JOURNAL OF OBJECT TECHNOLOGY
Online at www.jot.fm. Published by ETH Zurich, Chair of Software Engineering. ©JOT, 2002

Vol. 0, No. 0, Z

A Static Analysis for Synthesizing Parametric Specifications of Dynamic Memory Consumption

Víctor Braberman, School of Computer Sciences, Universidad de Buenos Aires, Buenos Aires, Argentina
Diego Garbervetsky, School of Computer Sciences, Universidad de Buenos Aires, Buenos Aires, Argentina
Sergio Yovine, Verimag, Grenoble, France

We present a static analysis for computing a parametric upper-bound of the amount of memory dynamically allocated by (Java-like) imperative object-oriented programs. We propose a general procedure for synthesizing non-linear formulas which conservatively estimate the quantity of memory explicitly allocated by a method as a function of its parameters. We have implemented the procedure and evaluated it on several benchmarks. Experimental results produced exact estimations for most test cases, and quite precise approximations for many of the others. We also apply our technique to compute usage in the context of scoped memory and discuss some open issues.

1 INTRODUCTION

The embedded and real-time software industry is leading towards the use of object-oriented programming languages such as Java. This trend brings in new research challenges.

A particular mechanism which is quite problematic in real-time embedded contexts is automatic dynamic memory management. One problem is that execution and response times are extremely difficult to predict in presence of a garbage collector. There has been significant research work to come up with a solution to this issue, either by building garbage collectors with real-time performance, e.g. [1, 24, 25, 39, 42], or by using a scope-based programming paradigm, e.g. [3, 8, 20, 21]. Another problem is that evaluating quantitative memory requirements becomes inherently hard. Indeed, finding a finite upper- bound on memory consumption is undecidable [22]. This is a major drawback since embedded systems have (in most cases) stringent memory constraints or are critical applications that cannot run out of memory.

In this paper we propose a novel technique for computing a parametric upper-bound of the amount of memory dynamically allocated by Java-like imperative object-oriented programs. As the major contribution, we present a technique to quantify the explicit dynamic allocations of a method. Given a method m with pa-

Cite this article as follows: : *A Static Analysis for Synthesizing Parametric Specifications of Dynamic Memory Consumption*, in Journal of Object Technology, vol. 0, no. 0, Z, pages 1–99,
http://www.jot.fm/issues/issues_Z_/

rameters $p_1, \ldots, p_k$ we exhibit an algorithm that computes a non-linear expression over $p_1, \ldots, p_k$ which over-approximates the amount of memory allocated during the execution of m.

Roughly speaking, our technique works as follows. For every allocation statement, we find an invariant that relates program variables in such a way that the amount of consumed memory is a function of the number of integer solutions of the invariant. This number is given in a parametric form as a polynomial where unknowns are method parameters. Our technique does not require annotating the program in any form and produces parametric non-linear upper-bounds on memory usage. The polynomials are to be evaluated on program (or method) inputs to obtain the actual bound.

To get a flavor of the approach, consider for instance the following program:

```
void m1(int k) {
  for(int i=1;i<=k;i++)  {
     A a = new A();
     m2(i);
   }
}
```

```
void m2(int n) {
  for(int j=1;j<=n;j++) {
     B b = new B();
  }
}
```

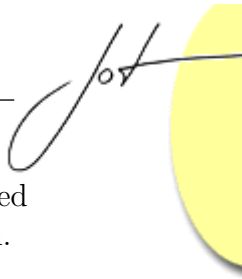
For $m2$, our technique computes the expression $size(\texttt{B}) \cdot n$ which is the amount of allocated memory if the program starts at $m2$ [1]. For $m1$, the computed expression is $size(\texttt{A}) \cdot k + size(\texttt{B}) \cdot \frac{1}{2}(k^2 + k)$ because starting at $m1$, the program will invoke $m2$ k times and, at each invocation $i \in [1, k]$, $m2(i)$ will allocate i instances of B, resulting in a total amount of $\sum_{i=1}^{k} i = \frac{1}{2}(k^2 + k)$ instances of B, which have to be added to the k instances of A directly allocated by $m1$.

Combining this algorithm with static pointer and escape analyses, we are able to compute memory region sizes to be used in scope-based memory management. Given a method m with parameters $p_1, \ldots, p_k$, we develop two algorithms that compute non-linear expressions over $p_1, \ldots, p_k$ which over-approximate, respectively, the amount of memory that *escapes from* and is *captured by* m.

These techniques can be used to predict explicit memory requirements, both during compilation and at runtime. Applications are manyfold, from improvements in memory management to the generation of parametric memory-allocation certificates. These specifications would enable application loaders and schedulers (e.g., [29]) to make decisions based on available memory resources and the memory-consumption estimates.

It should be noted that our analysis only copes with allocations explicitely made by a program through new statements in its code. The amount of "hidden" memory

[1] For simplicity, we assume here the constructor B() does not allocate memory. This issue will be handled later when we present the technique in detail.



allocated by native methods or by the virtual machine itself cannot be quantified with this technique. This is a very important issue that deserves further research.

Related Work

The problem of dynamic memory estimation has been studied for functional languages in [26, 27, 43]. The work in [26] statically infers, by typing derivation and linear programming, linear expressions that depend on function parameters. The technique is stated for functional programs running under a special memory mechanism (free list of cells and explicit deallocation in pattern matching). The computed expressions are linear constraints on the sizes of various parts of data. In [27] a variant of ML is proposed together with a type system based on the notion of sized types [28], such that well typed programs are proven to execute within the given memory bounds. The technique proposed in [43] consists in, given a function, constructing a new function that symbolically mimics the memory allocations of the former. The computed function has to be executed over a valuation of parameters to obtain a memory bound for that assignment. The evaluation of the bound function might not terminate, even if the original program does.

For imperative object-oriented languages, solutions have been proposed in [9, 10, 22]. The technique of [22] manipulates symbolic arithmetic expressions on unknowns that are not necessarily program variables, but added by the analysis to represent, for instance, loop iterations. The resulting formula has to be evaluated on an instantiation of the unknowns left to obtain the upper-bound. No benchmarking is available to assess the impact of this technique in practice. Nevertheless, two points may be made. Since the unknowns may not be program inputs, it is not clear how instances are produced. Second, it seems to be quite over-pessimistic for programs with dynamically created arrays whose size depends on loop variables. The method proposed in [9, 10] relies on a type system and type annotations, similar to [27]. It does not actually synthesize memory bounds, but statically checks whether size annotations (Presburger's formulas) are verified. It is therefore up to the programmer to state the size constraints, which are indeed linear.

Our approach combines techniques used for performance analysis [18], cache analysis [12], data locality [33], worst case execution time analysis [31], and memory optimization [23, 45]. To our knowledge, their use to automatically synthesize method-centric parametric non-linear over-approximations of memory consumption is novel.

Document Structure

In Section 2 we introduce useful definitions, notations, and some already developed techniques we rely on. In Section 3, we explain our general method for calculating memory consumption. In Section 4 we show our method for region-size estimation

in scope-based memory management. In section 5 we show the results of applying our technique to some well known benchmarks. Section 6 discusses some extensions and future work. Section 7 presents some conclusions.

2 PRELIMINARIES

Counting the number of solutions of a constraint

Let $\mathcal{I}$ be an arithmetic constraint over a set of integer variables $V = W \uplus P$ where P represents a set of distinguished variables (called parameters) and W is the remaining set of variables. We write $\mathbf{v}$, $\mathbf{p}$ and $\mathbf{w}$ to denote assignments of values to variables. $\mathcal{I}(\mathbf{v})$ is the result of evaluating $\mathcal{I}$ in $\mathbf{v}$.

$\mathcal{C}(\mathcal{I}, P)$ denotes the symbolic expression over P which provides the *number of integer solutions* of $\mathcal{I}$ for the set of variables W, assuming P has fixed values. More precisely:

$$\mathcal{C}(\mathcal{I}, P) \quad = \quad \lambda \mathbf{p}.\ \#\{\ \mathbf{w} \in \mathbb{Z}^{|W|} \mid \mathcal{I}(\mathbf{w}, \mathbf{p})\ \}$$

There are several techniques which can be used to obtain these symbolic expressions, e.g., [11, 18, 37, 44]. Here, we will briefly present the one described in [11, 44] which applies to linear constraints.

A *linear parametric set* S_P is defined as $S_P = \{\ \mathbf{w} \in \mathbb{Q}^{|W|} \ \mid\ A\mathbf{w} \geq B\mathbf{p} + \mathbf{c}\ \}$ where A and B are integer matrices, and $\mathbf{c}$ is an integer vector. S_P is called a *parametric polytope* whenever the number of points in S_P is finite for each $\mathbf{p}$.

A $|P|$-*periodic* number is a function $U : \mathbb{Z}^{|P|} \to \mathbb{Z}$ for which there exists $\mathbf{r} \in \mathbb{N}^{|P|}$ such that $U(\mathbf{p}) = U(\mathbf{p}')$ whenever $p_i \equiv p'_i \mod r_i$, for $1 \leq i \leq |P|$. The least common multiple of all r_i is called the *period* of U.

