `addAll`

Given set of constraints $\mathcal{I}$, over a set of variables $V = W \uplus P$, $\mathcal{C}(\mathcal{I}, P)$ denotes a symbolic expression over P that provides the *number of integer solutions* for the remaining variables W. More precisely, $\mathcal{C}$ yields an expression which is equivalent to $\lambda \vec{p}.(\ \#\{\ \vec{w} \mid \mathcal{I}\,[W/\vec{w}, P/\vec{p}\,]\ \}\ )$.

In general the resulting expression has the form:

$$\sum_{1 \leq i \leq k} \gamma(Cond_i(P)) * E_i(P)$$

where γ is such that $\gamma(Cond_i(P)) = 1$ if $Cond_i(P) = \text{TRUE}$, 0 otherwise.

There are several techniques that obtain these symbolic expressions [3,11,20]. In [3], E_i's are polynomials (called Ehrhart polynomials [9]) whose coefficients vary depending on the parameters' values. In [11], E_is are integer-valued symbolic expressions that may consist of arbitrary divisions, multiplications, subtractions, additions, exponentiations, and maximum and minimum. Its operands can be parameters, integer constants and infinity symbols (∞, $-\infty$).

The right column of Table 1 shows the Ehrhart polynomials that express the number of times the statements is potentially executed for some instrumentation points reachable from method *addAll* in terms of its parameters.

3 Analysis Phases

Our aim is to compute a function that given a method m, it yields a symbolic expression that over-approximates the number of times that a selected set of statements are executed assuming execution starts at m. As detailed in [23] it can be performed as stated in the following algorithm:

```
computeInstances(m, IS)
// returns an Expression (over $P_m$)
res:=0;
for each $is \in IS_m$ do
    get $\mathcal{I}_{is}$;
    instances:=$\mathcal{C}(\mathcal{I}_{is}, P_m)$;
    res:=res + instances;
end for;
return res;
```

The algorithm receives a method and a set of instrumentation sites. Then, they obtain a global invariant for each path in IS_m. Finally, the invariant is used to generate the symbolic expression.

In this section we will describe how to obtain the global invariant $\mathcal{I}_{is}$. It will be described in terms of the program variables, constants and method's parameters. If the creation is involved in one or several loops, the induction variables (i.e., those which have different values at different loop iterations) will appear in the invariant. The induction variables will belong to the set of free variables which are used to count the number of solutions. These variables have a direct impact on the counting phase because they influence the number of integer solutions for the obtained invariant.

3.1 Phase 1 - Program Instrumentation

In this phase, the application is instrumented in order to generate a new code that behaves as the original one but assists `Daikon` [19] in obtaining linear invariants. As `Daikon` discovers **local** invariants at method's entry and exit points and we are interesting in finding **global** invariants that reflex the valid programs estates at the selected program points, a trick has to be performed. This consist in generate dummy methods which has, as parameters, the variables we want to analyze. Then, we add a call to this method in program point subject to analysis with the appropriate arguments. We also need to instruments all call sites, in order to afterwards bind all call chain local invariant and generate the global one.

We are not interested in all program variables. We only need the ones that impact in the number of execution of the desired statements. Then, in order to avoid over-counting, we had to choose only the inductives variables.

We give a special treatment to iterator-driven cycles over collections. In order to obtain linear invariants for this class of cycle, we instrument the code in order to create a "counter" that is associated with the iterator and is incremented each time a `Iterator.next` is executed.

Finally, we need that linear invariants. To assist `Daikon` in finding them, we adapt some variables that are not of Integer type (Collections, Strings, Arrays) instrumenting information about their sizes.

We perform the following task:

Let $IS \subseteq Label$, $Var = \{v_1, v_2, \ldots\}$ be the set of variables.

