JScoper: Eclipse support for Research on Scoping and Instrumentation for Real Time Java Applications [*]

Andrés Ferrari, Diego Garbervetsky, Victor Braberman, Pablo Listingart, Sergio Yovine
School of Computer Sciences. Universidad de Buenos Aires, Argentina. Verimag, France.
aferrari@dc.uba.ar, diegog@dc.uba.ar, vbraber@dc.uba.ar,plistingart@gmail.com, yovine@imag.fr

ABSTRACT

We present JScoper, an Eclipse plug-in which will help developers, researchers and students, to generate, understand, and manipulate memory regions in scoped-memory management setting. The main goal of the plug-in is to provide a tool that will transparently assist the translation of Java applications into Real-time Specification for Java (RTSJ) compliant applications. More accurately, its purpose is to enable automatic and semi-automatic ways to translate heap-based Java programs into scope-based ones, by leveraging GUI features for navigation, specification and debugging.

Keywords

Eclipse plug-in, memory management, Real-time Java

1. INTRODUCTION

Current trends in the embedded and real-time software industry are leading practitioners towards the use of object-oriented programming languages such as Java. From a software engineering perspective, one of the most attractive issues in object-oriented design is the encapsulation of abstractions into objects that communicate through clearly defined interfaces. Because programmer-controlled memory management hinders modularity, object-oriented languages like Java provide built-in garbage collection, i.e. the automatic reclaiming of heap-allocated storage after its last use by a program.

However, automatic memory management is not used in real-time embedded systems. The main reason for this is that the execution time of software with dynamic memory reclaiming is extremely difficult to predict. Therefore, in current industrial practices the use of garbage collection in real-time applications is simply forbidden. The typical alternative approach is to have programs allocate all memory during their initialization phase and free it upon termination. This leads to very inefficient memory use, usually resulting in over-dimensioning physical memory requirements at an unnecessary additional cost.

A automatic memory management techniques that meet real-time requirements would clearly have a huge impact on the design, implementation, and analysis of embedded software. These techniques would prevent programming errors produced by hazardous memory handling, which are both hard to find and to correct. As a result, they would drastically reduce implementation and validation costs while considerably improving software quality.

In order to overcome the drawbacks of current garbage collection algorithms, the Real-Time Specification for Java (RTSJ)[2] proposes the use of application-level memory management, based on the concept of "scoped memory", for which an appropriate API is specified. Scoped-memory management relies on the idea of allocating objects in regions associated with the lifetime of a computation unit (method or thread). Regions are deallocated when the corresponding computational units finish their execution [9, 6, 2, 5]. Unfortunately, the task of determining object scopes is left to the programmer.

Some techniques have been proposed to address this problem by automatically mapping sets of objects with regions[3, 5]. These techniques typically use Pointer and Escape Analysis [8, 4, 1] to conservatively approximate object lifetimes. Informally, an object escapes a method when its lifetime is longer than the method's lifetime, so it cannot be collected when the method finishes its execution. In contrast, an object is captured by the method when it can be safely collected at the end of the method's execution.

Our main goal is to provide developers with a tool that will assist the translation of Java applications into Java Real-time compliant applications. More accurately, the idea is to enable translation of heap-based Java programs into scoped-based ones, by leveraging GUI features for navigation, specification, translation, fine-tuning and debugging.

2. SCOPED MEMORY MANAGEMENT

The aim of the Real-Time Specification for Java (RTSJ) [2] is to enable the development of real-time applications using Java. One of its most remarkable characteristics is a new memory hierarchy which incorporates several kinds of memory models that do not use garbage collection. It proposes multiple kinds of memory models: Heap memory (garbage collected), Immortal memory and Scoped memory.

Both Immortal and Scoped memory do not use garbage collection. Objects allocated in Immortal memory are never collected and live throughout program lifetime. Scoped-memory management is based on the idea of allocating ob-

[*]IBM Eclipse Innovation Grants 2004.

jects in **regions** associated with the lifetime of a runnable object. 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; $o2$ is in the Heap; $o2$ is in Immortal (or static) memory. An object $o1$ cannot point to an object $o2$ in region r if: $o1$ is in the heap; $o1$ is in immortal memory; r is not active at some point during $o1$'s lifetime.