A *quasi-polynomial* in $|P|$ variables is a $|P|$-dimensional polynomial in variables over $|P|$-periodic numbers. That is, the coefficients of a quasi-polynomial depend periodically on the variables.

Ehrhart [16] showed that $\mathcal{C}(S_P, P)$ for a parametric polytope S_P, can be represented as a *quasi-polynomial*, provided S_P can be represented as a *convex* combination of its parametric *vertices*, where each vertex is an affine combination of the parameters with *rational* coefficients. This result can be extended to unions of parametric polytopes defined as $\{\ \mathbf{w} \in \mathbb{Q}^{|W|} \ \mid\ A\mathbf{w} \geq B\mathbf{p} + \mathbf{c}\ ,\ M\mathbf{w} \mod \mathbf{d} \geq \mathbf{e}\ \}$, where M is an integer matrix, and $\mathbf{d}, \mathbf{e}$ are integer vectors.

Example Consider, for instance, the linear parametric set $S^1 = \{\mathbf{w} \mid \mathcal{I}^1(\mathbf{w}, \mathbf{p})\}$, where $\mathcal{I}^1$ is defined as follows:

$$\mathcal{I}^1 \quad = \quad \{k = mc,\ 1 \leq i \leq k,\ 1 \leq j \leq i,\ n = i\}$$

where $W = \{k,\ i,\ j,\ n\}$, and $P = \{mc\}$. The corresponding Ehrhart polynomial is:

$$\mathcal{C}(S^1, mc) \quad = \quad \frac{1}{2}mc^2 + \frac{1}{2}mc$$

For the linear parametric set $S^2 = \{\mathbf{w} \mid \mathcal{I}^2(\mathbf{w}, \mathbf{p})\}$, with

$$\mathcal{I}^2 \quad = \quad \{k = mc,\ 1 \leq i \leq k,\ 1 \leq j \leq i,\ n = i,\ j \mod 3 = 0\}$$

the Ehrhart polynomial is:

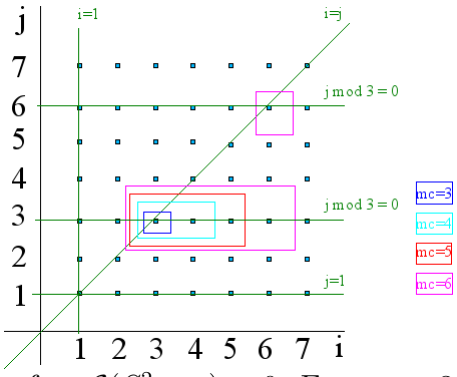
$$\mathcal{C}(S^2, mc) \quad = \quad \frac{1}{6}mc^2 - \frac{1}{6}mc + \left[0, 0, -\frac{1}{3}\right]_{mc}$$

where the period is 3 and the last coefficient of the polynomial depends periodically on mc as follows:

$$\left[0, 0, -\frac{1}{3}\right]_{mc} = \begin{cases} -\frac{1}{3} & \text{when } mc \mod 3 = 2 \\ 0 & \text{otherwise} \end{cases}$$

The following illustration depicts the result of evaluating $\mathcal{C}(S^2, mc)$ in the interval $[1, 6]$.

mc	$\mathcal{C}(S^2, mc)$
1	0
2	0
3	1
4	2
5	3
6	5



When $mc \in \{1, 2\}$, the are no solutions, therefore $\mathcal{C}(S^2, mc) = 0$. For $mc = 3$, there is only one solution given by $k = mc = j = i = n = 3$ (blue box), and $\mathcal{C}(S^2, mc) = 1$. For $mc = 4$, there are two solutions ($\mathcal{C}(S^2, mc) = 2$), given by $k = mc = 4$, $j = 3$, and $i = n \in \{3, 4\}$ (cyan box). For $mc = 5$, the number of solutions is three ($\mathcal{C}(S^2, mc) = 3$): $k = mc = 5$, $j = 3$ and $i = n \in \{3, 4, 5\}$ (red box). For $mc = 6$, the solution space is non-convex and contains $\mathcal{C}(S^2, mc) = 5$ points (magenta box). □

Several algorithms have been proposed for computing Ehrhart polynomials. The first one is discussed in [11]. This algorithm is not complete and has exponential-time complexity, even when the number of variables in the inequalities is fixed. This happens because the periods are only bounded by the values of the coefficients in the linear inequalities of the input. A more efficient algorithm proven to have polynomial-time complexity for fixed dimensions has been developed in [44]. Still,

the output polynomials can be relatively large in some *degenerate* cases. Recently, a fast algorithm for computing Ehrhart polynomials that over-approximate $\mathcal{C}(S_P, P)$ has been proposed in [34]. All these algorithms are implemented in the Polyhedral Library `PolyLib` [32] used in this article. Computing Ehrhart polynomials is quite involved as it resorts to very technical results in discrete mathematics which are out of the scope of this paper. The interested reader is referred to [11, 34, 44] for a detailed explanation.

Notation for Programs

We define a program as a set $\{m_0, m_1, \ldots\}$ of methods. A method has a list P_m of parameters ($\mathbf{p_m}$ will denote the method arguments when m is called by another method m') and a sequence of statements.

Programs are sequential and non-recursive. We assume that there is no variable name clashing including formal parameters, local and global variable names. For the sake of the presentation, we assume that method parameters are of integer type. This restriction is, however, not essential as later discussed in Section 6.

Example In Figure 1 we present the program we will use throughout the paper to illustrate our approach. The program creates two arrays: a (bi-dimensional) and e, whose cells can contain an Integer (`new Integer`) or an array of Integers (`newA Integer`) depending on an expression evaluated over a loop variable. □

```
    void m0(int mc) {
1:    Ref0 h = new Ref0();
2:    Object[] a = m1(mc);
3:    Object[] e = m2(2*mc,h);
    }
    Object[] m1(int k) {
1:    int i;
2:    Ref0 l = new Ref0();
3:    Object[] b = newA Object[k];
4:    for(i=1;i<=k;i++)  {
5:        b[i-1] = m2(i,l);
      }
6:    Object[] c = newA Integer[9];
7:    return b;
    }
class Ref0 {
   public Object ref;
}
```

```
    Object[] m2(int n, Ref0 s) {
1:    int j;
2:    Object c,d,e;
3:    Object[] f = newA Object[n]

4:    for(j=1;j<=n;j++) {
5:      if(j % 3 == 0) {
6:        c = newA Integer[j*2+1];
        }
        else {
7:        c = new Integer(0);
        }
8:      d = newA Integer[4];
9:      f[j-1] = c;
      }
10:   e = newA Integer[1];
11:   s.ref = e;
12:   return f;
    }
```

Figure 1: Motivating example

Each statement in a program is identified with a *control location* $\ell = (m, n) \in$

$Label =_{def} Method \times \mathbb{N}$ (a method and a position inside the method) which uniquely characterizes the statement via the *stm* mapping ($stm : Label \to Statement$). We write $\mathsf{mth}(\ell)$ to denote m.

The *call graph* $G \subseteq Method \times Label \times Method$ of a program is such that $(m, \ell, m') \in G$ whenever $\ell = (m, n)$ and $stm(\ell)$ is a method call to m'. A (finite) *path* π in G is a sequence $m_1.\ell_1.\ldots.m_k.\ell_k.m_{k+1}$, $k \geq 1$, such that $(m_i, \ell_i, m_{i+1}) \in G$. $|\pi| = k$ is the *length* of π. For $j \in [1, |\pi|]$, we define $\pi_{\ldots j}$ to be the sub-sequence $m_1.\ell_1.\ldots.m_j.\ell_j$ of π, and we write $\mathsf{ploc}(\pi, j)$ to denote the control location ℓ_j. For $j \in [1, |\pi| + 1]$, $\mathsf{pmth}(\pi, j)$ denotes the method m_j.

Example The call graph of our example is $\{(m0, 2, m1), (m0, 3, m2), (m1, 5, m2)\}$ (see Fig. 2). $m0.2.m1.5.m2$ is a path. For simplicity, in the examples we will only use the position of the control location rather than the label. □

Representing a program state

For the sake of simplicity, we would not formally define program semantics. Such a formalization is given in, for instance, [40]. Informally, a state σ of a program in run-time is given by the values of the variables, the heap, the control location and the call stack. A program run is a sequence $\sigma_1 \ldots$ of states. Notice that, the absence of recursion and name clashing implies that mapping variable names to values is enough to model program data (i.e., no environment or data stacks are required).