Given a class C, each method m is instrumented according to the following algorithm:

```
instrumentMethod($m$)
// instruments method $m$
// returns the set of created dummy method
$\varepsilon = \emptyset$;
$IMs = \emptyset$;
initCode:=codeInitParams($m$);
insert($m$,initCode);
for each $l \in Body_m$ do
    $(im_l, \varepsilon')$= gen($l, \varepsilon$);
    code = instrument($l, im_l, \varepsilon'$);
    insertBefore(m,l,code);
    $IMs = IMs \cup im$
    $\varepsilon = \varepsilon'$;
end for;
return $IMs$;
```

The algorithm first instruments code to record the original value of the method parameters. They will be used later, when binding invariant via the assignment of parameters with arguments. Then, for each statement in the method's body generates code that will be inserted before.

All resulting set of methods IMs are appended to a class, that will be passed to `Daikon` together with the application. The invariant at each im_l entry point will be the corresponding local invariant at location l.

The function gen : $Label \to Method \times \mathbb{P}(Var)$ is defined as follows (assuming $l = (C.m, n)$):

gen(l , ε)=$(\bot, \varepsilon \oplus \texttt{count}_{\texttt{it}})$ if stm(l)=`itDef it`

gen(l ,ε) =(im, ε) if stm(l)=`call` m'

where p_{im} = sizeParams($p_m \cup pInit_m \cup$ inductives$(l) \cup args_{\texttt{call}_{m'}} \cup \varepsilon)$

gen(l, ε) =(im, ε) if $l \in IS$

where p_{im} = sizeParams($p_m \cup pInit_m \cup$ inductives$(l) \cup \varepsilon)$

gen(`l`, ε)=$(\bot, \varepsilon)$ if $l \notin IS$

$pInit_m = \{p\_init \cdot p \in p_m\}$ refers to new special variables we use keep a copy of their original value. $pInit_p$ refers to "init variable that corresponds to p.

The function `sizeParams` is responsible for obtaining the integer representatives of a list of variables.

$\texttt{sizeParams}(pl) = \bigcup_{v \in (pl)} \texttt{sizeVar}(v)$

The function `sizeVar` takes a variable of any type and yields its integer representatives.

$$\texttt{sizeVar}(v) = \begin{cases} \{size_v\} & \text{if } \texttt{type}(v) \in \texttt{Sizeable} \\ \{v\} & \text{if } \texttt{type}(v) = \texttt{Int} \\ \bigcup_{f \in \texttt{fields}(v)} v\_\texttt{sizevar}(f) & \text{if } \texttt{type}(v) \sqsubseteq \texttt{Object} \end{cases}$$

$v\_\texttt{sizevar}(f)$ is a notation to refer the set variables with is equal to $\texttt{sizevar}(f)$ but adding a "$v\_$" to the name of each variable in the set.

The function `codeInitParams` $Method \to List[stm]$ takes a method a yields the code that initializes the value of each variable in the $pInit$ set.

$\texttt{codeInitParams}(m) = \bigsqcup_{p \in p_m} pInit_P = p\ .\ \texttt{codeForParams}(pInit_m)$

The function instrument : $Label \times Method \to List[stm]$ is defined as follows (assuming $l = (C.m, n)$):

instrument(l, $\perp$, ε) =[$\texttt{count}_{\texttt{it}}$=0] if stm($l$)=`itDef it`

instrument(l, $\perp$, ε) =[$\texttt{count}_{\texttt{it}}$++] if stm($l$)=`itNext it`

instrument(l, im, ε)=instr($p_m \cup$ inductives(l) $\cup$ $args_l \cup \varepsilon$) if stm(l)=`call` m'

instrument(l, im, ε)=instr($p_m \cup$ inductives(l) $\cup$ ε) if $l \in IS$

instrument(l, im, ε)=[] if $l \notin IS$

instr(pl, im) =codeForParams(pl).call im(`sizeParams`(pl))

The function `codeForParams` is responsible for obtaining the code necessary to reflex its ”sizeable” information of a list of variables.

