On transforming Java-like programs into memory-predictable code

Diego Garbervetsky Sergio Yovine Víctor Braberman
Martín Rouaux Alejandro Taboada
Departamento de Computación, FCEyN, UBA, Argentina
{diegog, syovine, vbraber, mrouaux, ataboada}@dc.uba.ar

ABSTRACT

The `ScopedMemory` class of the RTSJ enables the organization of objects into regions. This ensures time-predictable management of dynamic memory. Using scopes forces the programmer to reason in terms of *locality*, to comply with RTSJ restrictions. The programmer is also faced with the problem of providing upper-bounds for regions. Without appropriate compile-time support, scoped-memory management may lead to unexpected runtime errors.

This work presents the integration of a series of compile-time analysis techniques to help identifying memory regions, their sizes, and overall memory usage. First, the tool synthesizes a scoped-based memory organization where regions are associated with methods. Second, it infers their sizes in *parametric* forms in terms of relevant program variables. Third, it exhibits a parametric upper-bound on the total amount of memory required to run a method. We present some preliminary results showing that semi-automatic, tool-assisted generation of scoped-based code is both helpful and doable.

1. INTRODUCTION

The use of object-oriented languages, like Java, for programming embedded, real-time applications poses the challenge of predicting their execution time and memory consumption. A factor that has an important impact on these issues is automatic memory management.

There have been significant improvements in the development of automatic garbage collectors in order to meet real-time requirements: such us Metronome [4], Sun GC based on [20] and JamaicaVM [27]. These GCs enable automatic memory management without compromising time-predictability. However, there is still the problem of predicting how much memory a program needs to be executed without rising an out-of-memory exception. This issue is particularly important in critical systems requiring certification, and in memory-constrained embedded applications. Determining upper-bounds on memory consumption is very hard in the presence of GC due to the difficulty of quantifying the behavior of GCs with respect to application's memory usage.

An interesting approach to gain control on time and space is to change the memory organization model, and to group objects in regions [28, 18]. The idea behind region-based memory management is to group objects of similar lifetimes: within a region, one cannot deallocate any individual object, but must wait until the region can be destroyed as a whole. The Real-time Specification for Java (RTSJ) [19] supports application-level region-based memory management through `ScopedMemory` areas. This environment guarantees predictable-time memory operations but it makes programming more difficult. Complying with RTSJ rules complicates reusing legacy code without careful modification [23] (that also applies for the Standard Library) and it forces the programmer to adopt new coding habits and to reason in a new paradigm quite different from Java. In addition, in order to develop efficient region-based memory managers, as stated in the RTSJ, *it is necessary to provide upper-bounds of the amount of memory to be allocated in each region.*

This paper presents an integration of a series of techniques into a tool-suite to help developers the tackle aforementioned problems. That is, the paper explains how a chain of techniques helps programmers to produce sound and predictable scope-memory managed code from conventional Java code. In particular, the tools assist programmers to generate memory regions, by the aid of escape analysis [26, 25], and to manually fine tuning memory scopes using a region edition tool [15] or by using explicit annotations.

This tool-chain also provides two key features, which are independent of the way objects are organized into regions. The first one is the ability to determine the size of all memory regions as a function of program variables [6]. This is required by the RTSJ for creating memory scopes. The second unique feature is the symbolic over-approximation of the total amount of memory required to execute a method (including its callees) [5]. This enables checking whether the method can be safely executed without running out of memory. That is, the resulting expression (actually, a polynomial) can be seen both as a *pre-condition*, stating that the method requires that amount of free memory to be available before executing and also as a *certificate*, ensuring that the method will not use more memory than the specified amount.

We illustrate the use of the tool-chain with two case studies: an aircraft collision detector and a banking application. These experiments show how the tool-chain can as-

JTRES'09, September 23-25, 2009 Madrid, Spain

sist programmers in several ways, by proposing non-evident memory scopes, enabling manipulation of the memory organization, yielding parametric expressions to appropriately dimension regions at runtime, and providing upper-bound of memory footprint. In particular, this helps analyzing at compile-time the effects of several possible memory organizations, in order to choose the best one. We believe that these case studies are representative of real applications. Thus, they are useful both for validating the preliminary ideas, and for showing the current limitations of the tools.

2. TOOL OVERVIEW

An architectural view of the key functional components is shown in Fig. 1.

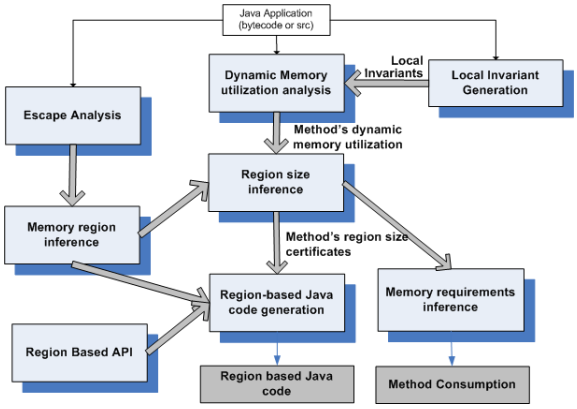

Figure 1: Tool flow.

Region synthesis takes care of analyzing the Java application code in order to identify which, where, and how objects could be placed together in regions according to their lifetimes. This module is composed of the following parts.

- *Static escape analysis* determines object lifetimes. The current prototype is instantiated to use the object-interference static analysis developed in [24].
- *Memory region inference* statically associates objects (creation sites) to regions based on the output of the escape analysis. Since the number and sizes of the regions may impact performance and memory requirements, `JScoper` an Eclipse plug-in [15] allows visualizing and manual editing memory regions in order to apply different region assignment strategies.
- *Region based API* is the set of classes that implement the region-based manager. Currently, there are four different implementations: one based on Jikes[3], one based on JITS[24], and one based on the RTSJ runtime of JamaicaVM[27], and, finally a simulator that runs on a standard Java VM and GJC using [18].
- *Region-based code generator* translates standard Java code to a region-based one (see §4.2).

Quantitative memory analysis characterizes the behavior of the application code with respect to the amount and lifetime of dynamic memory allocated by that code. Since such measures may depend on application parameters, the analysis is *parametric*, that is, it yields an expression on the parameters. For this, this module is composed of the following parts.