A static analysis for safely estimating memory consumption requires defining an abstraction that conservatively describes program states and runs in a suitable way. In our case, this abstraction only needs to keep enough information about the program state to be able to *count* the number of times object creation statements are executed in a program run. For simplicity, we assume that counting only depends on non heap-allocated[2], integer-valued variables. Therefore, it is important to notice that the heap in a program state can be abstracted away. This is due to the fact that the points-to relationship between objects in the heap *is not relevant* for computing the amount of explicitly allocated memory, which is, indeed, equal to the size of the portion of the heap *directly* created by `new` statements in the program code.

For the purpose of the analysis, the program control state can be characterized by the control location and the call stack. A *control state* ζ is the sequence $\pi.\ell$, where $\ell \in Label$ is a location and π is a path to method $\mathsf{mth}(\ell)$ in the call graph G.

Example $m0.2.m1.5.m2.3$ is a control state. □

Let $\zeta = \pi.\ell$ be a control state and $\sigma_1 \ldots \sigma_t$ be a finite run such that the location of state σ_t is ℓ, and the call stack of σ_t is π. Then, there exists a set of indexes

[2]We will discuss about relaxing this assumption in Section 6.

$\{i_1, \ldots, i_{|\pi|}\}$, such that the control state ζ_j of σ_{i_j} is $\pi_{\ldots j}$, $j \in [1, |\,\pi\,|]$. That is, $\mathsf{pmth}(\pi, j)$ is the method on the top of the stack in state σ_{i_j}, $\mathsf{ploc}(\pi, j)$ is the control location corresponding to the method call to $\mathsf{pmth}(\pi, j+1)$, and $\mathsf{pmth}(\pi, |\pi| +1)$ is $\mathsf{mth}(\ell)$. We say that run $\sigma_1 \ldots \sigma_t$ *reaches* the control state ζ.

Example Let ζ be the control state $m0.2.m1.5.m2.3$. Consider the run $\sigma_1 \ldots \sigma_{10}$ defined as: $(m0.1, \theta_1)$ $(m0.2, \theta_2)$ $(m1.1, \theta_3)$ $(m1.2, \theta_4)$ $(m1.3, \theta_5)$ $(m1.4, \theta_6)$ $(m1.5, \theta_7)$ $(m2.1, \theta_8)$ $(m2.2, \theta_9)$ $(m2.3, \theta_{10})$, where θ_i, $1 \leq i \leq 10$, record the valuations of program variables, the heap, and the call stack. We have that the $\sigma_1 \ldots \sigma_{10}$ reaches ζ, the call stack of σ_{10} is the path $\pi = m0.2.m1.5.m2$, and the set of indexes $\{2, 7\}$ is such that $\zeta_1 = \pi_{\ldots 1} = m0.2$ is the control state of $\sigma_{i_1} = \sigma_2$, and $\zeta_2 = \pi_{\ldots 2} = m0.2.m1.5$ is the control state of $\sigma_{i_2} = \sigma_7$. These indexes correspond to the times in the run where a method yet in the call stack of state σ_{10} (i.e., $m1$ at 2 and $m2$ at 7), or equivalently, in π, has been pushed (i.e., called). □

An *invariant* for a control state ζ is an assertion over program variables (local, global and method parameters) that holds whenever such a control state is reached in any run.

Given a method m and a control state $\zeta = \pi.\ell$ such that $\mathsf{pmth}(\pi, 1) = m$, that is, π is a path in the call graph G that starts in m, $\mathcal{I}^m_\zeta$ denotes an invariant predicate for ζ. We call the pair $(\zeta, \mathcal{I}^m_\zeta)$ an *abstract state* as it is a *conservative* approximation of the possible program states at location ℓ and stack π in any run starting at method m. That is, for every run $\sigma_1 \ldots \sigma_t$ starting at $(m, 1)$, that reaches ζ, $\mathcal{I}^m_\zeta(\sigma_t)$ holds.

Example Let $\zeta = m0.2.m1.5.m2.8$. The constraint $\mathcal{I}^{m_0}_\zeta$ defined by set of linear inequalities $\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$ is an invariant for ζ. □

Whenever $(m, \ell, m') \in G$ (i.e., $stm(\ell)$ is a method call), we assume the invariant $\mathcal{I}^m_\zeta$, for any $\zeta = \pi.\ell$, constrains not only the values of variables local to the caller m, but also equates actual parameters (local variables of the caller m) with formal parameters (local variables of the callee m'). This assumption simplifies the presentation.

Let m, m' be two methods such that $(m, \ell, m') \in G$, $\zeta = m_1 \ldots m.\ell$ and $\zeta' = m' \ldots m_s.\ell_s$ be two control states, and $\mathcal{I}^m_\zeta$ and $\mathcal{I}^{m'}_{\zeta'}$ be two invariants. We have that $\zeta.\zeta'$ is a control state and $\mathcal{I}^m_{\zeta.\zeta'}$ defined as $\mathcal{I}^m_\zeta \wedge \mathcal{I}^{m'}_{\zeta'}$ is an invariant for $\zeta.\zeta'$. In words, the invariant of a control state obtained by concatenating two control states is the conjunction of the respective invariants.

Example Let $\zeta = m0.2$ and $\zeta' = m1.5.m2.8$. We have that

$$\mathcal{I}^{m0}_{m0.2} = \{k = mc\}, \quad \mathcal{I}^{m1}_{m1.5} = \{1 \leq i \leq k, n = i\}, \text{ and } \quad \mathcal{I}^{m2}_{m2.8} = \{1 \leq j \leq n\}$$

are invariants, which gives that

$$\begin{aligned}
\mathcal{I}^{m0}_{m0.2.m1.5} &= \{k = mc, 1 \leq i \leq k, n = i\} \\
\mathcal{I}^{m1}_{m1.5.m2.8} &= \{1 \leq i \leq k, n = i, 1 \leq j \leq n\} \\
\mathcal{I}^{m0}_{m0.2.m1.5.m2.8} &= \{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}
\end{aligned}$$

are also invariants. □

Given a control state $\zeta = m_1.\ell_1 \ldots m_k.\ell_k$, the property above provides means for computing the invariant $\mathcal{I}^{m_1}_{\zeta}$ as the conjunction $\bigwedge_{i=1}^{k} \mathcal{I}^{m_i}_{m_i.\ell_i}$. Each $\mathcal{I}^{m_i}_{m_i.\ell_i}$ is called a *local* invariant.

Example Table 1 shows invariants that define iteration spaces and corresponding Ehrhart polynomials for some control states starting at method $m0$.□

ζ	$\mathcal{I}^{\mathbf{m0}}_{\zeta}$	$\mathcal{C}(\mathcal{I}^{\mathbf{m0}}_{\zeta}, \mathbf{P_{m0}})$
$m0.2.\mathbf{m1.2}$	$\{k = mc\}$	1
$m0.2.m1.5.\mathbf{m2.3}$	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0.2.m1.5.\mathbf{m2.6}$	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{1}{6}mc^2 - \frac{1}{6}mc + [0, 0, -\frac{1}{3}]_{mc}$
$m0.2.m1.5.\mathbf{m2.7}$	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n, j \bmod 3 > 0\}$	$\frac{1}{3}mc^2 + \frac{2}{3}mc + [0, 0, \frac{1}{3}]_{mc}$
$m0.2.m1.5.\mathbf{m2.8}$	$\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$	$\frac{1}{2}mc^2 + \frac{1}{2}mc$
$m0.2.m1.5.\mathbf{m2.10}$	$\{k = mc, 1 \leq i \leq k, n = i\}$	mc
$m0.3.\mathbf{m2.3}$	$\{n = 2mc\}$	1
$m0.3.\mathbf{m2.6}$	$\{n = 2mc, 1 \leq j \leq n, j \bmod 3 = 0\}$	$\frac{2}{3}mc + [0, -\frac{2}{3}, -\frac{1}{3}]_{mc}$
$m0.3.\mathbf{m2.7}$	$\{n = 2mc, 1 \leq j \leq n, j \ mod \ 3 > 0\}$	$\frac{4}{3}mc + [0, \frac{2}{3}, \frac{1}{3}]_{mc}$
$m0.3.\mathbf{m2.8}$	$\{n = 2mc, 1 \leq j \leq n\}$	$2mc$
$m0.3.\mathbf{m2.10}$	$\{n = 2mc\}$	1

Table 1: Some invariants and Ehrhart polynomials for $m0$

Counting the number of visits of a control state

Let $(\zeta, \mathcal{I}^m_{\zeta})$ be an abstract state such that the invariant $\mathcal{I}^m_{\zeta}$ defines a *polyhedral iteration space* [11], that is, a polytope that characterizes all possible values of loop-control variables and parameters involved in a program interation that passes through ζ.

Example Let ζ be the control state $m0.2.m1.5.m2.8$. The invariant $\mathcal{I}_{\zeta}^{m_0}$ defined by set of linear inequalities $\{k = mc, 1 \leq i \leq k, n = i, 1 \leq j \leq n\}$ defines a polyhedral iteration space for ζ. □

Therefore, given an invariant $\mathcal{I}_{\zeta}^{m}$ that defines a polyhedral iteration space, it follows that counting the number of integer solutions of $\mathcal{I}_{\zeta}^{m}$ yields an expression that over-approximates *the number of times* a concrete state, whose abstraction is $(\zeta, \mathcal{I}_{\zeta}^{m})$, is reached in a run starting at m.