$$\text{codeForParams}(pl) = \bigsqcup_{v \in pl} \texttt{codevar}(v)$$

The function `codeVar` takes a variable of any type and yields the corresponding code necessary to reflex its ”sizeable” information.

$$\texttt{codevar}(v) = \begin{cases} \{\texttt{if}(v \neq null) codeSize(v)\ \texttt{else}\ size_v = 0\} & \text{if } \texttt{type}(v) \in \texttt{Sizeable} \\ \{\} & \text{if } \texttt{type}(v) = \texttt{Int} \\ \bigsqcup_{f \in \texttt{fields}(v)} \{\texttt{if}(v \neq null) codeVar(v)\ \texttt{else}\ size_f = 0\} & \text{if } \texttt{type}(v) \sqsubseteq \texttt{Object} \end{cases}$$

$$\texttt{codeSize}(v) = \begin{cases} size_v = v.size() & \text{if } \texttt{type}(v) \sqsubseteq \texttt{Object} \\ size_v = v.length & \text{if } \texttt{type}(v) = \texttt{Array} \\ size_v = v.lengt & \text{if } \texttt{type}(v) = \texttt{String} \end{cases}$$

Table 2 shows part of the instrumented code for the example. The set if selected instrumented sites is $IS = newBlock.7$

3.2 Phase 2 - Finding local invariants

As we have explained previously, we use `Daikon` to obtain de local invariants using the instrumented application and the new class of dummy methods. We ask `Daikon` to only find invariants for this class. The obtained invariants will have the information about all instrumented sites in the original code. In order to formalize this procedure we define a function (that will be performed by `Daikon`) that given an instrumentation site or a call site it will provide a local invariant:

`invariant`: $Label \rightarrow Invariant$ $(\forall l \in Label.l \in IS \vee l \in CallSite)$

Table 3 shows some obtained invariants for our example. They correspond to the instrumentation site `newblock.7` and the call sites `addAll.7` and `add.3` that belong to its call chain.

```
public class ArrayDim {
  Vector list; int len;
  final static int BSIZE = 5;
  public void add(Object o) {
    Object[] block;
    ArrayDim this_init=this;
    int this_init_list_size, this_init_len;
    int this_list_size, this_len;
    int ArrayDim_BSIZE;
    if(this_init!=null) {
      if(this_init_list!=null)
        this_init_list_size=this_init_list.size();
      else this_init_list_size = 0;
      this_init_len = this_init.len; }
    else { this_init_list_size = 0;
    this_init_len = 0; }
    if (len % BSIZE == 0) {
      ArrayDim_BSIZE = BSIZE;
      if(this!=null) {
        if(this_list!=null)
          this_list_size = this_list.size();
        else this_list_size = 0;
        this_len = this.len; }
      else { this_list_size = 0;
        this_len = 0; }
      IM.ArrayDim_3(this_list_size,this_list_len,
        this_init_list_size, this_init_len,
        ArrayDim_BSIZE);
      block = newBlock(BSIZE);
    }
    else {
      block=(Object [])list.lastElement(); }
      block[len % BSIZE] = o;
      len++; }
  }
  Object[] newBlock(int how) {
    int how_init = how;
    ArrayDim this_init=this;
    int this_init_list_size,this_init_len;
    int this_list_size,this_len;
    int ArrayDim_BSIZE;
    this_init = this;
    if(this_init!=null) {
      if(this_init_list!=null)
        this_init_list_size=this_init_list.size();
      else this_init_list_size = 0;
      this_init_len = this_init.len; }
    else { this_init_list_size = 0;
      this_init_len = 0; }
    ArrayDim_BSIZE = BSIZE;
    if(this!=null) {
      if(this_list!=null)
        this_list_size = this_list.size();
      else this_list_size = 0;
      this_len = this.len; }
    else { this_list_size = 0;
      this_len = 0; }
    IM.ArrayDim_7(how, how_init,
      this_list_size,this_list_len,
      this_init_list_size, this_init_len,
      ArrayDim_BSIZE);
      Object[] block=new Object[how];
      list.add(block);
      return block;
    }
  void addAll(Collection c) {
    Collection c_init = c;
    int c_size, c_init_size;
    if(c_init!=null) c_init_size = c_init.size();
    else c_init_size = 0;
    ArrayDim this_init;
    int this_init_list_size, this_init_len;
    int this_list_size, this_len;
    int ArrayDim_BSIZE;
    this_init = this;
    int it_count;
    if(this_init!=null) {
      if(this_init_list!=null)
        this_init_list_size=this_init_list.size();
      else this_init_list_size = 0;
      this_init_len = this_init.len; }
    else { this_init_list_size = 0;
      this_init_len = 0; }
    it_count = 0;
    for(Iterator it=c.iterator();
        it.hasNext();) {
      if(c!=null) c_size = c.size();
      else c_size = 0;
      it_count++;
      ArrayDim_BSIZE = BSIZE;
      if(this!=null) {
        if(this_list!=null)
          this_list_size = this_list.size();
        else this_list_size = 0;
        this_len = this.len; }
      else { this_list_size = 0;
        this_len = 0; }
      IM.ArrayDim_11(it_count, c_size, c_init_size,
        this_list_size,this_list_len,
        this_init_list_size, this_init_len,
        ArrayDim_BSIZE);
      add(it.next());
    }
  }
}
```