	Heap	Inmortal	Scoped
Heap	Yes	Yes	No
Inmortal	Yes	Yes	No
Scoped	Yes	Yes	if active

Table 1: Scoped-memory reference rules.

At runtime, region activity is related to the execution of computational units (e.g., methods or threads). 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. In a multi-threaded program, where regions are associated with threads and methods, there is a region tree whose branches are related to each execution thread.

In order to perform scoped-memory management at program level, an API is proposed which differs from the RTSJ one, described in [2], in three main points. First, in the proposed API memory scopes are not bound to runnable objects. In this point, this API is closer to the RC library [6]. Second, the API does not specify the region where an object will be allocated, but rather a set of regions corresponding to methods in a prefix of the corresponding call stack. The actual region where the object will be allocated at runtime is left out to the implementation. To determine in which region an object will be allocated we use a registering mechanism. Basically, when regions are created, they are informed about the set of creation sites (*new* statements) it will allocate. When object instantiation is requested, the API allocates the object in the last region the creation site was registered in. Finally, there is no Immortal memory; instead, it is simulated by a "main" region with a global scope. The API is shown in Table 2.

`enter(r,lCSs)`	push r into the region stack and register the creation sites it will allocate
`exit()`	collect the objects in top region
`newInstance(cs,c)`	create an object identified by the creation site cs of class c
`newAInstance(cs,c,n)`	same but for arrays of dimension n

Table 2: Scoped-memory API.

3. ECLIPSE PLUG-IN: JSCOPER

The Eclipse Java Development Toolkit (JDT) is one of the most popular and feature rich platforms currently available to Java developers. Because Eclipse is not only an IDE but an extensible plug-in platform, it is the ideal framework to use for the development of tools aimed at transforming Java code. Currently there are few tools that can be used to assist in the conversion of standard Java code to scoped-memory code. An Eclipse plug-in called *JScoper* that fullfils this purpose is presented in this paper. This is a tool that can be used to support both automatic and semi-automatic translation of heap-based Java programs into scope-based ones. Although the resulting programs are not fully compliant with RTJS (this will be supported in the future), they also implement a scope-based memory management mechanism which replaces the garbage collector from the Java Virtual Machine [5].

JScoper allows the user to visualize, debug and control the transformation process. Its GUI facilities provide a user-friendly way of gaining insight into the underlying concepts of controlled memory management.

3.1 Usage and Features

JScoper makes use of three main windows: the `Callgraph Browser`, the `Scoped-Memory Java Editor`, and the standard `Java Editor` provided with the Eclipse Java Development Toolkit. It also features additional views that provide alternative representations of the callgraph and memory regions.