Example Let $\zeta = m0.2.m1.5.m2.8$. We have that ζ is reached at most $\frac{1}{2}mc^2 + \frac{1}{2}mc$ times in a run starting at m_0 for any value of parameter mc.□

3 SYNTHESIZING MEMORY CONSUMPTION

In this section we present our technique for synthesizing non-linear formulas (actually, quasi-polynomials) to conservatively over-estimate memory consumption in terms of method parameters. First, we show how to adapt the counting technique discussed in Section 2 to cope with memory allocations. Second, we show how to compute the total amount of memory allocated by a method.

Memory allocated by a creation site

We now focus on statements that create new objects (i.e., allocate memory): `new` and `newA` statements. We assume that those statements only create object instances and constructors are called separately and handled as any other method call. We call *creation site*, and denote cs, a control state associated to such operations: $cs \in CS = \{\ \pi.\ell \in Label^{+} \mid stm(\ell) \in \{\texttt{new T}, \texttt{newA T}[\cdot]\ldots[\cdot]\}\ \}$.

To compute the amount of memory allocated by a creation site cs we define the function $\mathcal{S}$ (see below). Given an invariant $\mathcal{I}_{cs}^{m}$ for cs and method m with parameters P_m, $\mathcal{S}$ computes the parametric number of visits to cs and multiplies the resulting expression for the size of the allocated object. This parametric expression over-estimates the memory allocate by cs whenever cs is a `new` statement. Nevertheless, when cs is an array allocation (i.e., `newA T`$[e_1]\ldots[e_n]$), this technique needs to be slightly adapted considering the fact that an array is a collection of elements of the same type. In fact, the `newA T`$[e_1]\ldots[e_n]$ statement creates the same number of instances (and, therefore, allocates the same amount of memory) as n nested loops of the form:

```
for( h_1 = 1; h_1 ≤ e_1; h_1++ )
  ...
    for( h_n = 1; h_n ≤ e_n; h_n++ )
      newA T[1]
```

whose iteration space can be described by the invariant $\bigcup_{i=1..n}\{1 \leq h_i \leq e_i\}$.

Thus, we define the function $\mathcal{S}$ as follows:

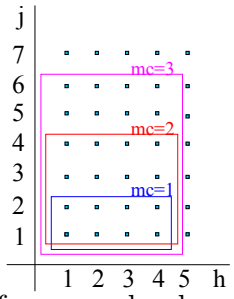
```
S(I^m_cs, P_m, cs) // returns an Expression over P_m
ℓ = last(cs); // (cs = π.ℓ)
if stm(ℓ)= new T
   res:=size(T) · C(I^m_cs, P_m);
else if stm(ℓ)= newA T[e_1]...[e_n]
   Inv_array:= I^m_cs ∪ ⋃_{i=1..n}{1 ≤ h_i ≤ e_i}
   res:=size(T[]) · C(Inv_array, P_m);
end if;
return res;
```

where $size(\texttt{T})$ is a symbolic expression that denotes the size of an object of type `T`, and $size(\texttt{T[]})$ is a symbolic expression that denotes the size of a *cell* of an array of type `T` [3]. $\mathcal{C}$ is the symbolic expression that counts the number of integer solutions for an invariant as defined in Section 2.

As linear invariants are conservative, $\mathcal{S}(\mathcal{I}^m_{cs}, P_m, cs)$ *over-approximates*, in general, the amount of memory allocated by cs in any run starting at m. That is, for any run $\sigma_1 \ldots \sigma_t$ that starts at m and reaches cs, the amount of memory in the heap of σ_t occupied by objects *allocated by* creation site cs is bounded by the result of evaluating $\mathcal{S}(\mathcal{I}^m_{cs}, P_m, cs)$ in the values of parameters P_m in σ_1.

Example Consider the creation site $m0.3.m2.8$, which corresponds to statement `d = newA Integer[4]` in line 8 of method $m2$ when called from $m0$ at line 3.

$$
\begin{aligned}
&\mathcal{S}(\mathcal{I}^{m0}_{m0.3.m2.8}, mc, m0.3.m2.8) = \\
&\quad = \; size(\texttt{Integer[]}) \cdot \mathcal{C}(\mathcal{I}^{m0}_{m0.3.m2.8} \cup \{1 \leq h \leq 4\}, mc) \\
&\quad = \; size(\texttt{Integer[]}) \cdot \mathcal{C}(\{n = 2mc, 1 \leq j \leq n, 1 \leq h \leq 4\}, mc) \\
&\quad = \; size(\texttt{Integer[]}) \cdot \mathcal{C}(\{1 \leq j \leq 2mc, 1 \leq h \leq 4\}, mc) \\
&\quad = \; size(\texttt{Integer[]}) \cdot 8mc
\end{aligned}
$$



The figure on the right depicts the sets of points in the invariant for several values of parameter mc.□

Example Table 2 shows the polynomials that over-approximate the amount of memory allocated for (some selected) creation sites reachable from method $m0$.□

[3] $size(\texttt{T[]})$ will be the same for all `Object` subclasses and will differ for arrays of basic types.

$\mathbf{cs}$	$\mathcal{S}(\mathcal{I}^{\mathbf{m0}}_{\mathbf{cs}}, \mathbf{P_{m0}}, \mathbf{cs})$
$m0.2.m1.2$	$size(\texttt{Ref0})$
$m0.2.m1.6$	$size(\texttt{Integer[]}) \cdot 9$
$m0.2.m1.5.m2.3$	$size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{1}{2}mc\right)$
$m0.2.m1.5.m2.6$	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{1}{2}mc^2 + [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}]_{mc} \cdot mc + [0, -\frac{4}{9}, -\frac{11}{9}]_{mc}\right)$
$m0.2.m1.5.m2.7$	$size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + \frac{2}{3}mc + [0, 0, \frac{1}{3}]_{mc}\right)$
$m0.2.m1.5.m2.8$	$size(\texttt{Integer[]}) \cdot (2mc^2 + 2mc)$
$m0.3.m2.3$	$size(\texttt{Object[]}) \cdot 2mc$
$m0.3.m2.6$	$size(\texttt{Integer[]}) \cdot \left(\frac{4}{3}mc^2 + [2, -\frac{2}{3}, \frac{2}{3}]_{mc} \cdot mc + [0, -\frac{2}{3}, -\frac{2}{3}]_{mc}\right)$
$m0.3.m2.7$	$size(\texttt{Integer}) \cdot \left(\frac{4}{3}mc + [0, \frac{2}{3}, \frac{1}{3}]_{mc}\right)$
$m0.3.m2.8$	$size(\texttt{Integer[]}) \cdot 8mc$

Table 2: Polynomials of memory allocation.

Memory allocated by a method

Having shown how to compute the amount of memory allocated by a single creation site, we determine how much memory is allocated by a run starting at method m. Basically, our technique identifies the creation sites reachable from method m, gets the corresponding invariants, computes the amount of memory allocated by each one and finally yields the sum of them.

Let $CS_m \subseteq CS$ denote the set of creation sites reachable from method m that is, the set of creation sites $cs = \pi.\ell \in CS$, where π is a path starting at m.

Example The creation sites of the example in Fig. 1 are:

$CS_{m0} = \{$ $m0.1$, $m0.2.m1.2$, $m0.2.m1.3$, $m0.2.m1.6$, $m0.2.m1.5.m2.3$, $m0.2.m1.5.m2.6$, $m0.2.m1.5.m2.7$, $m0.2.m1.5.m2.8$, $m0.2.m1.5.m2.10$, $m0.3.m2.3$, $m0.3.m2.6$, $m0.3.m2.7$, $m0.3.m2.8$, $m0.3.m2.10$ $\}$

$CS_{m1} = \{$ $m1.2$, $m1.3$, $m1.6$, $m1.5.m2.3$, $m1.5.m2.6$, $m1.5.m2.7$, $m1.5.m2.8$, $m1.5.m2.10$ $\}$

$CS_{m2} = \{$ $m2.3$, $m2.6$, $m2.7$, $m2.8$, $m2.10$ $\}$

Fig. 2 shows the call graph augmented with creation sites. This graph is automatically constructed with the tool described in [4].□

Observe that, since we are not dealing with recursive programs, the number of paths in the call graph and thus the number of control states is finite. Now, the problem of computing a parametric upper-bound of the amount of memory allocated by a method m can be reduced to: for each $cs \in CS_m$, obtain an invariant, compute the function $\mathcal{S}$ and sum up the results.