Table 2
Instrumented code for the example

3.3 Phase 3 - Generating global invariants

We generate a global invariant for each instrumentation site. To do that, we process all the call chains from m to each reachable method that contains creations sites. Then, for every chain a global invariant is generated by bind-

label	invariant
addAll.11	$BSIZE = 5, size_{f\_this\_init\_list} = 0, f\_this\_init\_len = 0, size_{c\_init} = size_c, size_{f\_this\_list} >= 0, size_{f\_this\_list} < size_c, f\_this\_len >= 0, f\_this\_len < size_c, size_{f\_this\_list} <= f\_this\_len, count\_it = f\_this\_len + 1, count\_it >= 1, count\_it <= size_c$
add.3	$BSIZE = 5, size_{f\_this\_list} = size_{f\_this\_init\_list}, f\_this\_len = f\_this\_init\_len, f\_this\_len\%5 = 0, size_{f\_this\_list} <= f\_this\_len, size_{f\_this\_list} < 5, f\_this\_len = (size_{f\_this\_list} * 5)$
newBlock.7	$BSIZE = 5, size_{f\_this\_list} = size_{f\_this\_init\_list}, f\_this\_len = f\_this\_init\_len, how = how\_init, f\_this\_len\%5 = 0, how = 5, size_{f\_this\_list} <= f\_this\_len, size_{f\_this\_list} < how, f\_this\_len = (size_{f\_this\_list} * how)$

Table 3
Local invariants found by `Daikon` for two call sites and one instrumentation site

ing the local invariant of each call site (they can be many) and the creation site,using a technique similar to [6].

To perform the call chain binding we associate the caller arguments with the callee formal parameters. As the parameters value could have changed, we use the artificially included $pInit$ that represent the original value of the formal parameters. Given $(l = cc.is \in IS_m)$, a call chain that finish with an instrumentation site, $\mathcal{I}$ is defined as follows:

$\mathcal{I}$: $Label^+ \rightarrow Invariant$

$$\mathcal{I}_{cc.is} = \bigsqcup_{i=1..length(cc)} \Big(\mathtt{inv}(cc[i]) \cup \mathtt{invCall}(args_{cc[i]}, pInit_{m_{cc.is[i+1]}}) \Big) \cup \mathtt{inv}(is)$$

$$\mathtt{invCall}(args, params) = \bigcup_{i=1..length(args)} \{args[i] = params[i]\}$$

Tengo que decir que en realidad los invariantes de cada is son renombrados antes (para evitar problemas de variables con el mismo nombre), y el binding de parametros los engancha.