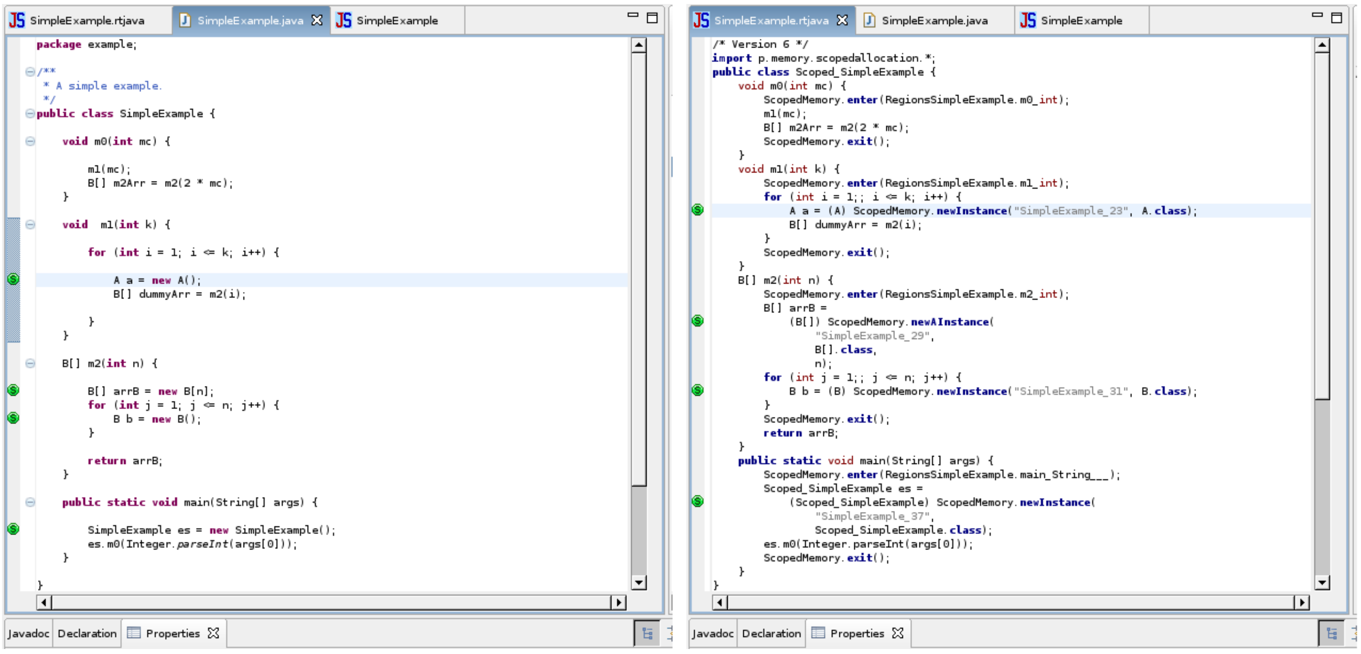
The *CallGraph Browser* is used for the visualization of the code callgraph and creation sites corresponding to dynamic memory allocation statements. It also has some editing capabilities: the manual creation of memory regions and the movement of creation sites between different regions. These editing features are meant to allow for manual adjustment of the automated output of the tool.

The *Scoped-Memory Java Editor* is a source code editor with syntax highlighting support for scoped-memory Java code, as well as special marker icons which act as hyperlinks between the different plug-in windows. These markers will be discussed later.

The *Java Editor* is the standard editor provided with the Eclipse JDT, with additional support for special marker icons analogous to those of the Scoped-Memory Java Editor.

During a normal usage workflow, the user will start from regular Java source code, use the integrated tools to identify the creation sites, perform escape analysis [4] (an optional step) and generate the callgraph, and then examine the resulting graph in the Callgraph Browser window. Memory region and creation site adjustments are possible at this stage. The user may also switch between the three editors (Callgraph, Instrumented and standard Java), using special marker icons which link related memory allocation sites. The final output of the plug-in will be stored as a series of XML files describing memory regions, creation sites and callgraph of the source code. These files are described with more depth in the following section, "Design and Implementation". The workflow consists of the following steps:

1. Start from Java source: this is the program the developer originally coded, with no concern for real-time issues. Positioned in the package explorer of the Eclipse Java view, the user must select the appropriate options provided by JScoper in order to analyze the code and memory regions (optional) and generate the callgraph. This will create a series of XML files corresponding to the callgraph, memory regions and creation sites, the `rtjava` instrumented code file and a `jscoper` project file which links all the previous files together.
2. Output visualization: the user can now examine the result of the automated code analysis and instrumen-



```
package example;

/**
 * A simple example.
 */
public class SimpleExample {

    void m0(int mc) {

        m1(mc);
        B[] m2Arr = m2(2 * mc);
    }

    void  m1(int k) {

        for (int i = 1; i <= k; i++) {

            A a = new A();
            B[] dummyArr = m2(i);

        }
    }

    B[] m2(int n) {

        B[] arrB = new B[n];
        for (int j = 1; j <= n; j++) {
            B b = new B();
        }

        return arrB;
    }

    public static void main(String[] args) {

        SimpleExample es = new SimpleExample();
        es.m0(Integer.parseInt(args[0]));
    }

}
```

```
/* Version 6 */
import p.memory.scopedallocation.*;
public class Scoped_SimpleExample {
    void m0(int mc) {
        ScopedMemory.enter(RegionsSimpleExample.m0_int);
        m1(mc);
        B[] m2Arr = m2(2 * mc);
        ScopedMemory.exit();
    }
    void m1(int k) {
        ScopedMemory.enter(RegionsSimpleExample.m1_int);
        for (int i = 1;; i <= k; i++) {
            A a = (A) ScopedMemory.newInstance("SimpleExample_23", A.class);
            B[] dummyArr = m2(i);
        }
        ScopedMemory.exit();
    }
    B[] m2(int n) {
        ScopedMemory.enter(RegionsSimpleExample.m2_int);
        B[] arrB =
            (B[]) ScopedMemory.newAInstance(
                "SimpleExample_29",
                B[].class,
                n);
        for (int j = 1;; j <= n; j++) {
            B b = (B) ScopedMemory.newInstance("SimpleExample_31", B.class);
        }
        ScopedMemory.exit();
        return arrB;
    }
    public static void main(String[] args) {
        ScopedMemory.enter(RegionsSimpleExample.main_String__);
        Scoped_SimpleExample es =
            (Scoped_SimpleExample) ScopedMemory.newInstance(
                "SimpleExample_37",
                Scoped_SimpleExample.class);
        es.m0(Integer.parseInt(args[0]));
        ScopedMemory.exit();
    }
}
```

Figure 1: A side by side view of the two code editors. Left: the standard Java Editor. Right: the Scoped-Memory Java Editor.

tation. The Scoped-Memory Java Editor (figure 1, right) is used to browse the instrumented code, which is a file with extension `rtjava`. Instrumented Java files contain an extension of Java code with special scoped-memory related statements. This editor can be used to switch to the relevant sections in the original source code, for comparison purposes. In order to allow this, there are special icons called markers that connect dynamic memory allocation statements in the original Java code with the corresponding statements in the instrumented code. It also links the `java` and `rtjava` files with the callgraph. The user is able to inspect related locations in the original source code, the instrumented code and the callgraph.

The code callgraph is represented visually in a directed graph form (figure 2). Nodes represent Java methods and show their corresponding creation sites (dynamic memory allocation statements, like *new*). When a Java method calls another, an arrow with a label stating the line number is drawn to connect the corresponding two nodes in the graph. Each creation site lists the memory regions that capture it. Several filters that can reduce visual clutter and are useful to inspect the code flow are provided: for example, it is possible to trace a path from the root node (which represents the initial caller method) to any selected node in the graph, focus on the subgraph that spans from any given node or hide the region information so that only the code flow is shown. In addition, there are two side views that can also be inspected: a hierarchical tree view of the callgraph and a tree view of the current memory regions. Image snapshots of the callgraph may be exported at any time.

3. Manual adjustments: both the generated memory regions and the creation sites location within those regions may be manually adjusted. If the automatically generated regions are not satisfactory (for example, because they are too conservative), they can be deleted, modified or added at will using a region management window which can be accessed both from the toolbar and from a context menu. This manager also allows the reassigning of creation sites to different regions (figure 3).

All intermediate files are persisted to disk storage and can be inspected at any time with a text editor. JScoper can be used to explicitly write the current state of region/creation site mappings at any time.

3.2 Design and Implementation

JScoper was developed for the 3.x series of the Eclipse platform. Currently there is no support for versions 2.x or earlier. It was developed and tested in Linux and Windows XP. It has not been tested (yet) on other operating systems, but it should work on any platform supported by Eclipse and Java 1.4.x.