- *Dynamic memory utilization analysis* computes a parametric conservative approximation of the amount of memory *requested* by a method. This module implements the static analysis technique for synthesizing parametric specification of dynamic memory consumption proposed in [6].
- *Region-size inference* outputs an expression on method parameters which, evaluated at runtime, gives the *size of the memory region* associated with the method. This analysis combines the result of the region inference phase with memory utilization analysis as described in [6]. Region size may be utilized by a JVM to perform efficient memory allocation and deallocation of regions.
- *Memory requirements inference* provides a parametric conservative approximation of the amount of memory *required* to safely run a method with the synthesized region-based manager. This measure gives a parametric bound of the maximum amount of memory occupied by the region stack all along the execution of the program [5].
- *Local invariant generation* produces the invariants required by the memory prediction technique [6]. The current prototype, uses `Daikon` [14] to generate likely invariants, which are then verified.

3. PRELIMINARIES

3.1 Running example

Fig. 2 presents a toy schematic example we will use to introduce the different aspects of the technique.

```
  void m0(int mc) {
1:   m1(mc);
2:   B[] m2Arr=m2(2 * mc);
  }
  void  m1(int k) {
3:  for (int i = 1; i <= k;
4:    A a = new A();
5:    B[] dummyArr= m2(i);
    }
  }
  B[] m2(int n) {
6:  B[] arrB = new B[n];
7:  for (int j = 1; j <= n;
8:    arrB[j-1] = new B();
9:    C c = new C();
10:   c.value = arrB[j-1];
    }
11: return arrB;
   }
```

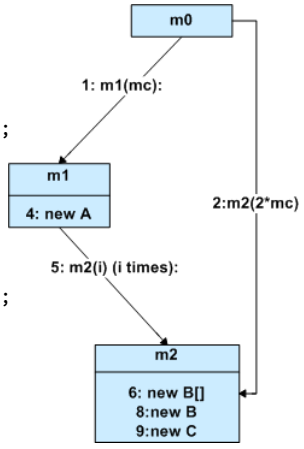

Figure 2: An example program with its detailed call graph

3.2 Modeling allocation sites

In order to obtain more precise bounds and fined grained regions, the techniques distinguish program locations not only by a "method-local" control location but also by the different control stack configurations that lead to that location. Specifically, allocation sites are identified by the call chain starting from the method under analysis (MUA) and finishing in a `new` statement. These chains are denoted as *Creation Sites* and form a particular case of *Control States* which are basically sequences of program locations modeling the *control* part of call stack configurations. The *data*

counterpart of a control state is represented using *Control State Invariant* that models a set of possible program states for a given of control state. That is, as creation sites represent a traversal trough several methods, global invariants are used to characterize the potential set of valid program states reachable at a given control state (i.e. the data part of the call stack). For instance: $m0.2.\mathbf{m2.6}$ is a creation site that represent the program location $m2.6$ with control stack $m0.2$. $m0.1.m1.5.\mathbf{m2.6}$ is a creation site that represent the program location $m2.6$ with control stack $m0.1.m1.5$

EXAMPLE: In this example the creation sites reachable from $m0$, $m1$ and $m2$ are:

$$\begin{aligned} CS_{m0} &= \{\text{m0.1.m1.4, m0.1.m1.5.m2.6, m0.1.m1.5.m2.8,} \\ &\quad \text{m0.1.m1.5.m2.9, m0.2.m2.6, m0.2.m2.8,} \\ &\quad \text{m0.2.m2.9}\} \\ CS_{m1} &= \{\text{m1.4, m1.5.m2.6, m1.5.m2.8, m1.5.m2.9 }\} \\ CS_{m2} &= \{\text{m2.6, m2.8, m2.9 }\} \end{aligned}$$

To accurately compute call chains and getting all allocation sites reachable from the application under analysis is required a precise call graph (the tool relies on`Soot` [29] for that).

Currently, there are two main limitations in the analyses: (1) There is no recursion and all allocation sites in the application can be reached by static analysis. (2) The amount of "hidden" memory allocated by native methods or by the virtual machine itself cannot be quantified with this technique.

4. REGION SYNTHESIS

Scoped-memory management is based on the idea of allocating objects in *regions* associated with the lifetime of a computation unit, i.e., its scope. A computational unit can be a method, a thread, etc. When a computational unit finishes its execution, its objects are automatically collected. This approach imposes restrictions on the way objects can reference each other in order to avoid the occurrence of dangling references. An object $o1$ belonging to region r references an object $o2$ only if one of the following conditions holds: $o2$ belongs to r; $o2$ belongs to a region that is always active when r is active or $o2$ is in the Heap. An object $o1$ cannot point to an object $o2$ in region r if: $o1$ is in the heap or r is not active at some point during $o1$'s lifetime.

This kind of memory management mechanism is used to overcome the problem of time-predictability of garbage collector. In addition to this, our analysis take advantages on the imposed order in allocations and deallocations in order to predict memory requirements (space-predictability). In particular, this work assumes, 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. For instance, in a single-threaded program, if each region is associated with one method, then there is a region stack where the number and ordering of active regions corresponds exactly to the appearances of each method in the call stack. Region whose lifetime is associated with method's lifetimes are denoted as m-region.

4.1 Inferring method regions

In order to alleviate the burden and risks of manual operation of regions, memory regions are automatically and safely inferred from program code based on escape analysis [17]. This could be an starting point for further refinement by adding user annotations if better precision is required. Intuitively, an object *escapes* a method m when its lifetime is longer than m's lifetime. It cannot be safely collected when this unit finishes its execution. An object is *captured* by the method m when it can be safely collected at the end of the execution of m. It is possible to synthesize a memory organization that associates a memory region to each method m (called m-region) in such a way that scoped-memory restrictions (like RTSJ) are fulfilled by construction.

Determining precise object lifetime is undecidable. Therefore, we advocate a semi-automatic approach by making the programmer participate in the analysis and the transformation process. For this, tool chain currently involves the use of automatic static analysis tools [25] in combination with a tool, called `JScoper` [15], allowing visualization and, optionally, the edition of memory regions. This tool also is able to generate java code using a region-based memory managed mechanism using the originally synthesized or manipulated regions.

EXAMPLE: Using escape analysis the tool infers the creation sites that escape and are captured by $m0$, $m1$, and $m2$ in the example presented in Fig. 2. Objects referred to by the allocation site $m2.9$ do not escape the scope of $m2$ provided they are not referenced from outside $m2$ by any parameter or return value. Objects referred-to by the allocation site $m2.6$ are pointed-to by `arrB` which is returned by $m2$ escaping its scope. The same happens with the objects referred-to by $m2.8$ which are pointed-to by `arrB[i]` which is referenced-to by `arrB`. The object referred-to by $m1.4$ is allocated in method $m1$ and is not referenced from outside. Since `dummyArr` refers to the objects returned by $m2$ this variable has references to objects referred-to by $m2.6$ and $m2.8$. Since `dummyArr` is a local variable not referenced by a variable or field reachable from outside, those objects do not escape the scope of method $m1$. In fact, all objects reachable from $m1$ are captured by this method.