4 The tool

Given a Java application, a method m to analyze and a set of statements to analyze, the tool generates a polynomial (over the method m parameters) that over estimate the number of times that this statements would be potentially executed.

The tool is composed by different components that are showed in figure 3.

(i) Code instrumentation (using `soot` [21]): It applies the technique explained in section 3.1.

The tool automatically (and conservatively) discover inductive variables for all instrumentation and call sites. By the moment we over-approximates them by using the live variables at the instrumentation point. The tools allow us to filter non desired variables via configurations files.

(ii) Local invariants computation using `Daikon` [19]. The obtained invari-

invariant
$\mathbf{l11@size_{f\_this\_init\_list} = size_{f\_this\_list}, l11@f\_this\_init\_len = f\_this\_len, l11@size_{c\_init} = size_c}$
$BSIZE = 5, l11@size_{f\_this\_init\_list} = 0, l11@f\_this\_init\_len = 0, l11@size_{c\_init} = l11@size_c, l11@size_{f\_this\_list} >= 0, l11@size_{f\_this\_list} < l11@size_c, l11@f\_this\_len >= 0, l11@f\_this\_len < size_c, l11@size_{f\_this\_list} <= l11@f\_this\_len, l11@count\_it = l11@f\_this\_len + 1, l11@count\_it >= 1, l11@count\_it <= l11@size_c$
$\mathbf{l3@size_{f\_this\_init\_list} = l11@size_{f\_this\_list}, l3@f\_this\_init\_len = l11@f\_this\_len,}$
$BSIZE = 5, l3@size_{f\_this\_list} = l3@size_{f\_this\_init\_list}, l3@f\_this\_len = l3@f\_this\_init\_len, l3@f\_this\_len\%BSIZE = 0, l3@size_{f\_this\_list} <= l3@f\_this\_len, l3@size_{f\_this\_list} < BSIZE, l3@f\_this\_len = (l3@size_{f\_this\_list} * BSIZE),$
$\mathbf{l7@size_{f\_this\_init\_list} = l3@size_{f\_this\_list}, l7@f\_this\_init\_len = l3@f\_this\_len, l7@how\_init = 5,}$
$BSIZE = 5, l7@size_f\_this\_list = l7@size_f\_this\_init\_list, l7@f\_this\_len = l7@f\_this\_init\_len, l7@how = l7@how\_init, l7@f\_this\_len\%BSIZE = 0, l7@how = BSIZE, l7@size_{f\_this\_list} <= l7@f\_this\_len, l7@size_{f\_this\_list} < l7@how, l7@f\_this\_len = (l7@size_{f\_this\_list} * l7@how)$
$BSIZE = 5, count\_it >= 1, count\_it <= size_c\, count\_it = f\_this\_len + 1, f\_this\_len\%BSIZE = 0$

$\mathtt{C}(\mathcal{I}^{addAll}_{newBlock.7}) = \frac{1}{5} size_c + (per(size_c, [0, \frac{4}{5}, \frac{3}{5}, \frac{2}{5}, \frac{1}{5}])$

Table 4
Original and simplified global invariant for the call chain and the counting expression for `addAll.11.add.3.newBlock.7`

ants for the generated dummy methods reflex the **local** invariant of each creation or call site.

(iii) Global invariants computation: We generate a global invariant for each instrumentation site in IS_m. To do that, we process all the call chains from m to each reachable method that contains Instrumentation sites. Then, for every chain a global invariant is generated by binding the local invariant of each call site (they can be many) and the creation site, using a technique similar to [6].

(iv) Invariant simplification and computation of polynomials : Invariants are simplified using the symbolic calculator SPPoC [1]. Finally, we use [24] to generate Ehrhart polynomial for each instrumentation site.

The experiments were carried out on a significant subset of programs from JOlden [2] and JGrande [8] benchmarks. In each one select the places in the application where there are memory allocation (i.e. $IS = \{l \ Label \dot{s}tm(l) = \texttt{new}\}$. For each example, we analyzed the most relevant method according to the number of allocations points. Although our current prototype does not handle recursion in general, it is able to deal with some recursive patterns such as tail recursion.

