# Towards an Analysis of Race Carrier Conditions in Real-time Java [*]

M. T. Higuera-Toledano
DACYA, Facultad de Informática
Universidad Complutense de Madrid
Ciudad Universitaria, Madrid 28040, Spain
mthiguer@dacya.ucm.es

## Abstract

*The RTSJ memory model propose a mechanism based on a scope three containing all region-stacks in the system and a reference-counter collector. In order to avoid reference cycles among regions on the region-stack, RTSJ defines the single parent rule. The given algorithms to maintain the region-stack structure are not compliant with the defined parentage relation. More over, the suggested algorithms to maintain the single parent rule introduces race carrier conditions on the application behaviour. This paper proposes alternative approaches in order to avoid this problem.*

**keywords:** Java, Real-Time Systems, Memory Regions, Garbage Collection.

## 1. Introduction

The Java environment provides attributes that make it a powerful platform to develop embedded real-time applications (e.g., architecture-neutral, multithreaded, dynamic loading, and garbage collection). However, it does not provide predictability facilities nor bounded resource usage, which are needed for the above applications. The *N*ational Institute of Standards and Technology (NIST), has produced a basic requirements document for a standard real--time Java API extension. The NIST document identifies seven areas for modification; one of them is the memory management. A solution that complies with this document is the *R*eal-time Specification for Java (RTSJ) [10]. One of the main advantages of using high-level languages is that the programmer must not deal with many low-level resource allocation issues. Unfortunately, for embedded real-time systems there is a conflict. The memory management is one of the major issues that need research when considering the extension of Java for real-time.

The run-time implementation of Java provides two basic data structures to treat the memory generated by the program: the *stack* and the *heap*. Only primitive types (e.g., `int`, `boolean`, and reference variables) are allocated in the runtime stack. Objects created from class definitions are allocated within the heap, and are collected by an implicit *Garbage Collector* (GC), which runs as part of the JVM. Implicit garbage collection has always been recognized as a beneficial support from the standpoint of promoting the development of robust programs. However, this comes along with overhead regarding both execution time and memory consumption, which makes (implicit) garbage collection poorly suited for small-sized embedded real-time systems. Although there has been extensive research work in the area of making garbage collection compliant with real-time requirements, there are still problems to use this technique in time-critical systems. An alternative to the classical GC is to use region-based memory allocation (e.g., [3]), which enables grouping related objects within a region. This technique, commonly called Memory Regions (MRs) is used explicitly in the program. This is an intermediate solution between explicit memory allocation/deallocation (e.g., `malloc()` and `free()` in C) and garbage collection.

RTSJ [10], which use in mission critical systems is currently being evaluated in a number of projects such as [3], combines MRs within which objects are not collected, and a GC within the heap. In real-time systems, the GC must ensure memory availability for new objects without interfering with real-time constraints. In this sense, current Java collectors present a problem because threads may be blocked while the GC is executing. As an example, consider a time-critical real-time thread that executes periodically and a non-critical real-time thread having lower priority than the critical one. While the non-critical one is running the GC (e.g., as consequence of an object allocation), it is not safe for the critical real-time thread to execute, even if it does not require any new memory. Then, the critical real-time thread must wait to preempt the non-critical real-time thread until the GC has finished.

[*]Ministry of Education of Spain (CICYT); Grant Number TIC2003–01321

Then, the only way to offer real-time guarantees is by turning off the GC during the execution of critical real-time threads. In order to do that, critical real-time threads only allocate objects outside the heap and cannot reference objects within the heap. Then, RTSJ introduces immortal and scoped MRs, which are outside the Java heap and objects within they are not subject to garbage collection. In this paper we review the RTSJ memory management semantic and requirements for nested scoped memory regions.

The remainder of the paper is organized as follows: after a brief discussion of related work (Section 2), we outline de nition and use of the RTSJ memory model (Section 3). We study race carrier conditions on programs using scope MRs in RTSJ (Section 4). Then, we give the main guidelines of two alternative solutions to solve this problem, considering and analyzing compliance with RTSJ and deterministic behaviour (Section 5). Finally a summary of our contribution conclude this paper (Section 6).

## 2. Related Work

The RTSJ reference implementation and the guidelines given by some RTSJ members (e.g., [8] [2]) are based on a dynamic parentage relation of scoped memory regions. The main contribution of this paper is to show that this relation presents several problems:

*(i)* It results in an unfamiliar programming model, because the parentage relation is not trivial: there are orphans regions and the parent of a region can change along its life.

*(ii)* It requires checks for all ancestors of a scoped region each time a real-time thread changes its allocation context (e.g., it is created/destroyed or enters/exits a region).

*(iii)* It introduces a high overhead: the algorithms checking for the scoped region ancestors are stack-based, having a time complexity of *O(n)*.