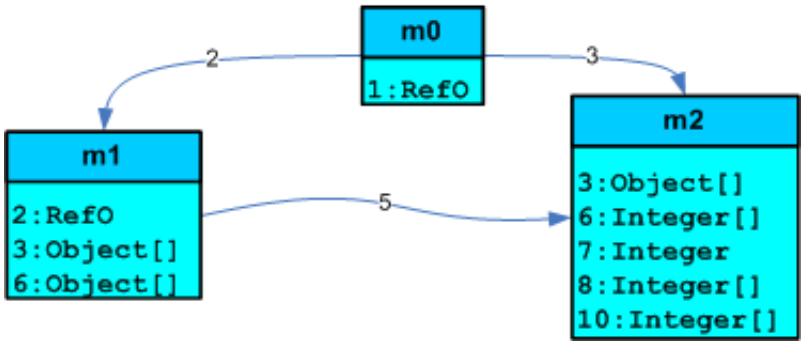


Figure 2: Call Graph and Creation Sites

The function computeAlloc computes an expression (in terms of method parameters) that over-approximates the amount of memory allocated by a selected set of creations sites:

$$\texttt{computeAlloc}(m,\ CS) = \sum_{cs\ \in\ CS} \mathcal{S}(\mathcal{I}^m_{cs}, P_m, cs)\ \text{, where } CS \subseteq CS_m$$

Given a method m, the symbolic estimator of the memory dynamically allocated by m is defined as follows:

$$\texttt{memAlloc}(m) = \texttt{computeAlloc}(m,\ CS_m)$$

That is, for any run $\sigma_1 \ldots$ that starts at m, the amount of memory, in the heap of any state in the run, occupied by objects *allocated by* a creation site in CS_m reached by the run, is bounded by the result of evaluating memAlloc(m) in the values of parameters P_m in σ_1.

Notice that the over-estimation may arise because invariants are conservative, but also as a consequence of summing up *all* creation sites reachable in the call graph, which may not all be executed by a given run.

Example Table 3 shows the expressions computed for $m0$, $m1$ and $m2$.□

The complexity of the method depends on the number of configurations of the call stack from the analyzed method to each creation site. Though this number is in the worst case exponential in the number of methods, in many cases, the topology of the call graph leads to few paths and thus the presented technique is still feasible. This actually happens for the benchmarks analyzed in Section 5. Further discussion on this topic can be found in Section 6.

Notice that, using the technique we are able to evaluate the consumption of a program starting at *any* method m. For instance, in case of a batch program it

memAlloc($m0$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{23}{6}mc^2 + ([\frac{29}{2}, \frac{71}{6}, \frac{25}{2}]_{mc}) \cdot mc + [11, \frac{83}{9}, \frac{79}{9}]_{mc}\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}m^2 + 2mc + [0, \frac{2}{3}, \frac{2}{3}]_{mc}\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) + 2 \cdot size(\texttt{Ref0})$
memAlloc($m1$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{5}{2}k^2 + [\frac{23}{6}, \frac{23}{6}, \frac{19}{6}]_k \cdot k + [9, \frac{77}{9}, \frac{70}{9}]_k\right) + size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}k + [0, 0, \frac{1}{3}]_k\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{3}{2}k\right) + size(\texttt{Ref0})$
memAlloc($m2$)	$size(\texttt{Integer[]}) \cdot \left(\frac{1}{3}n^2 + [\frac{16}{3}, \frac{14}{3}, 4]_n \cdot n + [2, 1, \frac{2}{3}]_n\right) + size(\texttt{Integer}) \cdot \left(\frac{2}{3}n + [0, \frac{1}{3}, \frac{2}{3}]_n\right) + size(\texttt{Object[]}) \cdot n$

Table 3: Memory allocated by methods $m0$, $m1$, and $m2$

would be reasonable to compute the consumption from the actual main method of the program since the consumption usually depends on command line arguments or contextual objects like the size of a referenced file. Nevertheless, the ability to compute consumption for any given method is useful to get different context-independent consumption specifications at a finer level of granularity. Besides, in cases where the application model is reactive event-driven, the consumption should be measured from a dispatched method according to the parameter values conveyed in the event.

4 APPLICATIONS TO SCOPED-MEMORY

Scoped-memory management is based on the idea of grouping sets of objects into regions associated with the lifetime of a computation unit. Thus, objects are collected together when their corresponding computation unit finishes its execution. In order to infer scope information we use pointer and escape analysis (e.g., [2, 41]). In particular, we assume that, at method invocation, a new region is created which will contain all objects *captured* by this method. When it finishes, the region is collected with all its objects. An implementation of scoped memory following this approach can be found in [20].

An object *escapes* a method when its lifetime is longer than the method's lifetime, and it cannot be safely collected when this unit finishes its execution. Let $escape : Method \to \mathbb{P}(CreationSite)$ be a function that given a method m returns (an over-approximation of) the set of creation sites $escape(m) \subseteq CS_m$ that escape m.

An object is *captured* by the method m when it can be safely collected at the end of the execution of m. Let $capture : Method \to \mathbb{P}(CreationSite)$ be a function that given a method m returns (an under-approximation of) the creation sites $capture(m) \in CS_m$ that are captured by m.

These functions can be computed using any escape analysis technique.

Example For instance, for our example in Figure 1 we have:

$$
\begin{aligned}
escape(m0) &= \{\} \\
escape(m1) &= \{m1.3, m1.5.m2.3, m1.5.m2.6, m1.5.m2.7\} \\
escape(m2) &= \{m2.3, m2.6, m2.7, m2.10\} \\
capture(m0) &= \{m0.1, m0.2.m1.3, m0.2.m1.5.m2.3, m0.2.m1.5.m2.6, m0.2.m1.5.m2.7, \\
&\quad m0.2.m1.5.m2.10, m0.3.m2.3, m0.3.m2.6, m0.3.m2.7, m0.3.m2.10\} \\
capture(m1) &= \{m1.5.m2.10, m1.2, m1.6\} \\
capture(m2) &= \{m2.8\} \ \square
\end{aligned}
$$

Memory that escapes a method

In order to symbolically characterize the amount of memory that *escapes* a method, we use the algorithm developed in Section 3, but we restrict the search to creation sites that escape the method:

$$\texttt{memEscapes}(m) = \texttt{computeAlloc}(m, escape(m))$$

This information can be used to know how much memory the method leaves allocated in the active regions (the caller region or their parent regions in the call stack) after its own region is deallocated, or to measure the amount of memory that cannot be collected by a garbage collector after the method terminates.

Example In Table 4 we show the memory-consumption expressions for the creation sites escaping $m1$. Observe that expressions are defined only on the method parameters.$\square$

memEscapes($m1$)=	$size(\texttt{Object[]}) \cdot k$	$m1.3$
	$+size(\texttt{Object[]}) \cdot \left(\frac{1}{2}k^2 + \frac{1}{2}k\right)$	$m1.5.m2.3$
	$+size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}k^3 + \frac{1}{2}k^2 + [\frac{5}{6}, \frac{5}{6}, \frac{1}{6}]_k \cdot k + [0, -\frac{4}{9}, -\frac{11}{9}]_k\right)$	$m1.5.m2.6$
	$+size(\texttt{Integer}) \cdot \left(\frac{1}{3}k^2 + \frac{2}{3}k + [0, 0, \frac{1}{3}]_k\right)$	$m1.5.m2.7$

Table 4: Amount of memory escaping from $m1$.

Memory captured by a method

To compute the expression over-estimating the amount of allocated memory that is *captured* by a method, we use the algorithm developed in Section 3, but we restrict the search to creation sites that are captured by the method:

$$\texttt{memCaptured}(m) = \texttt{computeAlloc}(m, capture(m))$$

Example Table 5 shows the expression that over-approximates the amount of memory captured by each method for our example.□

$\text{memCaptured}(m0) = size(\texttt{Ref0})$	$m0.1$
$+size(\texttt{Object[]}) \cdot mc$	$m0.2.m1.3$
$+size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{1}{2}mc\right) +$	$m0.2.m1.5.m2.3$
$+size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{1}{2}mc^2 + [-\frac{1}{6}, -\frac{1}{6}, -\frac{5}{6}]_{mc} \cdot mc + [0, -\frac{4}{9}, -\frac{11}{9}]_{mc}\right)$	$m0.2.m1.5.m2.6$
$+size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + \frac{2}{3}mc + [0, 0, \frac{1}{3}]_{mc}\right)$	$m0.2.m1.5.m2.7$
$+size(\texttt{Integer[]}) \cdot mc$	$m0.2.m1.5.m2.10$
$+ size(\texttt{Object[]}) \cdot 2mc$	$m0.3.m2.3$
$+size(\texttt{Integer[]}) \cdot \left(\frac{4}{3}mc^2 + [2, -\frac{2}{3}, \frac{2}{3}]_{mc} \cdot mc + [0, -\frac{2}{3}, -\frac{2}{3}]_{mc}\right)$	$m0.3.m2.6$
$+size(\texttt{Integer}) \cdot \left(\frac{4}{3}mc + [0, \frac{2}{3}, \frac{1}{3}]_{mc}\right)$	$m0.3.m2.7$
$+size(\texttt{Integer[]})$	$m0.3.m2.10$
$= size(\texttt{Integer[]}) \cdot \left(\frac{1}{9}mc^3 + \frac{11}{6}mc^2 + ([\frac{9}{2}, \frac{11}{6}, \frac{5}{2}]_{mc}) \cdot mc + [2, \frac{2}{9}, -\frac{2}{9}]_{mc}\right) +$ $size(\texttt{Integer}) \cdot \left(\frac{1}{3}mc^2 + 2mc + [0, \frac{2}{3}, \frac{2}{3}]_{mc}\right) + size(\texttt{Object[]}) \cdot \left(\frac{1}{2}mc^2 + \frac{7}{2}mc\right) +$ $size(\texttt{Ref0})$	Total
$\text{memCaptured}(m1) = size(\texttt{Ref0})$	$m1.2$
$+size(\texttt{Integer[]}) \cdot 9$	$m1.6$
$+size(\texttt{Integer[]}) \cdot k$	$m1.5.m2.10$
$\text{memCaptured}(m2) = size(\texttt{Integer[]}) \cdot 4n$	$m2.8$