Table 5 shows the calculated polynomials and a comparison between real executions and estimations obtained by evaluating the polynomials with the corresponding values of parameters.

The last column shows the relative error ((#Obs - Estimation)/Estimation).

The studies showed that the technique was indeed very accurate, actually yielding exact figures in most benchmarks. In some cases, the over-approximation was due to the presence of allocations sites associated with

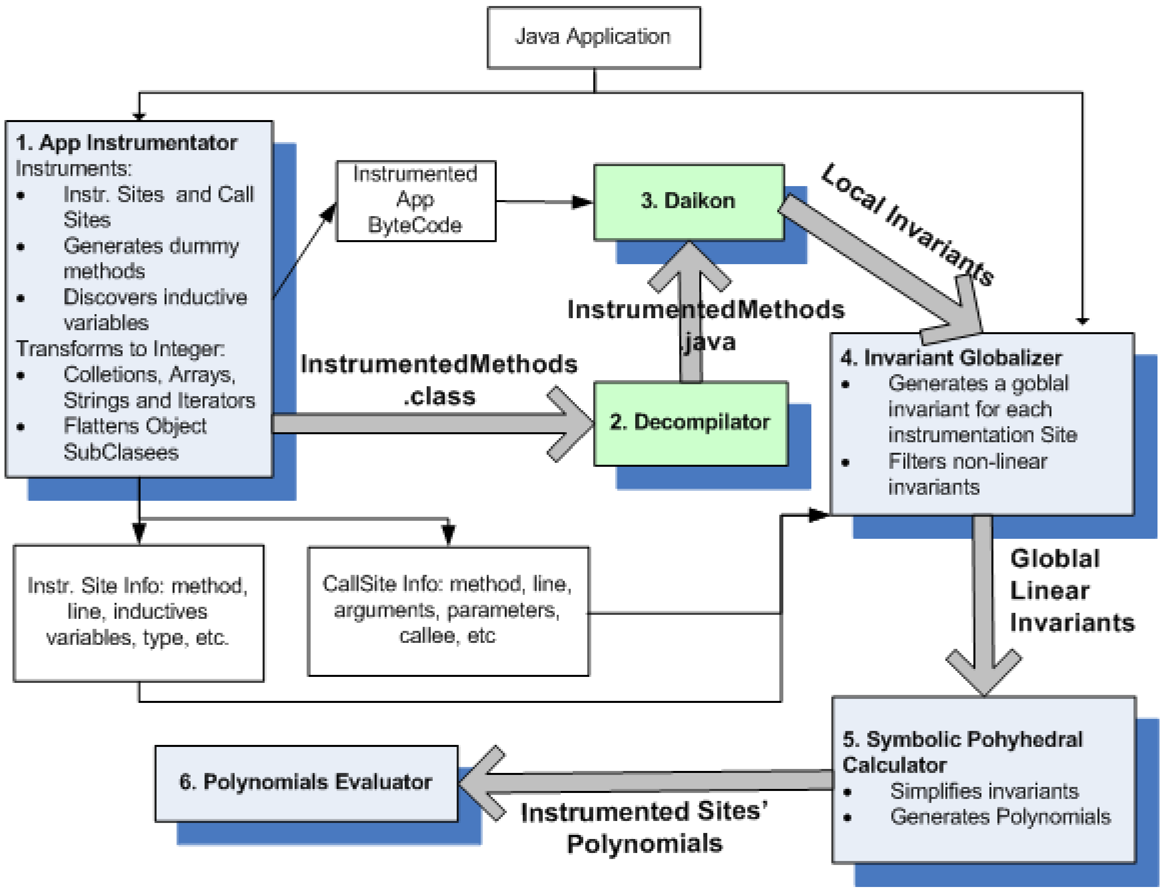
Java Application
1. App Instrumentator
Instruments:
Instr. Sites and Call Sites
Generates dummy methods
Discovers inductive variables
Transforms to Integer:
Colletions, Arrays, Strings and Iterators
Flattens Object SubClasees
Instrumented App ByteCode
3. Daikon
Local Invariants
InstrumentedMethods .java
InstrumentedMethods .class
2. Decompilator
4. Invariant Globalizer
Generates a goblal invariant for each instrumentation Site
Filters non-linear invariants
Globlal Linear Invariants
Instr. Site Info: method, line, inductives variables, type, etc.
CallSite Info: method, line, arguments, parameters, callee, etc
5. Symbolic Pohyhedral Calculator
Simplifies invariants
Generates Polynomials
6. Polynomials Evaluator
Instrumented Sites' Polynomials