*(iv)* It is not time-predictable, which is contradictory with real-time systems: the introduced overhead must be bounded because the size of the stack is only known at run time.

*(v)* It presents race carrier conditions which makes non-deterministic the RTSJ application behaviour.

A study of the behaviour of the RTSJ parentage relation of scoped regions, and a rst approach in order to solve some of these problems (e.g., to avoid checks for all ancestors when a real-time thread enters/exits a scoped region) has been presented in [5].

Given that to collect scoped regions the used mechanism is different from the GC within the heap and local variables within the stack, RTSJ impose some restrictions on assignments. Otherwise dangling references (i.e., references to objects that have been collected when reclaiming the scoped region) between the different types of memory may occur. In order to maintain the safety of applications, RTSJ requires that the imposed restrictions be enforced.

A technique to subtype test in Java have been presented in [4]. This technique has been extended to perform memory access checks in constant-time. Given that one of the general requirements of RTSJ is that the existing Java baseline compilers must be used to compile the RTSJ programs, the assignment rules must be enforced by the JVM. That is at run-time. Static analysis on the compiler can be used in order to detect illegal assignment. However, given the dynamic nature of the Java language, and that there is not a special representation in the Java bytecode, only static analysis is not enough. Then, the solution to detect illegal references requires write barriers.

The most common approach to implement write barriers is by in-line code, consisting in generating the instructions executing barrier events for every load/store operation. Beebe and Rinard use this approach [11], and their implementation uses ve runtime heap checks to ensure that a critical real-time thread does not manipulate heap references. Alternatively, the solution proposed in [6] instruments the bytecode interpreter, avoiding space problems, but this still requires a complementary solution to handle native code.

The success of RTSJ depends on the possibility to offer an ef cient implementation of the assignments restrictions. The use of hardware support for write barriers has been studied in [7], where an existing microprocessor architecture has been used in order to improve the performance of checks for illegal references. The solution proposed in [5] makes the scoped ancestor tree static and it is based on the display structure used to check illegal assignments in [1]. Hence, the improvement of one of solution proposed in this paper come along name-based checking for illegal references, which is ef cient and time-predictable.

## 3. The RTSJ Memory Model

*Region-based memory allocation* (e.g., [3]) enables grouping related objects within a region, which are used explicitly on the program code. The RTSJ speci cation [10] [9] supports the region paradigm through three kinds of memory regions (see Figure 1): (*i*) immortal memory that contain objects whose life ends only when the JVM terminates; (*ii*) (nested) scoped memory, that enables grouping objects having well-de ned lifetimes; and (*iii*) the conventional heap.

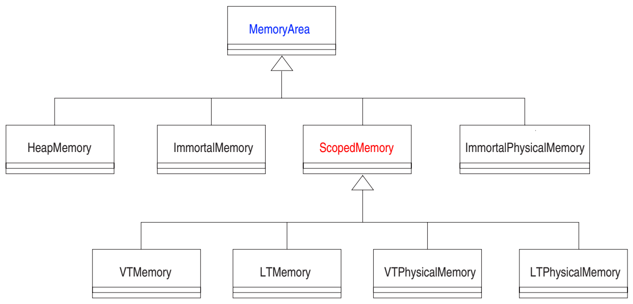

**Figure 1. The `MemoryArea` hierarchy in RTSJ.**

There is only one object instance of the heap and the immortal region in the system, which are resources shared among all threads in the system and whose reference is given by calling the `instance()` method. In contrast, for scoped and immortal physical regions several instances can be created by the application. An application can allocate memory into the system heap, the immortal system memory region, several scoped memory regions, and several immortal regions associated with physical characteristics. Several related real-time threads, can share a memory region, and the region must be active until at least the last thread has exited.

The default memory region is either the heap or the immortal memory region. Also, the initial default memory allocation area of a real-time thread can be specified when the thread is constructed. The active region associated with the real-time thread change when executing the `enter()` method, which is the mechanism to activate a region. This method associates a memory area object to a real-time thread during the execution of the `run()` method of the object passed as parameter. Also, a real-time thread can allocate outside the active region by performing the `newInstance()` or the `newArray()` methods.

Since the lifetime of objects allocated in scoped regions is governed by the control flow. Strict assignment rules placed on assignments to or from memory regions prevent the creation of dangling pointers (i.e., references from an object to another one within a potentially shorter lifetime). Then, we must ensure that the following conditions are checked before executing an assignment:

- Objects within the heap or an immortal region cannot reference objects within a scoped region.
- Objects within a scoped region cannot reference objects within a non-outer scoped region.

Illegal assignments must be checkedwhen executing instructions that store references within objects or arrays. The `IllegalAssignment()` exception throws when detecting an attempt to make an illegal pointer. Since assignment rules cannot be fully enforced by the compiler, some dangling pointers must be detected at runtime, which requires the introduction of write barriers [6]. That is, to introduce a code checking for dangling pointers when creating an assignment.