The same procedure is applied to $m0$. The resulting escape and capture information is the following:

$$\begin{aligned} escape(m0) &= \{\} \\ capture(m0) &= \{m0.2.m2.6, m0.2.m2.8\} \\ escape(m1) &= \{\} \\ capture(m1) &= \{m1.4, m1.5.m2.6, m1.5.m2.8\} \\ escape(m2) &= \{m2.6, m2.8\} \\ capture(m2) &= \{m2.9\} \end{aligned}$$

Using this information the tool safely infer the following regions:

$$\begin{aligned} region(m0) &= \{m0.2.m2.6, m0.2.m2.8\} \\ region(m1) &= \{m1.4, m1.5.m2.6, m1.5.m2.8\} \\ region(m2) &= \{m2.9\} \end{aligned}$$

4.2 Compilation and runtime

In order to perform scoped-memory management at program level the tools uses an API enabling manipulation of method-scoped memory regions [17]. This API has constructs to create and destroy m-regions and a mechanism to associate objects to m-regions. Objects has to be identified by IDs, such us the program location where they are created (option used in this work). The programmer, indicates (in general at region creation) which are the objects the region

is going to allocate, by providing the IDs of these objects. At object creation, the object identifies itself by presenting its ID. The memory manager checks whether that ID is registered by at least one m-region, the manager allocates the object in the last region that registered that object or, by default, in the active region.

This approach was implemented in a region-based manager for **JikesVM**[3], a code translation to **RTSJ** and a code translation using a region library [18]. Code for registering objects to regions is automatically generated by using the region synthesis output. For **JITS** we took a slightly different approach.

JikesVM. We are currently implementing a scoped-based region manager were every method header is annotated with region information using the registering mechanism explained before. The implementation heavily uses the fact that regions sizes are provided (because we can infer it) to implement a very simple intra-region allocator.

RTSJ. To generate code, the approach is setting up a method wrapper, say `m_wrapper` for each method `m`. The code of `m` is replaced by a call to the wrapper which is responsible for creating a region and a runnable, and entering the region with the runnable:

```
... m_wrapper( ... ) {
    Region m_reg = new Region(m_reg_size(...));
    Runnable m_run = new m_run(this, ...);
    m_region.enter(m_run);
    ...
}
```

where class `Region` extends RTSJ-class `LTMemory`. The (generated) method `m_reg_size(...)` evaluates the expression computed by the *Region size inference* module on the parameters of `m`. To allocate an object in a region, `Region` uses the registration mechanism mentioned before. Every `new` in `m` is replaced by a call `r.newInstance(Class.forName(...))` in the `run` method of `m_run`. The mechanism is explained in detail in [17]. This approach has been implemented and tested on the JamaicaVM.

JITS. JITS is a software framework dedicated to generation, deployment and execution of low-footprint embedded J2SE Java applications. In [24] JITS is extended to support a regions: bytecode interpreter was modified to keep track of regions, without changing nor annotating the bytecode itself. The class loader was also modified to take into account the metadata computed by the static analysis. This implementation was successfully used to execute an embedded real-time MP3 converter.

5. QUANTITATIVE MEMORY UTILIZATION ANALYSIS

This section shows the components related with the analysis of the quantitative dynamic memory utilization. This set of techniques span from computing total memory allocations to predicting memory-region sizes and upper bounds of application memory requirements considering a memory reclaiming mechanism.