JScoper integrates 4 distinct modules which roughly correspond to the editors described in the previous section, "Usage and Features": the Callgraph Browser, the Scoped-Memory Java Editor, the standard Java Editor and the Backend (which is actually a collection of different tools itself). This paper focuses on the frontend of the plug-in.

- The Callgraph Browser handles the visual representation of the program callgraph and allows the manual editing of memory regions and creation sites. This module uses an add-on for Eclipse called GEF, the Graphical Editor Framework [1], which is used to implement the graphical editor following the Model-View-Controller pattern.

[1]See the homepage at http://www.eclipse.org/gef/

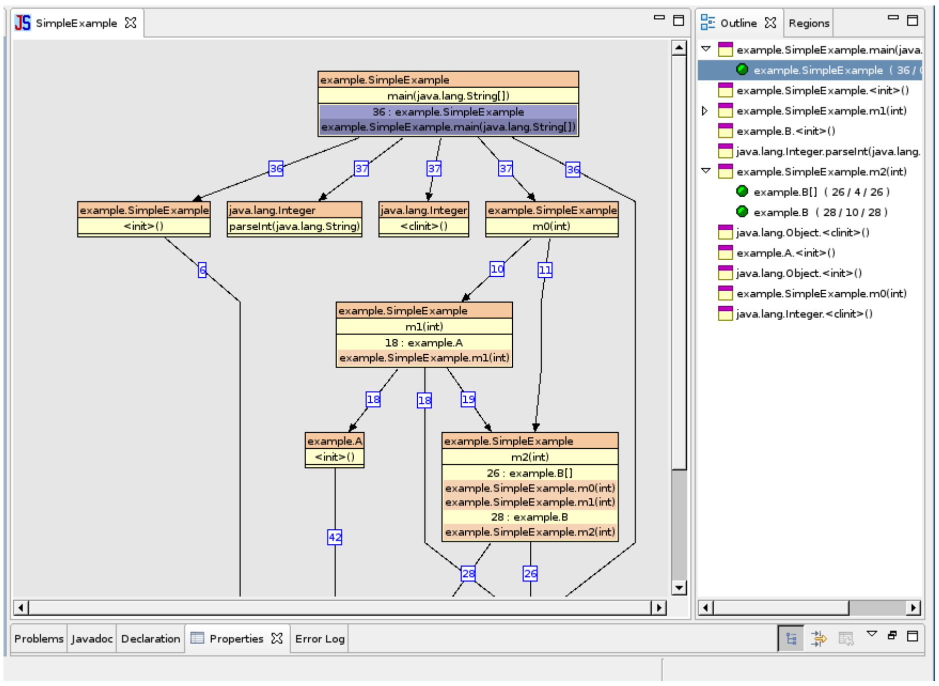

SimpleExample
example.SimpleExample main(java.lang.String[])
36 : example.SimpleExample
example.SimpleExample.main(java.lang.String[])
example.SimpleExample <init>()
java.lang.Integer parseInt(java.lang.String)
java.lang.Integer <clinit>()
example.SimpleExample m0(int)
example.SimpleExample m1(int)
18 : example.A
example.SimpleExample.m1(int)
example.A <init>()
example.SimpleExample m2(int)
26 : example.B[]
example.SimpleExample.m0(int)
example.SimpleExample.m1(int)
28 : example.B
example.SimpleExample.m2(int)
Outline
Regions
Problems
Javadoc
Declaration
Properties
Error Log

Figure 2: The callgraph browser window. The view on the right shows a tree outline of the callgraph.