Table 5: Memory captured by methods $m0$, $m1$ and $m2$

Assuming the resulting expression is a symbolic estimator of the size of the memory region associated to the method's scope, this information can be used to specify the size of the memory region to be allocated at run-time, as required by the RTSJ [3]. Moreover, it can be used to improve memory management algorithms.

5 METHOD VALIDATION

We have developed a proof-of-concept tool-suite to perform the initial experiments aiming at validating our approach for Java applications. This section identifies the key conceptual components of the technique, their associated challenges and briefly describes the implemented solution that was suitable to treat some well-known benchmarks.

Tool

The proof-of-concept architecture is shown in Fig.3. The tool can effectively analyze single-threaded Java programs provided they do not feature recursion or complex data structures.

Call graphs are obtained with `Soot` [38]. Invariants can be either provided by programmer assertions "à la" JML [30], or computed using general analysis techniques [14, 13] or Java-oriented ones[36, 19, 17, 7]. `PolyLib` [32] is used to compute

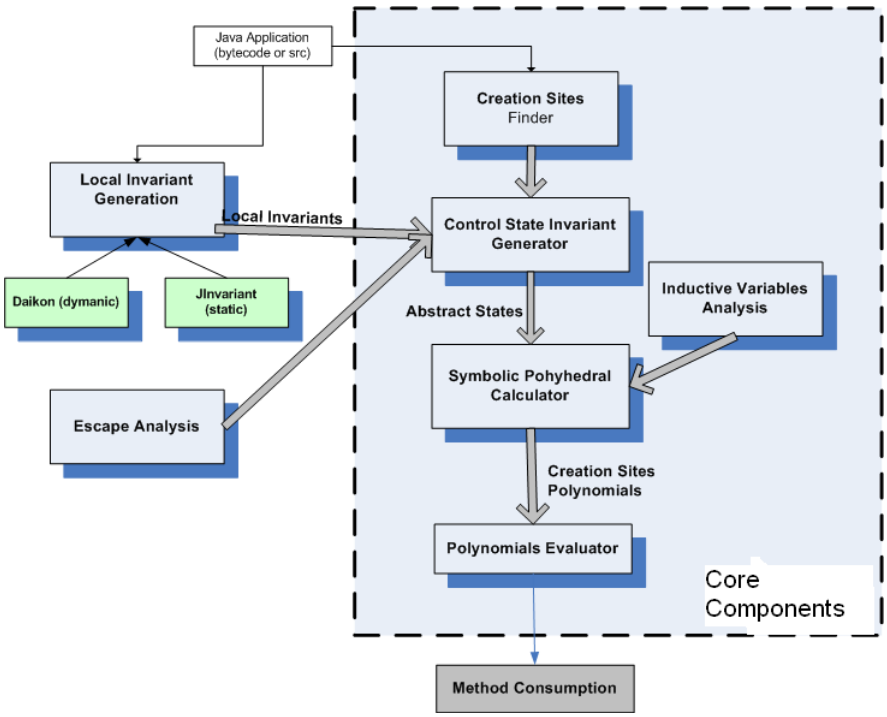


Figure 3: Proof-of-concept tool-suite

Ehrhart polynomials. In the experiments, local invariants were generated using Daikon[17]. It should be noted here that Daikon is a tool for dynamic dectection of "likely" invariants by executing the program over a set of test cases. Even if the properties generated by Daikon have a high probability of being true in all runs, that is, being invariants, they might not be. In our experiments, we have manually verified all properties to be invariants.

None of the techniques for computing invariants deal with our concept of control state invariant since they only compute local invariants. Thus, the tool builds a control state invariant by computing the conjunction of the local invariants that hold in the control locations along the path as explained in Section 2.

Note that the precision of our analysis depends on the accuracy of both the invariant generation and call graph generation techniques (specially in the presence of dynamic binding). Weak invariants and unfeasible calls make our technique to over-approximate too much. In section 6 we comment this issue in more detail.

In order to increase the precision of computed upper-bounds, it is preferable to obtain invariants that only capture what is required to be known about the relevant iteration spaces [11]. A key concept for our characterization of iteration spaces is the set of *inductive variables* for a control location, that is, a subset of program variables which cannot repeat the very same value assignment in two different visits of the given control state (except in the case where the program halts). An invariant that only involves parameters and an inductive variables is called an *inductive invariant*.

To compute inductive variables we developed a conservative dataflow analysis that combines a live variables analysis augmented with field sensitivity with a loop inductive analysis [35]. This problem has been studied for programs that make use of iteration patterns composed of `for` and `while` loops with simple conditions. Handling more complex iteration patterns and types beyond integers is a challenging issue related to finding variant functions for the iteration. In section 6 we brifely discuss our general strategy and we show how the tool currently deals with an iteration pattern pervading Java applications as it is the case of looping over collections. Indeed, while not dealing with recursive programs is an underlying limitation of the approach, handling complex data-structures (such as collections) is not precluded, but is a challenge for building good linear invariants.

Experiments

The initial set of experiments were carried out on a significant subset of programs from JOlden [6] and JGrande [15] benchmarks. It is worth mentioning that these are classical benchmarks and they are not biased towards embedded and loop intensive applications – the target application classes we had in mind when we devised the technique.

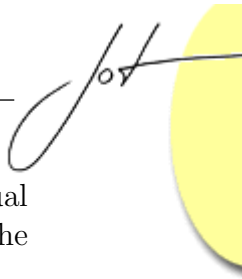
Indeed, our method faced serious obstacles when dealing with these examples. First, in most examples some of the memory-consuming methods reside into recursive structures. Second, inductive variables include not only integer-typed variables but also object fields and complex data-structures.

Despite these issues, the tool was able to synthesize very accurate and non-trivial estimators for the number of object instances created (and memory allocated) in terms of program parameters for two examples that do not feature recursion (mst and em3d examples). In all test cases, execution times were less than 30 sec. in a Pentium 4 3Ghz PC for the core components (Fig. 3): (1) find creation sites, and compute (2) control-state invariants, (3) inductive variables, and (4) Ehrhart polynomials. Moreover, the tool was also able to analyze most non-recursive (and tail-recursive) application methods for the rest of the examples.

All these results were achieved using the **original code** as input for the method and reducing human intervention to a minimum (i.e., creation of test cases for `Daikon`, strengthening some of automatically detected invariants and reducing some of automatically detected inductive sets). Remaining obstacles that prevent fully automatic analysis of some examples are complex data-structures which must be considered part of any set of inductive variables and thus, an integer interpretation of them should be provided by the user to build a useful linear invariant.

These experimental results focused on the allocation estimation (Section 3). The application of our technique to the scoped memory management (Section 4) needs further work.

In order to make the result more readable, the tool computes the number of



object instances created when running the selected method, rather than the actual memory allocated by the execution of the method[4]. Also, we set aside analyzing the standard Java library in order to keep examples manageable.

Table 6 shows the computed polynomials, the analysis time (of core components), and the 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).