5.1 Dynamic memory utilization analysis

To get a flavor of the analysis, 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();
 }
}
```

The amount of memory requested by `m2` is $size(\mathtt{B}) \cdot \mathtt{n}$, as it creates `n` instances of `B`. `m1` creates `k` instances of `A`, and calls `k` times `m2`. At each invocation `m2(i)` will request `i` instances of `B`, resulting in a total amount of $\sum_{i=1}^{k} \mathtt{i} = \frac{1}{2}(\mathtt{k}^2 + \mathtt{k})$ instances of `B`, which have to be added to the `k` instances of `A` directly allocated by `m1`.

In [6] we have presented a general technique to perform quantitative memory analysis, which is indeed capable of giving the exact answer for this example. The idea is a follows. The amount of dynamic memory allocated is related to the number of visit to `new` statements. Such number can be computed by characterizing the iteration space of each allocating statement. Using a combinatorial approach, this can be related to the number of possible valuations of variables that it might feature at its control location. Furthermore, this can be related to the number of integer solutions of a predicate constraining variable valuations at its control location (i.e. an invariant). For linear invariants, the number of integer solutions is equivalent to the number of integer points which can be expressed as an Ehrhart polynomial [11].

In the example above, the linear invariant $\phi \equiv \{1 \leq i \leq k, n = i, 1 \leq j \leq n\}$ characterizes the iteration space at the allocation site `B b = new B()`. The number of integer points in this space is exactly the number of times the `new` statement is executed, which is equal to the overall number of instaces of `B` which are created by the repeated calls to `m2` by `m1`, i.e., $\frac{1}{2}(k^2 + k)$.

A detailed view of the components involved in the module that implements this approach is shown in Fig. 3. The flow is the following:

1. The *Creation Site finder* identifies allocation sites (`new` statement) reachable from the method under analysis (MUA).
2. The *Control State Invariant generator* generate linear invariants describing possible variables valuations at each allocation site. Invariants are generated by combining local invariants obtained from the *Local Invariant Generator*.
3. The *Symbolic Polyhedral Calculator* is used to count the number solutions for the invariant in terms of MUA parameters (# of visits to the allocation site). It produces the polynomials representing parametric expressions of the number of solutions of the given invariants. These expressions are adapted to consider the size of allocated objects (as a function of their types).

Computing invariants The memory-consumption analysis technique relies on having invariants that constrain the possible variable assignments of a specific program point. *Control State Invariants* are fundamental for the approach as they not only are used to model the potential variables valuation at a control state, they also are used to bind the parameters of the MUA with the different variables in the global state.

Local invariants can be either provided by programmer assertions "à la" JML, or computed using general analysis

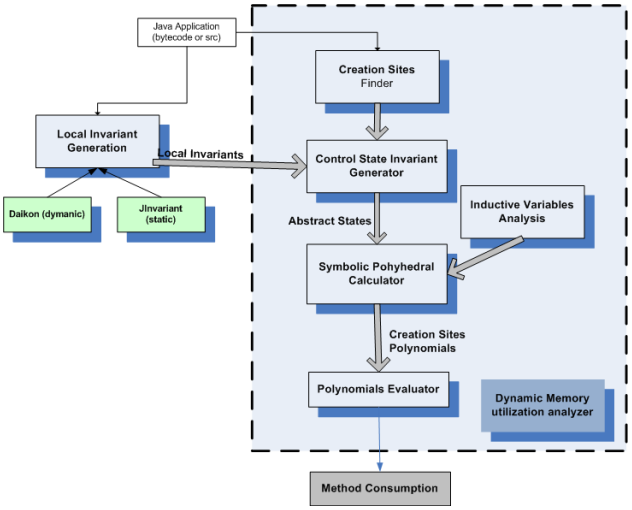

Figure 3: Quantitative memory analysis flow

techniques or Java-oriented ones. Currently, the prototype uses `Daikon` for dynamic detection 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. During experimentation, invariant have been manually verified to ensure its correctness. To produce useful invariants the tool "guides" `Daikon` to search for program invariants. Basically, code is instrumented (at call sites and allocation sites) to introduce new local variables for expressions presumed to have impact in the number of times allocation sites are visited (e.g, integer class fields, size of collections, length of arrays and strings, etc. [16]).

The tool builds a control state invariant by computing the conjunction of the local invariants that hold in the control locations along the path. That task is performed by the *Control State Invariant Generator*.

Counting the number of visits The *Symbolic polyhedral calculator* represents a tool that can manipulate linear invariants. It consist of the algorithms used to count the number of solutions of a given invariant. To count the number of solutions of a predicate is important to identify which variables are fixed (parameters) and which are free. For linear invariants the technique presented in [11] obtain polynomials representing the number of solution of those invariants. Since the number of solutions may depend on the value assigned to the parameters, the polynomial variables are indeed the invariants parameters.

In general, invariants do not need to predicate about every variable in a global state, it is enough to constrain them to a subset of *inductive variables* (see bellow).

Computing memory consumption expressions To get the expressions that bound the amount of memory requested by a method firstly is computed a function called $\mathcal{S}(mua, cs)$ that given a creation site cs yields an expression in terms of mua parameters that bounds the amount of memory requested by cs. That function is computed by multiplying the number of visits of a creation site by the size corresponding to the type of the allocated object. For arrays the computation is a little bit trickier (more details can be found [6]). Once the size of all creation sites is computed, the total amount of memory requested by a method can be easily computed by summing up the size expression for all creation sites reachable from the method.

Note that the precision of the analysis depends on the accuracy of both invariant and call graph generation techniques (specially in the presence of dynamic binding). Weak invariants and infeasible calls may make the technique over-approximate too much. In [6] we show some ideas in order to try to mitigate this problem. In particular it is fundamental to discover what a set of inductive variables.

Computing set of inductive variables A key concept for the 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 loops forever).

An invariant that only involves parameters and a set of inductive variables is called an *inductive invariant*. As the number of visits of a given statement is thus associated to the number of solutions of an inductive invariant. Sets of inductive variables are approximated using conservative dataflow analysis that combines a live variables analysis augmented with field sensitivity with a loop inductive analysis [21]. 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 and it is related to finding loop variant functions. In [6] there is a brief discussion about different strategies for different iteration patterns. In particular the tool currently deals with some patterns of iterators looping over collections.

5.2 Region size inference

Recall that m-regions are defined by the set creation sites that are captured by a m. Thus, the size of the m-region is directly associated with the size and the number of objects it captures. Since the memory utilization analysis uses creation sites as input to its algorithm bound for region sizes can be obtained by simply applying the previous technique to the set of creation sites captured by a method.

Notice that the obtained bounds are parametric in terms of method parameters modeling the fact that region sizes may vary according to the calling context. In a similar way, the technique can compute the amount of memory that escapes the method and has to be collected by methods that precede it in the call stack.