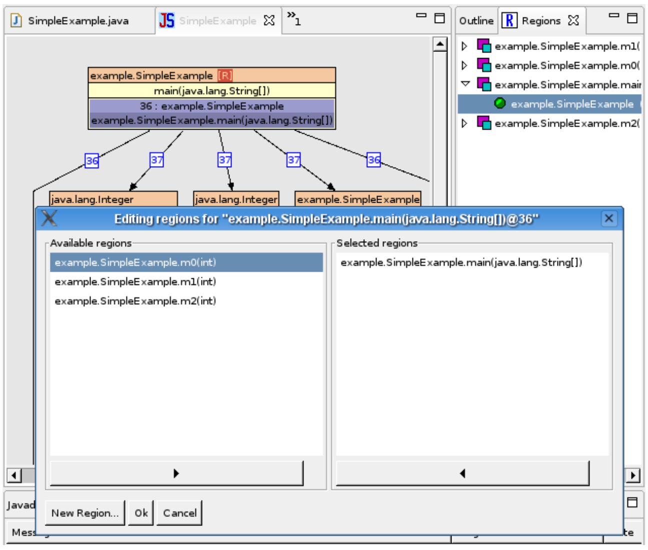

SimpleExample.java
SimpleExample
Outline
Regions
Editing regions for "example.SimpleExample.main(java.lang.String[])@36"
Available regions
example.SimpleExample.m0(int)
example.SimpleExample.m1(int)
example.SimpleExample.m2(int)
Selected regions
example.SimpleExample.main(java.lang.String[])
New Region...
Ok
Cancel

Figure 3: The Region Manager.

- The Scoped-Memory Java Editor is used to inspect and edit the instrumented source code. Special Eclipse markers allow switching to and from creation sites in the regular Java source code and also to the corresponding nodes in the Callgraph Browser window.
- The Java Editor mimics the behavior of the standard source editor included with the Eclipse platform, and adds support for the special markers mentioned above.
- The Backend consists of a collection of tools that actually perform the code analysis, including a code instrumentator [5], a callgraph generator based on *Soot* [7], an escape analysis and region inferrer [4] and a creation sites finder.

A sketch of the plug-in model is shown in figure 4. The original `Code Model` is the basis for establishing derived models (and their corresponding views), namely, `Call Graphs` and `Creation Sites`. The `Point of View` defines the abstraction parameters used to obtain call graphs and creation sites (e.g., root method for the analysis, whether or not to include standard Java API creation sites, etc.). The `Region Model` is a mapping from creation sites to sets of regions, and it is used as the input for the instrumentation procedure that generates a `Scoped Code Model`. The `Object Lifetime Model` is an escape analysis [4] representation and holds the relationship between creation sites, the regions that contain them, and their paths within the call graph. This model can be used to either automatically synthesize a Region Model, and in the future it will also be used to validate a manually created one. Each of these models has a corresponding view in the plug-in, with the exception of the Point of View (which is currently unimplemented) and the Object Lifetime, whose graphical visualization, while currently unavailable, will be a call graph coloring.

The interface between the plug-in modules comprises several XML files. Assuming the original Java source file is named `MyClass.java`, then the XML files are:

- The callgraph file, `MyClassCallGraph.xml`. This is an XML that contains graph information in the form of nodes (class methods) with items (creation sites) linked to other nodes (method calls). Each node is identified by a classname and a fully qualified method name, and it has a list of all the "children" or nodes it is linked to. Each child node represents a method that is called from the parent method at the line number specified by attribute *line*. Any arbitrary callgraph may be represented and cycles are possible.