Fig. 3. Tool suite

exceptions (which did not occur in the real execution), or because the number of instances could not be expressed as a polynomial. For instance, in the `bisort` example, the reason of the over-approximation is that the actual number of instances is always bounded by $2^i - 1$ being $i = \lceil \log_2 size \rceil$. Indeed, the estimation was exact for arguments power of 2. For the (*)`health` example, it was impossible to find a non-trivial linear invariant. It actually turns out that memory consumption happens to be exponential[4] (the given result was calculated by hand). For `fft`, the argument n was required to be a power of 2 for not to throw an exception.

In [23] we show an application of this technique to estimate memory consumption. In this case, we do not only care out the number of times an allocation point is executed, but also we estimate the amount of memory requested.

[4] The JOlden programs not considered here also lead to exponential memory usage

Example:Class.Method	**#CS$_m$**	**memAlloc**	**Param.Val**	**#Objs**	**Estimation**	**Err%**
mst:MST.computeMST(g, nv)	1	$nv - 1$	10	9	9	0,00
			20	19	19	0,00
			100	99	99	0,00
			1000	999	999	0,00
bh:Tree.createTestData(nb)	23	$9nb + 14$	10	104	104	0,00
			20	194	194	0,00
			100	914	914	0,00
			1000	9014	9014	0,00
bisort (rec):Value.createTree(size,sd)	1	$size - 1$	10	7	9	22,22
			20	15	19	21,1
			200	127	199	36,2
			64	63	63	0,0
			128	127	127	0,0
			256	255	255	0,0
power (rec):Root.<init>	14	23613	-	23403	23613	0,64
em3d:Bigraph.create(nN, nD)	32	$8nN + 10$	(10, 5)	90	90	0,00
			(20, 6)	170	170	0,00
			(100, 7)	810	810	0,00
			(1000, 8)	8010	8010	0,00
(*)health (rec):Village.createVillage(l, label, back, seed)	8	$8(4^l - 1)/3$	2	40	∞	∞
			4	680	∞	∞
			6	10920	∞	∞
			8	174760	∞	∞
fft:FFT.test(n)	10	10	8	8	10	20,00
			32	8	10	20,00
			256	8	10	20,00
			1024	8	10	20,00
heapsort:JGFHeapSortBench.JGFinitialise	2	2	-	2	2	0,00
crypt:JGFCryptBench.JGFinitialise	7	7	-	7	7	0,00
series:JGFSeriesBench.JGFinitialise	1	1	-	1	1	0,00
sparsematmult:JGFSparseMatmultBench.JGFinitialise	5	5	-	5	5	0,00

Table 5
Comparison between actual executions and estimations

5 Conclusions and Future Work

In this paper we show a technique to guide `Daikon` to obtain local linear invariant. Then we show how to obtain global invariants from the locals ones.

We use those invariants to obtain symbolic expression of the number of times a set of statements are executed.

We have developed a prototype tool that allowed us to experimentally evaluate the accuracy of the method on several Java benchmarks. The results were very encouraging.

We used this tool to generate estimator on memory consumption [23] and we have obtained very promising results.

Other aspect to explore is the optimization of our method. Slicing techniques [22] could help in reducing the number of variables and statements considered when building the invariants.