Example:Class.Method	Static Analysis			Precision Analysis			
	$\#CS_m$	memAlloc	Time	Param.	#Objs	Estim.	Err%
mst:MST.main(nv)	13	$(2+[\frac{1}{4},0,0,0]_{nv})nv^2$ $+4nv+5$	26.04s	10	240	245	2,00
				20	940	985	5,00
				100	22700	22905	1,00
				1000	2252000	2254005	0,09
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
mst:Graph.Graph(nv)	6	$(2+[\frac{1}{4},0,0,0]_{nv})nv^2$ $+3nv$		10	230	230	0,00
				20	920	960	4,17
				100	22600	22800	0,88
				1000	2251000	2253000	0,09
mst:Graph.addEgdes(nv)	2	$2nv^2$		10	180	200	10,00
				20	760	800	5,00
				100	19800	20000	1,00
				1000	1998000	2000000	0,10
Em3d.main(nN, nD)	28	$6nD \cdot nN + 4nN + 14$	30.57s	(10, 5)	350	354	1,13
				(20, 6)	810	814	0,49
				(100, 7)	4610	4614	0,09
				(1000, 8)	52010	52014	0,01
Bigraph.create(nN, nD)	22	$6nD \cdot nN + 4nN + 8$		(10, 5)	348	348	0,00
				(20, 6)	808	808	0,00
				(100, 7)	4608	4608	0,00
				(1000, 8)	52008	52008	0,00
Node.makeFromNodes	2	$2 \cdot this.fromCount$		10	20	20	0,00
				20	40	40	0,00
				100	200	200	0,00
				1000	2000	2000	0,00
Tree.createTestData(nb)	23	$17nb + 26$	7.22s	10	196	196	0,00
				20	366	366	0,00
				100	1726	1726	0,00
				1000	17026	17026	0,00
Value.createTree(size,sd)	1	$size-1$	2.74s	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:Root.<init>	14	32622	5.82s	-	32412	32622	0,64
(*)health: (recursive) Village.createVillage(l, lab, b, s)	8	$11(4^l-1)/3$		2	55	∞	∞
				4	935	∞	∞
				6	15015	∞	∞
				8	240295	∞	∞
FFT.test(n)	10	$4n + 8$	5.02s	8	38	40	5,00
				32	134	136	1,47
				256	1030	1032	0,19
				1024	4102	4104	0,05
JGFHeapSortBench.JGFinitialise	2	1000001	4.63s	-	1000001	1000001	0,00
JGFCryptBench.JGFinitialise	7	9000113	5.76s	-	9000113	9000113	0,00
JGFSeriesBench.JGFinitialise	1	20000	5.16s	-	20000	20000	0,00

Table 6: Experimental results

These experiments showed that the technique was indeed efficient and very accurate, actually yielding exact figures in most benchmarks. In some cases, the over-approximation was due to the presence of creation sites associated with 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

[4]For simplicity we assume that the function `size(T)`=1 for all type `T`

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[5] (the given result was calculated by hand). For `fft`, the argument n was required to be a power of 2 for not throwing an exception.

Table 7 shows the polynomials that over-approximate the amount of memory captured by methods of the MST and Em3d examples from JOlden. We show only methods that capture some creation sites. For the others, the estimation yields 0 as they do not allocate objects or they escape their scope.

m	$\#\mathbf{CS_m}$	memCaptured(m)
mst		
MST.main(nv)	13	size(mst.Graph) + (size(Integer) + size(mst.HashEntry)) $\cdot nv^2$ + $[1/4, 0, 0, 0]_{nv}$ · size(mst.Hashtable) $\cdot nv^2$ + (size(mst.Vertex) + size(mst.Vertex[])) $\cdot nv$ + 5 · size(StringBuffer)
MST.parseCmdLine()	2	size(java.lang.RuntimeException)+size(Integer)
MST.computeMST(g, nv)	1	size(mst.BlueReturn) $\cdot (nv - 1)$
em3d		
Em3d.main(nN,nD)	26	size(em3d.BiGraph) + nN · (2 · size(em3d.Node) + 4 · size(em3d.Node[]) · nD + 2 · size(double[]) · nD) + 8 · size(em3d.NodelEnumerate) + 4 · size(java.lang.StringBuffer) + size(java.util.Random)
Em3d.parseCmdLine()	6	3 · size(Integer) + 3 · size(java.lang.Error)
BiGraph.create(nN,nD)	2	size(em3d.Node[]) $\cdot nN$

Table 7: Capturing estimation for MST and Em3d examples.

Additional experiments and details about the the tool can be found in [5].

6 DISCUSSION AND FUTURE WORK

Dealing with recursion

As stated, currently we are not dealing with general recursion. This is probably the most challenging theoretical obstacle for our method since some basic concepts are rooted in the assumption of finite call chains. However, not supporting recursion does not constitute a major drawback in many cases since our focus are embedded applications where recursion is a "rara avis". Nevertheless, we are looking for ways of relaxing this limitation like counting the number of possible stack configurations when recursion is eliminated.

[5]Some JOlden programs not considered here also lead to exponential memory usage

Beyond classical iteration spaces

State of complex data-structures may impact the number of times a control state is visited (e.g., iterating a collection). The basic idea to handle this problem is, firstly, to abstract away data-structures into "integer views" (e.g., size of a collection, array length, integer class-attributes, counters standing for iteration progress, largest integer member of the collection, the size of the largest collection inside the collection, the number of objects satisfying a given property, etc.). Then, inductive invariants may be built using those integer-typed variables that capture the relevant state of the data structure (e.g., current index position) and integer-typed expressions over the data-structure that may serve as complexity parameters (e..g, size of array). The tool provides basic functionality to apply this pre-processing for structures such as collections and arrays.

As an example, we illustrate here how to handle collections. Consider an iteration of the form:

```
Iterator it1= collection1.iterator();
while (it1.hasNext() && condition) {
   a = (Type)it1.next();
   ...
}
```

To analyze this kind of pattern the following pre-processing is to be done:

1. As the counting method deals with integer-valued inductive variables, each iterator `it` should be associated to a "virtual" counter it. This counter is initialized when the iterator is created and incremented when the corresponding `it.next()` is called. Consequently, loop invariants involving iterators will include a constraint of the form $\{0 \leq\ it < \texttt{collection.size()}\}$.
2. The parameter to be used when computing the invariant is its `size`.

Figure 4 shows a (very simple) implementation of a dynamic array using a list of fixed sized nodes. The memory allocated by the method `addAll` depends on the size of the collection passed as a parameter. The actual allocation takes place in the method `newBlock` where a new block of memory is allocated only when the previous block is full. Our method yields the following invariant for the control state $addAll.2.add.3$:

$$\begin{aligned}\mathcal{I}^{addAll}_{addAll.2.add.3.newBlock.1} \quad = \quad & \{\text{BSIZE} = 5,\ 0 \leq it < \text{c.size()},\ \text{len} = it,\\ & \text{len} \mod \text{BSIZE} = 0, \text{how} = \text{BSIZE}\}\end{aligned}$$

and the corresponding allocation expression in terms of the collection size[6]:

$$\mathcal{S}(\mathcal{I}^{addAll}_{addAll.2.add.3.newBlock.1}, \{c\}) \quad = \quad \text{c.size()} + [0,4,3,2,1]_{\text{c.size()}}$$

[6]The function $\mathcal{S}$ will add the constraint $\{\ 1 \leq h_1 \leq how\}$ since the involved creation site is a `newA` statement.

```
public class ArrayDim {
  Vector list; int len;
  final static int BSIZE = 5;
  ArrayDim() {
1:  list= new Vector();
2:  len = 0;    }
  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) {
1:  Object[] block=new Object[how];
2:  list.add(block);
3:  return block;
    }
  void addAll(Collection c) {
1:  for(Iterator it=c.iterator();
        it.hasNext();)
    {
2:    add(it.next());
    }
  }
}
```

Figure 4: Collection Example

Improving method precision

When programs feature `if` statements with non-linear conditions or polymorphic invocations, it is usually the case of having control states that, by the control structure, are mutually exclusive but their invariants have non-empty intersection. This implies that some statement occurrences are counted more than once by the current technique.

Consider the following example:

```
0: void test(int n,Object a[]) {
1:    for(int i=1;i<=n;i++) {
2:       if(t(i))
3:           a[i] = new Integer[2*i];
4:       else
5:           a[i] = new Integer[10];
      }
   }
```

If `t(i)` is abstracted away, the invariants at $test.3$ and $test.5$ will be identical:

$\mathcal{I}^{test}_{test.3} = \mathcal{I}^{test}_{test.5} = \{1 <= i <= n\}$

and their corresponding size expressions[7]:

$\mathcal{S}(\mathcal{I}^{test}_{test.3}, n) = n^2 + n$, $\mathcal{S}(\mathcal{I}^{test}_{test.5}, n) = 10n$.

The `computeAlloc` function will sum up these expressions and yield the expression $n^2 + 11n$. This result, although safe, would be too conservative. For instance,

[7]To simplify the explanation, we intentionally omit the size(Integer) factor.

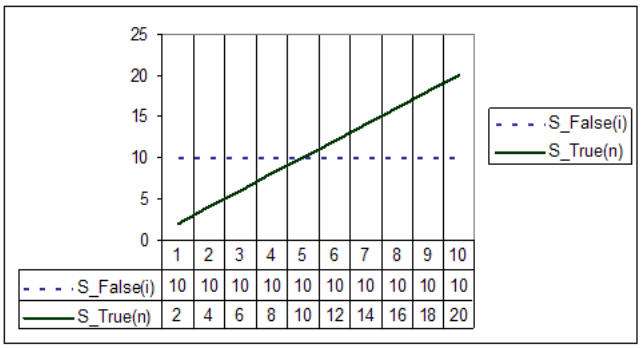


Figure 5: Evolution of size functions for the "test" example

for $n = 6$, the estimated memory utilization for `test` will be 102. Nevertheless, analyzing the program, it is easy to see that the maximum amount consumed is 62. This corresponds to choosing creation site *test*.5 when `i` is between 1 and 5 and taking creation site *test*.3 when `i` is greater than 5 (see figure 5). In [5] we show some advances in the direction of improving precision.