EXAMPLE: Table 1 shows the computed regions sizes for the example presented in Fig. 2 using the synthesized region information (see §4.1).

Table 1: Polynomials approximating regions sizes of the running example

$$
\begin{aligned}
\texttt{memCap}(m0)(mc) &= \mathcal{S}(m0, m0.2.m2.6)(mc) \\
&\quad +\mathcal{S}(m0, m0.2.m2.8)(mc) \\
&= (\texttt{size}(B[]) + \texttt{size}(B)).(2mc) \\
\texttt{memCap}(m1)(k) &= \mathcal{S}(m1, m1.4)(k) + \mathcal{S}(m1, m1.5.m2.6)(k) \\
&\quad +\mathcal{S}(m1, m1.5.m2.8)(k) \\
&= \texttt{size(A)}k + (\texttt{size}(B[]) \\
&\quad +\texttt{size}(B)).(\frac{1}{2}k^2 + \frac{1}{2}k) \\
\texttt{memCap}(m2)(n) &= \mathcal{S}(m2, m2.9)(n) = \texttt{size}(C).n
\end{aligned}
$$

5.3 Inference of memory requirements

To address the problem of computing memory requirements, the work presented in [5], proposes a technique to over-approximate the amount of memory *required* to run a method. Given a method the technique yields a polynomial that upper-bounds the amount of memory necessary to *safely* execute the method and all methods it calls, without running out of memory. This polynomial can be seen as a *pre-condition* stating that the method requires that much free memory to be available before executing, and also as a *certificate* ensuring the method is not going to use more memory than the specified amount.

The chosen strategy is to leverage on the ability of inferring memory regions, computing their sizes and the fact that it is easy to characterize where (and when) regions are created and destroyed. Specifically the algorithm for computing dynamic-memory requirements takes into account that m-region's lifetime is determined by method's lifetime.

Fig. 4 shows the main components of the tool that computes the memory requirements.

Modeling region-stacks Knowing how to compute the size of a *mua*-region is not enough: to compute the amount of memory required to run a method it is necessary to consider also the sizes of all m-regions of every method that may be called during the execution of *mua*. In this model potential region stacks are closely related with paths in the application call graph. There are two important facts to take into account: (1) There are some region stack configurations that cannot happen at the same time. (2) Given a path π in the call graph there will be always a region-stack of length $|\pi|$ but featuring different region sizes depending on the of the values assigned to the parameters of each method in π when invoked (the calling context).

Thus, to approximate the amount of memory necessary to safely run a method it is enough to consider every potential path in the call graph starting from the *mua* and, for each one, the largest size the associated m-regions can get. This estimation needs to be expressed in terms of the formal parameters of *mua*. However, memory requirements of the callees are expressed in terms of their own parameters. Therefore, there is also a need of some sort of binding between *mua* parameters and callees parameters. That binding is performed by leveraging on program invariants as in the case of computing consumption for creation sites. Those invariants are denoted as *binding invariants* since they are control state invariants that explains not only the relationship among variables in the application call stack, but also the link between the caller arguments and callee parameters. Binding invariant model the data counterpart of the calling context givel the paths in the call graph.

Lets $\texttt{MaxRegSize}^{\pi.m}_{mua}$ be the function that yields an expression in terms of the *mua*parameters of the size of the largest m-region created by any call to m with control stack π in a program starting at method *mua*. π represents the calling context used to restrict the maximization.

Suppose that method a m calls methods $m1$ and then calls $m2$. Regions for $m1$ and $m2$ can not be active at the same time since $m2$ is executed just after $m1$ finishes it execution. Thus, the memory requirement for running m is composed by its own region size plus the maximum between region sizes for $m1$ and $m2$

In general, this function can be defined as follows:

$$\begin{aligned}\texttt{memRq}^{\pi.m}_{mua}(p_{mua}) \;=\; & \texttt{MaxRegSize}^{\pi.m}_{mua}(p_{mua}) + \\ & \max\{\texttt{memRq}^{\pi.m.l.m_i}_{mua}(p_{mua}) \\ & \mid (m,l,m_i) \;\in\; edges(CG_{mua} \downarrow \pi.m)\}\end{aligned}$$

where $CG_{mua} \downarrow \pi.m$ is a projection over the path $\pi.m$ of the call graph of the program starting at method *mua* and *edges* is the set of its edges.

Note that this recursive definition leads to an *evaluation tree* where leaves are related with `MaxRegSize` problems and nodes with *max* or *sum* operations. Since the objective is to evaluate this formula in run-time (i.e. when method parameters are instantiated) is important to make the evaluation as fast as possible. That is why is it important to simplify the underlying evaluation tree as much as possible. Nevertheless, evaluation tree do not affect predictability since its size is know at compile time. Also note that, in order to properly define `memRq` it is necessary to rule out recursive calls. In other words, the underlying evaluation tree has to be finite [5].

Finally, in order to safely predict the amount of memory required by the *mua*, it is necessary to consider also the objects that were allocated during its execution but cannot be collected when it finishes. Since the escape property is absorbent, it is enough to consider only objects escaping the *mua* ([5]). Thus, function that approximates memory requirements is defined as follows:

$$\boxed{\texttt{memRq}_{mua}(p_{mua}) = \texttt{memEsc}(mua)(p_{mua}) + \texttt{memRq}^{mua}_{mua}(p_{mua})}$$

EXAMPLE: Let $\texttt{MaxRegSize}^{m0\ldots m'}_{m0}$ be the size of the maximum m'-region in terms of $m0$ for a control stack given by a path starting from $m0$ and finish in a method m'. The size of memory required to run $m0$ is computed as follows:

$$\begin{aligned}\texttt{memRq}_{m0}(mc) \;=\; & \texttt{memEsc}(m0)(mc) + \texttt{MaxRegSize}^{m0}_{m0}(mc) \\ & + max\{\texttt{MaxRegSize}^{m0.1.m1}_{m0}(mc) + \texttt{MaxRegSize}^{m0.1.m1.5.m2}_{m0}(mc), \\ & \texttt{MaxRegSize}^{m0.2.m2}_{m0}(mc)\}\end{aligned}$$

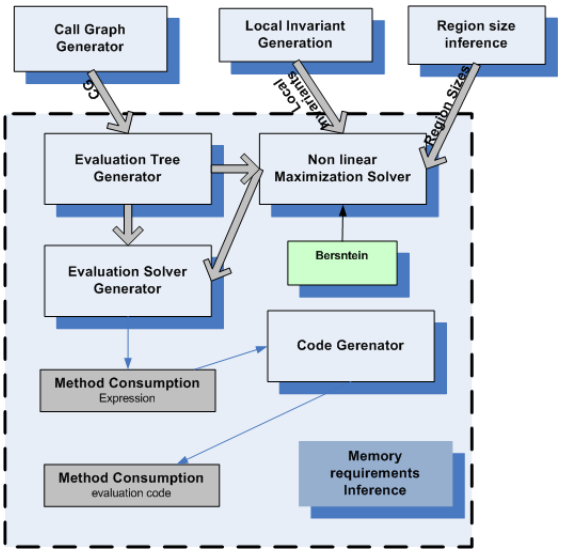

Figure 4: Memory Requirements tool

Maximizing region memory sizes As mentioned, the size of every m-region may vary according to its calling context. For every method m' reachable from the *mua*, we denoted `MaxRegSize` to represents the maximum size an m'-region may reach restricted by a call chain π starting from the *mua*. Similarly to the technique presented in **??**, to compute this expression is required the use of program invariants to bind the method m' parameters with the MUA parameters.

Let $mua...m'$ a path starting from *mua* finishing in a method m'. Maximum size of region is modeled as follows:

$$\begin{aligned}\texttt{MaxRegSize}_{mua}^{mua...m'}(P_{mua}) &= \text{Maximize } \texttt{memCap}(m')(P_m) \\ &\quad \text{subject to } I_{mua}^{mua...m'}(P_{mua}, P_m, W)\end{aligned}$$

$\texttt{memCap}(m')$ (the size of an m'-region) is a polynomial in terms of m' parameters and the invariant $I_{mua}^{mua...m'}$ binds m' parameters with the MUA parameters[1].