References

[1] P. Boulet and X. Redon. Sppoc: fonctionnemen et applications. Research Report 00-04, LIFL, 2000.

[2] B. Cahoon and K. S. McKinley. Data flow analysis for software prefetching linked data structures in java controller. In *PACT 2001*, pages 280–291, 2001.

[3] P. Clauss. Counting solutions to linear and nonlinear constraints through ehrhart polynomials: Applications to analyze and transform scientific programs. In *ICS'96*, pages 278–285, 1996.

[4] P. Clauss. Handling memory cache policy with integer points counting. In *Euro-Par'97*, pages 285–293, 1997.

[5] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction of approximation of fixed points. In *POPL 77*, pages 238–252, 1977.

[6] P. Cousot and R. Cousot. Modular static program analysis, invited paper. In *CC 02*, pages 159–178, Grenoble, France, April 6—14 2002. LNCS 2304.

[7] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL 78*, pages 84–97, Tucson, Arizona, 1978.

[8] C. Daly, J. Horgan, J. Power, and J. Waldron. Platform independent dynamic java virtual machine analysis: the java grande forum benchmark suite. In *Java Grande*, pages 106–115, 2001.

[9] E. Ehrhart. Polynômes arithmetiques et methode des polyedres en combinatorie. *Series of Numerical Mathematics*, 35:25–49, 1977.

[10] M. D. Ernst, J. Cockrell, W. G. Griswold, and D. Notkin. Dynamically discovering likely program invariants to support program evolution. In *ICSE99*, pages 213–224, 1999.

[11] T. Fahringer. Efficient symbolic analysis for parallelizing compilers and performance estimators. *TJS*, 12(3), 1998.

[12] T. Fahringer and B. Scholz. A unified symbolic evaluation framework for parallelizing compilers. *TPDS*, 11(11), 2000.

[13] C. Flanagan and K. Rustan M. Leino. Houdini, an annotation assistant for ESC/Java. *LNCS*, 2021:500+, 2001.

[14] P. Grun, F. Balasa, and N. Dutt. Memory size estimation for multimedia applications. In *CODES/CASHE '98*, pages 145–149. IEEE, 1998.

[15] G.T. Leavens, K. Rustan M. Leino, E. Poll, C. Ruby, and B. Jacobs. JML: notations and tools supporting detailed design in Java. In *OOPSLA'00*, pages 105–106, 2000.

[16] B. Lisper. Fully automatic, parametric worst-case execution time analysis. In *WCET 03*, 2003.

[17] V. Loechner, B. Meister, and P. Clauss. Precise data locality optimization of nested loops. *TJS*, 21(1):37–76, 2002.

[18] H. B. Sipma M. A. Colon, S. Sankaranarayanan. Linear invariant generation using non-linear constraint solving. In *CAV'03*, volume 2725, 2003.

[19] J. W. Nimmer and M. D. Ernst. Static verification of dynamically detected program invariants:integrating Daikon and ESC/Java. In *RV 2001,ENTCS*.

[20] W. Pugh. Counting solutions to presburger formulas: How and why. In *PLDI 94*, pages 121–134, 1994.

[21] V. Sundaresan P. Lam E. Gagnon R. Vallée-Rai, L. Hendren and P. Co. Soot - A java optimization framework. In *CASCON'99*, pages 125–135, 1999.

[22] F. Tip. A survey of program slicing techniques. *Journal of programming languages*, 3:121–189, 1995.

[23] D. Garbervetsky V. Braberman and S. Yovine. Synthesizing parametric specifications of dynamic memory utilization in imperative object-oriented programs. *LCTES'05*, 2005.

[24] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Analytical computation of ehrhart polynomials: enabling more compiler analyses and optimizations. In *CASES '04*, 2004.

[25] Y. Zhao and S. Malik. Exact memory size estimation for array computations without loop unrolling. In *DAC '99*, pages 811–816. ACM Press, 1999.