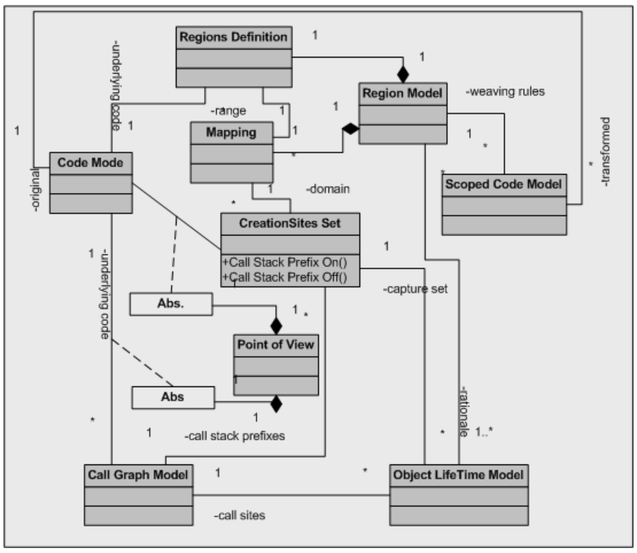


Figure 4: Modules of JScoper

- The creation sites file, `MyClassCreationSites.xml`. This is an XML that lists the line numbers of dynamic memory allocation statements within the Java source code. A simplified version looks like this:

```
<CreationSites id="example.SimpleExample">
    <CreationSite method="m0" line="26"/>
    <CreationSite method="m0" line="27"/>
    <CreationSite method="m1" line="29"/>
</CreationSites>
```

  Method *m0* in the source for class *example.SimpleExample* has creation sites at lines 26 and 27, while method *m0* has its sites at 29 and 32.
- The memory regions file, `MyClassRegions.xml` (optional). This is an XML that stores the assignment of creation sites to scoped memory regions. There may be more than one creation site within any given region. This file is optional; if it is not present when the user tries to visualize a callgraph, JScoper will simply generate default regions named after the corresponding method for each orphan site. A simplified version of this file has the following outline:

```
<Regions>
   <Region id="R1" scope="SimpleExample.m0"
           lineFrom="10" lineTo="28">
      <CreationSite method="m0" line="26" instancesExp="x"/>
      <CreationSite method="m0" line="27" instancesExp="x^2"/>
   </Region>
   <Region id="R2" scope="SimpleExample.m1"
           lineFrom="29" lineTo="50">
      <CreationSite method="m1" line="" instancesExp="2x"/>
      <CreationSite method="m1" line="" instancesExp="x"/>
   </Region>
</Regions>
```

  A region description states its scope (essentially, the classname and method where it is located), the line numbers it spans and the creation sites which it contains. Currently, regions cannot cross method or class boundaries but there may be two or more regions within a given Java method.
- The Java to Scoped-Memory Java file, `MyClassCSR.xml`. This is an XML mapping that links the location of creation sites (i.e. dynamic memory allocation statements such as *new*) to the corresponding statement in the instrumented code.

There are two additional files which are not XMLs and have special meanings:

- The instrumented code, `MyClass.rtjava`.
- The JScoper project file, `MyClass.jscoper`, which links all the previous files together.

4. CONCLUSIONS AND FUTURE WORK

JScoper is an Eclipse plug-in that assists the automatic translation of standard Java code to a RTJS-like code. It provides a graphical call graph browser that helps ease program understanding, supports the generation and edition of memory regions, automatic code generation and code visualization.

Future work plans include the implementation of debugging facilities such as runtime browsing of active regions, visualization of object-lifetimes, region-sizes and scoping-rules violations. It is also planned to include full RTSJ compatibility (automatic instrumentation and edition) and support for the automatic generation of memory size annotations [10].

5. ABOUT THE AUTHORS

Victor Braberman: Ph.D. in Computer Science and Associate Professor in C.S. Department of UBA (Argentina) working in modeling and verification of real-time and distributed systems. Diego Garbervetsky: Ph.D candidate in Computer Science and Lecturer working in Programm Analysis for Embedded Systems. Andrés Ferrari and Pablo Listingart: M.Sc in Computer Science students and main developers of JScoper. Sergio Yovine: Ph.D. in Computer Science and full time researcher (CNRS) working in modeling and verification of real-time and distributed systems.

6. REFERENCES

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

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

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

[4] S. Yovine G. Salagnac and D. Garbervetsky. Fast escape analysis for region based memory management.

[5] D. Garbervetsky, C. Nakhli, S. Yovine, and H. Zorgati. Program instrumentation and run-time analysis of scoped memory in java. *In RV 04, ETAPS 2004, Barcelona, Spain*, April 2004.

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

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

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

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

[10] D. Garbervetsky V. Braberman and S. Yovine. Synthesizing parametric specifications of dynamic memory utilization in object oriented programs. In *FTfJP 2005. Glasgow, Scotland*, July 2005.