This formula characterizes a non-linear maximization problem whose solution is an expression in terms of MUA parameters. Since the goal is to avoid expensive run-time computations it is necessary to perform off-line reduction as much as possible at compile time. Off-line calculation also means that the problem must be stated parametrically.

To solve this parametric maximization problem, the tool uses an approach based on the Bernstein expansion for handling parameterized multivariate polynomial considered over a parametric polyhedron [12]. Roughly speaking, given a polynomial and a restriction given by a parametric polyhedron the technique provides a set of polynomials candidates which bound the original polynomial in the domain given by the parametric polyhedron. The most interesting aspect of the technique is that obtained polynomials are in terms of the parameters of the restriction.

In this case, the polynomial is given by the `memCaptured` estimator (see §1) and the restriction is given by a binding invariant. The maximization problem can be therefore solved by picking the maximum Bernstein coefficient.

EXAMPLE: For the running example, the expression for `MaxRegSize` for each possible region is:

$$\begin{aligned}\texttt{MaxRegSize}_{m0}^{m0.2.m2}(mc) &= \text{Maximize } \big(\texttt{size}(C).n\big) \\ &\quad \text{subject to } \{n = 2mc\} \\ &= (\texttt{size}(C))2mc \\ \texttt{MaxRegSize}_{m0}^{m0.1.m1.5.m2}(mc) &= \text{Maximize } \big(\texttt{size}(C).n\big) \\ &\quad \text{subject to } \{k = mc, 1 \le i \le k, n = i\} \\ &= (\texttt{size}(C))mc \\ \texttt{MaxRegSize}_{m0}^{m0.1.m1}(mc) &= (\texttt{size}(B[]) + \texttt{size}(B)) \\ &\quad .(\frac{1}{2}mc^2 + \frac{1}{2}mc) + \texttt{size(A)}mc \\ \texttt{MaxRegSize}_{m0}^{m0}(mc) &= (\texttt{size}(B[]) + \texttt{size}(B))2mc\end{aligned}$$

EXAMPLE: Let's assume (for simplicity) that the size of all types is 1. In this case, the amount of memory required to safely run $m0$ is:

$$\begin{aligned}\texttt{memRq}_{m0}(mc) &= \texttt{memEsc}(m0)(mc) + \texttt{MaxRegSize}_{m0}^{m0}(mc) \\ &\quad + max(\texttt{MaxRegSize}_{m0}^{m0.1.m1.5)}(mc) \\ &\quad + \texttt{MaxRegSize}_{m0}^{m0.1.m1.5.m2}(mc), \texttt{MaxRegSize}_{m0}^{m0.2.m2}(mc)) \\ &= 0 + 2(2mc) + max(2(\frac{1}{2}mc^2 + \frac{1}{2}mc) \\ &\quad + 1mc + 1mc, 2mc) \\ &= 4mc + max\{mc^2 + 3mc, 2mc\} \\ &= mc^2 + 7mc\end{aligned}$$

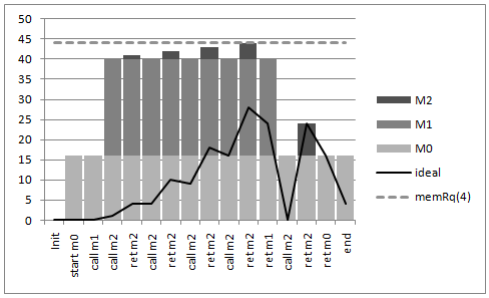

Figure 5: Computed memory requirements against actual memory consumption (# objects)

Fig. 5 shows a run of the example of Fig. 2 together with the evolution of the size of regions for $m0$, $m1$ and $m2$ and the computed approximation of the amount of memory required to safely run $m0$, ($memRq$). The prediction is accurate for the proposed region-based memory management since the expression exactly matches the actual value that an execution using m-regions will require to run. In this case the overhead in memory usage comes from the use of m-regions instead of using a more aggressive collection mechanism (represented by *ideal* in the figure). It is worth mentioning that this approach obtains safe bounds even for other memory reclaiming mechanisms ([5]). The intermediate region inference generation adds an additional level of over-approximation.

6. EVALUATION

While in [5] and [6] validation of memory requirements computation is performed on Java Grande and Jolden benchmarks, here the focus in the case studies is to show the capabilities and limitations of the current prototype in dealing with real-life applications.

Currently the tool is only able to quantify consumption performed on available source code since it requires annotations for program invariants (see future work). Although the tool may provide information segregated by types, expressions shown in the examples measure memory in terms of number of objects regardless their types and sizes. This was done for the sake of simplicity and comparison with actual execution (actual memory measured in bytes would be inaccurate due to the non-analyzable creation sites). Experiments source code and analysis results can be obtained from http://lafhis.dc.uba.ar/jconsume/examples. They were carried in a Intel P4, 3Ghz, 2GB RAM running Linux 2.6.8.

6.1 CDx benchmark

CDx (Collision Detector)[2] is an open source application

[1]Actually the invariant binds the parameters of all the sequence of methods that appears in the call chain.W are local variables appearing in the other methods in the call chain.

[2]`http://adam.lille.inria.fr/soleil/rcd/`

benchmark for real-time Java VMs. The CDx core (about 2K lines of code, 47 analyzable news) is a RTSJ compliant and implements a periodic task that performs an aircraft collision detection using (simulated) radar frames. The core of the application code is mainly made up of two classes (and many other helper classes as well):

- `PersistentDetectorScopeEntry` which synchronizes with the environment (i.e., the simulator) to store the produced radar frames.
- `TransientDetectorScopeEntry` where the actual collision detection algorithm is executed.

There is only one memory region used by the transient detector code. Almost all objects created by this piece of code are allocated there, except for a subset of them which are allocated in the persistent one.

The first goal in this experiment is to automatically compute the size required for the transient detector memory region. Unfortunately, fully automatic computation of this piece of code is not possible because limitation recursion free code. To overcome that problem, the recursive method `dfsVoxelHashRecurse` is replaced with a non-recursive bfs-based implementation. Then, we consider allocations performed by this method bounded by a parameter p denoting the maximum number of iterations required to process the voxel structure. We think this parameter is likely to depend on another application parameter, but there is no current tool support to link this complexity parameter to actual method parameters. Table 2, column 1, show the obtained region size which is a non-trivial expression in terms of this parameter p and simulator related parameter c (plane counter). Although this expression is based on introducing a fresh complexity parameter, we believe this is a step ahead compared to the approach followed in the original code where a very large constant is hard-coded as the required memory size. Besides, as we will see later on, this parameter do not hinder the possibility of using the computed memory requirement expressions to compare alternative region assignments.

The second goal is to use region inference to discover potential regions refining the original transient one. In fact, the tool obtains 3 new regions for methods `reduceCollisions`, `lookForCollisions` and `createMotions`. The tool computes region sizes and yields a new memory requirement expression which predicts a similar upper bound (see column 2). This is because no region is created within a loop. Thus, no gain is obtained.

Finally, we manually refine the memory regions assignment by including a new region for the temporary objects created during computation of the voxelHashing method. The new upper bound calculation shows a potential improvement of worst case memory footprint of this refined version (see column 3).

6.2 Banking case study

Using the prototype, we analyze semi-automatically a real-life, real-time data-base banking application, whose Java code was observed to spend a lot of time garbage-collecting. The application Java code is 3.4K lines long (600 comments), with around 360 `new` statements. The structure of the code is a nesting of calls to methods of the following (simplified) form:

```
class Work {
```

Table 2: Region sizes and memory requirements for executing method `run` on transient scope (in # of objects).

Region	Original	Inferred	Manually Refined
run	$\frac{5}{2}p.c^2+(\frac{21}{2}p+37)c-p+15$	8	8
reduceCollisions	N/A	$(11p+2)c+2$	$(4p+1)c+2$
lookForCollisions	N/A	$2p.c-p+3$	$2p.c-p+3$
createMotions	N/A	1	1
voxelHashing	N/A	N/A	$7p+1$
$\texttt{memRq}_{run}$	$13p.c+2c-p+11$	$13p.c+2c-p+10$	$6pc+6p+c+14$

```
...
  void exec(Object[] param) {
    Iterator it = select(param);
    if(<cond>) {
      while(it.hasNext()) {
        ...
      }
    }
    else {
      while(it.hasNext()) {
        Object[] filter = new Object[] { ... }
        Work w = new Work(filter);
        Object[] p = getParams(it.next());
        w.exec(p);
      }
    }
  }
}
```

Roughly speaking, `exec` first searches all records in the database matching `param` (e.g., SQL select), and then, it either performs some computations using the result, stopping the chain of calls, or, for each match, it creates a new worker which will perform a selection in another table based on a new set of parameters, and so on. In the full application code there are 6 different instances of `Work`-like classes, namely W-A to W-F (see Fig. 6) with a minimum of 10 and a maximum of 222 private instance fields each, and a maximum depth of 6 and breadth of 4 calls.

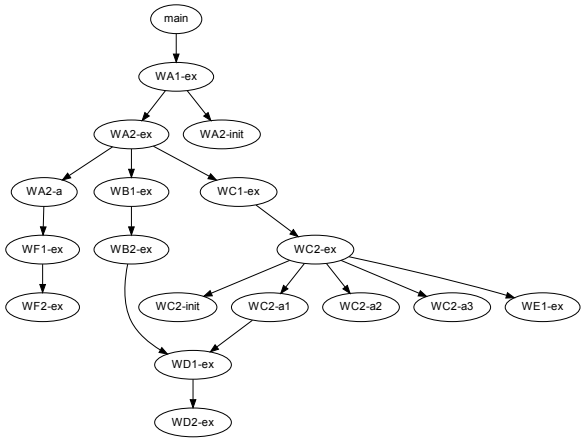

Figure 6: Simplified application call-graph showing its most relevant methods.

The original code was outside the scope of our current implementation of the quantitative-analysis application. Thus, to analyze it, we applied the following methodology. We run region synthesis on the original code. This served to identify the regions. In particular, it resulted that `exec` methods could be considered to be *waterproof*, that is, no object created within its lifetime will escape outside its scope. Even if it could be possible to have more regions, we decided to have one region per `exec` method. This yielded a total of 18 regions. Moreover, the analysis showed that no newly created objects were chained to `exec` parameters. That is, `param` was used as a placeholder for passing values down, while `filter` was used to pass values up. This information served to perform a (manual) slicing of the method parameters and further simplifications of the code, such as instance field elimination, without changing its behavior with respect to the number of created objects [3]. After this code transformation, we were able to run the prototype and to obtain in less than 10 minutes the polynomials for each of the 18 regions (about 5 minutes took `Daikon` to get the local invariants, 50 seconds for region synthesis and 75 seconds for quantitative analysis). Analysis results are shown in Table 3. `memalloc` is computed using the technique explained in §5.1. By applying region synthesis and the technique explained in §5.2 we computed the region sizes. Finally, using technique §5.3 we computed `MaxRegSize` then `memRq`. It is worth mentioning that in these cases the obtained polynomials, when instantiated, represented exactly the actual number of objects created by the application (considering only objects created by application methods the analysis can reach). Notice also the difference between total allocations and computed memory requirements using the synthesized memory regions. Using regions memory requirements are considerable lower than the number of created objects. For instance, for input size $m = 10$ 2311/57103, for $m = 20$ 7401/472103, for $m = 50$ 40671/10575103, etc.

By transforming the application into a functionally equivalent region-based one, we can get rid of garbage collecting an important amount of objects. In addition we obtain parametric certificates (1) of the requirements to safely run the whole application, and of (2) the space required to properly allocate the memory regions. We were able to measure consumption, but unfortunately, we could not evaluate the performance of the transformed code in comparison to the GC-based one because our Jikes region manager is still under development.

7. RELATED WORK

Most garbage collection algorithms are optimized to achieve high-throughput at the expense of occasional pauses that can block user threads for tens to hundreds of milliseconds. Real-time collectors have shown that sub-millisecond pause times can be achieved at the expense of a reduction in throughput [4]. However, they do not offer predictability in terms of space, and actually, make prediction of dynamic memory consumption even more difficult.

Our work integrates a series of techniques for region synthesis and static quantitative analysis. Although there has been a significant amount of work in escape analysis and region inference techniques (e.g.[26, 13, 8, 17, 25]) and some work in dynamic memory inference for imperative languages (e.g [1, 2, 10, 6, 5]), we did not find any integrated approach able to produce region-based code and memory requirements certificates (see [16] for detailed related work).

[3]Indeed, this simplification was done to ease `Daikon`'s task in finding local invariants. Another option would have been manual tuning of the set of inductive variables.

Table 3: Max. region sizes and application's memory requirements (all in objects). Parameter m:elements to process. #CS: Creations sites captured by the region.

Method	#CS	MRS	Method	#CS	MRS
WA1-ex	90	$3m^2 + 6m + 81$	WA2-init	3	3
WA2-ex	23	$5m + 18$	WA2-a	6	6
WB1-ex	23	$m + 22$	WC2-init	10	10
WB2-ex	7	$m + 6$	WC2-a2	4	4
WC1-ex	57	57	WC2-a3	5	5
WC2-ex	88	$12m^2 + 46m + 40$	WD2-ex	2	2
WC2-a1	8	$m + 7$	WE1-ex	2	2
WD1-ex	16	$m + 15$	WF2-ex	3	3
WF1-ex	43	$5m + 38$	main	1	1
$\texttt{memalloc}_{main}(m) = m^4 + 32m^3 + 130m^2 + 200m + 103$					
$\texttt{memRq}_{main}(m) = 15m^2 + 59m + 221$					

Methods based on type systems typically require aliasing and size annotations. For instance, the method proposed in [9] relies on a type system and 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 (we infer non-linear expressions). Their type system allows aliasing and object deallocation (dispose) annotations. [10] extend this technique to infer upper bounds of heap and stack usage provided they can be expressed as Presburger's formulas, still requiring explicit memory management.

As a related inference technique, Albert et al. [1] propose a parametric cost analysis for sequential Java code. The code is translated to a recursive representation. Then, they infer *size relations* which are similar to linear invariants. Using the size relation, and the recursive program representation they compute *cost relations* which are set of recurrent equation in term of input parameters. Applied to memory consumption their bounds are not limited to polynomials. Recently, [2] proposes an extension where object deallocation is handled, similarly to ours, by approximating objects lifetime using escape analysis. In this setting the use of escape analysis is not for region synthesis and program transformation, it is just used as a mean to approximate memory requirements.