Memory required to run a method

Computing good over-estimations of memory required to run a method in terms of its parameters may have deep impact in software development. Possible applications of this estimator are memory consumption certificates, over-estimation of heap usage, scheduling and dynamic load of application based of memory requirements, etc.

Our general technique is unsensitive to the presence of a memory manager that reclaims unused objects at runtime. Particulary, it does not leverage on the fact that there is a scoped-memory manager that garbage collects unused regions, in which case counting the number of visits to creation sites would yield a very pessimistic upper-bound for the memory required to run. In fact, in this setting, although a method can be potentially invoked several times, there will be at most one active region per method whose size may change according to the calling context (the value assigned to its parameters each time it is invoked).

Consider method $m0$ in the example of figure 1. At location $m0.2$, $m0$ calls $m1$ which calls $m2$. At location $m0.3$, $m0$ calls $m2$. Under a scoped memory management, there will be three active regions for the call chain $m0 \rightarrow m1 \rightarrow m2$, and two active regions for the call chain $m0 \rightarrow m2$. Memory requirements for these two chains should not be summed up since their stack of regions are not simultaneously active.

In [5] we present an initial approach that refines our consumption estimation considering memory reclaiming. Central to this approach is the over-estimation of the largest size required for a region associated to an invoked method. This requires symbolically solving a maximization problem.

Hybrid technique

Approaches like [9, 10] seem suitable for the verification of Presburger expressions accounting for memory consumption annotations for class methods. We believe that it is possible to devise a technique integrating our analysis together with those mentioned type-checking based ones. The approach would be as follows. While methods for data container classes (like the ones provided by standard libraries) are annotated and verified by type-checking techniques, loop intensive applications built on top of those verified libraries may be analyzed using our approach. The idea is to resort to verified annotations in the same spirit as we handle array creation. That is, it would be not necessary to reach the underlying creation sites of the library. Instead, invariants at the method invocation sites may be built by introducing an integer variable with the Presburger expression as upper-bound. Benefits are twofold: first, work done by our technique would be reduced since we would had to deal with significatively smaller call graphs, and second, our ability to synthesize non-linear consumption expressions would entail an increase of expressive power and usability of type-checking based techniques.

7 CONCLUSIONS

We have developed a technique to synthesize non-linear symbolic estimators of dynamic memory utilization. We first presented an algorithm for computing the estimator for a single method. We then specialized it for scope-based memory management. Our approach resorts to techniques for finding invariants and counting integer solutions of linear constraints. We believe that the combination of such techniques, and in particular, their application to obtain specifications that predict dynamic memory utilization is interesting and novel. Besides, it is suitable for accurately analyzing memory utilization in the context of loop-intensive programs. Memory estimators can be used both at compile- and run-time, for example, to set up the appropriate parameters required by the RTSJ scoped-memory API, to over estimate heap usage, to improve memory management and to accurately determine whether a new program can be safely dynamically loaded and scheduled without disturbing other programs behavior.

We have developed a prototype tool that allowed us to experimentally evaluate the efficiency and accuracy of the method on several Java benchmarks. The results were very encouraging. We are currently improving the tool in order to thoroughly test the complete approach (in particular integration with escape analysis) and make the approximations tighter.

Other aspect to explore is the optimization of our method. Slicing techniques and techniques to find inductive variables could help in reducing the number of variables and statements considered when building the invariants. On the other hand, techniques like [22] can be used to eliminate from our analysis creation sites that can be statically pre-allocated.

Acknowledgements This work has been partially supported by projects DYNAMO (Min. Research, France), MADEJA (Rhône-Alpes, France), ANCyT grant PICT 11738, UBACYT 0X20, and IBM Eclipse Innovation Grants.

REFERENCES

[1] D. Bacon, P. Cheng, and D. Grove. Garbage collection for embedded systems. *In EMSOFT'04*, 2004.

[2] B. Blanchet. Escape analysis for object-oriented languages: application to Java. In *OOPSLA 99*, volume 34, pages 20–34, 1999.

[3] G. Bollella and J. Gosling. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

[4] V. Braberman, A. Ferrari, D. Garbervetsky, P. Listingart, and S. Yovine. Jscoper: Eclipse support for research on scoping and instrumentation for real time java applications. In *ETX 2005*, San Diego, USA, October 2005.

[5] V. Braberman, D. Garbervetsky, and S. Yovine. On synthesizing parametric specifications of dynamic memory utilization. *Internal Report. Verimag, France*, 2005.

[6] 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.

[7] B. Chang and K. Rustan M. Leino. Infering object invariants. In *AIOOL'05*, 2005.

[8] S. Cherem and R. Rugina. Region analysis and transformation for Java programs. *ISMM'04*, 2004.

[9] W. Chin, S. Khoo, S. Qin, C. Popeea, and H. Nguyen. Verifying safety policies with size properties and alias controls. In *ICSE 2005*, 2005.

[10] W. Chin, H. H. Nguyen, S. Qin, and M. Rinard. Memory usage verification for oo programs. In *SAS 05*, 2005.

[11] 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.

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

[13] 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.

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

[15] 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.

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

[17] 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.

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

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

[20] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program Instrumentation and Run-Time Analysis of Scoped Memory in Java. *RV 04, ETAPS 2004, ENTCS, Barcelona, Spain*, April 2004.

[21] D. Gay and A. Aiken. Language support for regions. In *PLDI 01*, pages 70–80, 2001.

[22] O. Gheorghioiu. Statically determining memory consumption of real-time java threads. MEng thesis, Massachusetts Institute of Technology, June 2002.

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

[24] R. Henriksson. Scheduling garbage collection in embedded systems. *PhD. Thesis, Lund Institute of Technology*, 1998.

[25] T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J-Ph. Lesot, and F. Parain. Memory management for real-time Java: an efficient solution using hardware support. *Real-Time Systems Journal*, 2002.

[26] M. Hofman and S. Jost. Static prediction of heap usage for first-order functional programs. In *POPL 03*, New Orleans, LA, January 2003.

[27] J. Hughes and L. Pareto. Recursion and dynamic data-structures in bounded space: towards embedded ml programming. In *ICFP '99*, 1999.

[28] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In *POPL '96*, pages 410–423. ACM, 1996.

[29] Ch. Kloukinas, Ch. Nakhli, and S. Yovine. A methodology and tool support for generating scheduled native code for real-time java applications. In *EMSOFT'03*, Philadelphia, USA, October 2003.

[30] 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.

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

[32] V. Loechner. Polylib: A library for manipulating parameterized polyhedra. Technical report, ICPS, France, 1999.

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

[34] B. Meister. *Stating and Manipulating Periodicity in the Polytope Model. Applications to Program Analysis and Optimization.* PhD thesis, December 2004.

[35] Flemming Nielson, Hanne R. Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

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

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

[38] 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.

[39] T. Ritzau and P. Fritzon. Decreasing memory over-head in hard real-time garbage collection. *In EMSOFT'02, LNCS 2491*, 2002.

[40] A. Salcianu. Pointer analysis and its applications for java programs. SM Thesis, Massachusetts Institute of Technology, Cambridge, Massachusetts, September 2001.

[41] A. Salcianu and M. Rinard. Pointer and escape analysis for multithreaded programs. In *PPoPP 01*, volume 36, pages 12–23, 2001.

[42] F. Siebert. Eliminating external fragmentation in a non-moving garbage collector for Java. *CASES'00*, 2000.

[43] L. Unnikrishnan, S.D. Stoller, and Y.A. Liu. Optimized live heap bound analysis. In *VMCAI 03*, volume 2575 of *LNCS*, pages 70–85, January 2003.

[44] 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.

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

ABOUT THE AUTHORS

Víctor Braberman Ph.D. in Computer Science and full-time Associate Professor in CS. Department of University of Buenos Aires (Argentina) working in the area of Software Engineering . His is an active researcher in models, abstractions and verification of real-time and distributed systems. He can be reached at vbraber@dc.uba.ar.

Diego Garbervetsky Lecturer in the CS. Department of University of Buenos Aires (Argentina) finishing his Ph.D. in computer science. His research field is Embedded Systems, Formal Verification and Program Analysis focused on prediction of dynamic memory utilization. He can be reached at diegog@dc.uba.ar.

Sergio Yovine Ph.D. in Computer Science. Full time researcher at Centre National de la Recherche Scientifique (CNRS), France. He works in the area of Tools for Formal Verification of Real Time and Embedded Systems. He can be reached at sergio.yovine@imag.fr.