8. LIMITATIONS AND FUTURE WORK

We have presented our approach aiming at assisting the generation of region-based code and the automatic synthesis of parametric certificates of dynamic memory consumption. We think the tool can be used to assist developers annotating regions sizes for RTSJ or another approaches such us PERC PICO [22]. We have developed a prototype tool that covers the complete chain of techniques and allows us to evaluate experimentally the current strengths and weakness of the approach.

The current approach has still several limitations that we would like to address in the near future. As already mentioned previously, the approach does not support recursive method calls. This could be in principle acceptable for embedded applications but it may become an obstacle to when thinking in a broader spectrum of applications. Allocations made by native methods or internal allocations made by the runtime system (virtual machine) are not considered by the

techniques.

The analysis is better suited to deal with programs that operate with arrays or simple collections and integer variables but it may be difficult to accurately deal with allocations depending on values inside complex data structures.

The main difficulty that affect scalability is the need of precise program invariants over minimal set of inductive variables. This in general requires human intervention which makes analysis of real-world applications a time consuming task if programmers do not use assertions in code.

To tackle many of the detected limitations we are currently developing a new compositional analysis, based of computing method summaries. This enables local reasoning of memory consumption and annotations for non-analyzable methods. We think recursion will also naturally fit in this approach. Compositional analysis also is better suited for integration with type-checking based used to deal with more complex more complex or recursive data structures. The approach would use type-checked annotations for data-container classes (like the ones provided by standard libraries) and our inference approach for loop-intensive applications on top of those verified libraries.

Regarding region-based support we plan improve our implementation in `Jikes` to leverage on more quantitative information we are able to generate about regions-sizes and peak consumption. Instead of only using our prediction to provide sizes at region creation time, we would also use our prediction about maximum regions sizes for preallocating set of regions that are created more frequently.

Acknowledgments:

The authors want to thank the reviewers for their valuable comments and Jan Vitek and his group for the insightful observations and suggestions for paper improvement. This work has been partially funded by CONICET, ECOS A06E02 "Japiqay", ANPCyT grants PICT 32440 and UBACyTX021.

9. REFERENCES

[1] E. Albert, P. Arenas, S. Genaim, G. Puebla, and D. Zanardini. Cost analysis of java bytecode. *ESOP*, volume 4421 of *LNCS*, pages 157–172. Springer, 2007.

[2] E. Albert, S. Genaim, and M. Gómez-Zamalloa. Live heap space analysis for languages with garbage collection. In *ISMM*, pages 129–138, 2009.

[3] B. Alpern, S. Augart, S.M. Blackburn, M. Butrico, A. Cocchi, P. Cheng, J. Dolby, S. Fink, D. Grove, M. Hind, et al. The Jikes research virtual machine project: building an open-source research community. *IBM Systems Journal*, 44(2):399–417, 2005.

[4] D. Bacon, P. Cheng, and V. T. Rajan. Controlling fragmentation and space consumption in the metronome, a real-time garbage collector for java. In *LCTES '03*, pages 81–92.

[5] V. Braberman, F. Fernández, D. Garbervetsky, and S. Yovine. Parametric prediction of heap memory requirements. In *ISMM '08*, pages 141–150.

[6] V. Braberman, D. Garbervetsky, and S. Yovine. A static analysis for synthesizing parametric specifications of dynamic memory consumption. *Journal of Object Technology*, 5(5):31–58, 2006.

[7] 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.

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

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

[10] W.N. Chin, H.H. Nguyen, C. Popeea, and S. Qin. Analysing Memory Resource Bounds for Low-Level Programs. In *ISMM '08)*,pages 151–160.

[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] Ph. Clauss and I. Tchoupaeva. A symbolic approach to bernstein expansion for program analysis and optimization. In *CC 04*, *LNCS*, pages 120–133.

[13] M. Deters and R. K. Cytron. Automated discovery of scoped memory regions for real-time java. In *ISMM 02*, pages 25–35, 2002.

[14] M. Ernst, J. Perkins, P. Guo, S. McCamant, C. Pacheco, M. Tschantz, and Ch. Xiao. The Daikon system for dynamic detection of likely invariants. *Science of Computer Programming*, 2007.

[15] A. Ferrari, D. Garbervetsky, V. Braberman, P. Listingart, and S. Yovine. Jscoper: Eclipse support for research on scoping and instrumentation for real time java applications. In *eTx '05*, pages 50–54.

[16] D. Garbervetsky. *Parametric specification of dymamic memory utilization*. PhD thesis, DC, FCEyN, UBA, November 2007.

[17] D. Garbervetsky, Ch. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in Java. *RV 04*, ENTCS, Spain.

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

[19] James Gosling and Greg Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., 2000.

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

[21] F. Nielson, H. Nielson, and Ch. Hankin. *Principles of Program Analysis*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1999.

[22] K. Nilsen. Improving abstraction, encapsulation, and performance within mixed-mode real-time Java applications. In *JTRES 2007*, pages 13–22.

[23] F. Pizlo, J. Fox, D. Holmes, and J. Vitek. Real-time Java scoped memory: design patterns and semantics. In *ISORC '04*, Austria.

[24] G. Salagnac, Ch. Rippert, and S. Yovine. Semi-automatic region-based memory management for real-time java embedded systems. In *RTCSA'07*, 2007.

[25] G. Salagnac, S. Yovine, and D. Garbervetsky. Fast escape analysis for region-based memory management. *ENTCS*, 131:99–110, 2005.

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

[27] Fridtjof Siebert. Hard real-time garbage-collection in the jamaica virtual machine. *rtcsa*, 00:96, 1999.

[28] M. Tofte and J.P. Talpin. Region-based memory management. *Information and Computation*, 1997.

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