As an example, the program code of Figure 2 allocates an array `h` of 20 integers within the heap (line 17), and another array `i` of 30 integers within the immortal region (line 6). Also shows a real-time thread called `myTasks` that allocates an array `x` of 10 integers within the scoped region called `A` (line 6), an array `y` of 20 integers within the scoped region called `B` (line 10), which is inner to`A`. In this example, the following assignment statement can cause dangling pointers, as consequence are illegal references: m `h[i] = x[j]`, `h[i] = y[j]`, `i[i] = x[j]`, `i[i] = y[j]`, and `x[i] = y[j]`.

```
1:   import javax.realtime;
2:   class RegionUseExample {
3:
4:     class r1 implements Runnable {
5:     public void run() {
6:         int[] x = new int[10];
7:         ScopedMemory B = new LTMemory(1024, 1024);
8:         B.enter(new Runnable(){
9:             public void run() {
10:                int[] y = new int[20];
11:            }
12:        }
13:    }
14:
15:
16:    public static void main (String[] args) {
17:        int[] h = HeapMemory.instance().newArray(Integer, 20);
18:        int[] i = new int[30];
19:        ScopedMemory A = new VTMemory(2*1024, 4*1024);
20:        RealtimeThread myTask = new RealtimeThread(....., .....,
21:            new MemoryParameters(3*1024, 0, 1024),
22:            A, .....,
22:            new code());
23:        myTask.start();
24:    }
25:  }
```

**Figure 2. Using memory regions in RTSJ.**

Scoped regions can be nested. A safe region implementation requires that a region gets deleted only if there is no *external* reference to it. This problem has been solved by using a *reference-counter* for each region that keeps track of the use of the region by threads, and a simple reference-counting GC collects scoped memory regions when their counter reaches 0. Before cleaning a region, the `finalize()` method of all the objects in the region must be executed, and it cannot be reused until all the *f*inalizes execute to completion.

## 4. Analyzing Nested Memory Regions

In order to keep track of the currently active MR of each schedulable object, RTSJ uses a stack associated which each real-time thread. Every time a real-time thread enters a MR, the identi er of the region is pushed onto the stack. When the real-time thread leaves the MR, its identi er is popped of the stack. The stack can be used to check for illegal assignments among scoped MR[1]:

- A reference from an object *X* within a scoped region *A* to a object *Y* within a scoped region *B* is *allowed* whether the region *B* is *below* the region *A* on the stack.
- All other assignment cases among scoped regions (i.e., the region *B* is *above* the region *A* or it is not on the stack) are *forbidden*.

Note that it can appear cycles among scoped MRs on the stack. For example, if both scoped regions *A* and *B* appears on the following order: *A, B, A*, then are allowed both reference types: from *A* to *B*, and from *B* to *A*. That means that the *A* scoped MR is inner to the *A* scoped MR, and vice-versa. Since the assignment rules and the stack-based algorithm by themselves does not enforce safety pointers, the RTSJ de nes the single parent rule, which goal is to avoid scoped MR cycles on the stack.

### 4.1. The Single Parent Rule

Some of semantics and requirements that RTSJ establishes across classes supporting memory regions [10] relate to the parent of a scoped region and the single parent rule. These requirements establish a nested order for scoped regions and guarantees that a parent scope will have a lifetime that is at least that of its child scopes. The behaviour of the RTSJ suggested algorithm implicitly establishes the following parentage relation:

*"If a scoped region is not in use, it has no parent. For all other scoped objects, the parent is the nearest scope on the current entered scoped region stack. A scoped region has exactly zero or one parent."*

This parentage relation guarantees that once real-time thread has entered a set of scoped regions in a given order, any other real-time thread will have to enter them in the same order. At this time, if the scope region has no parent, then the entry is allowed. Otherwise, the real-time thread entering the scoped region must have entered every proper ancestor of it in the scope stack. There are four the operations affecting the scope stack:

1. The `enter()` method in the `MemoryArea` class.
2. The construction of a new `RealtimeThread` object.
3. The `executeInArea()` method in the `MemoryArea` class.
4. The `newInstance()` and `newInstance()` methods in the `MemoryArea` class.

Since the suggested algorithms implementing these methods require an exploration of the stack, they have a complexity of *O(n)*, where *n* is the depth of the stack.

The reference-counter of a scoped MR is increased when entering a new scoped through the `enter()` method, the creation of a real-time thread with a scoped region, or the opening of an inner scope. It is decreased when returning from the `enter()` method, when the real-time thread using the scoped region exits, or when an inner scope returns from its `enter()` method. When the reference-counter of a scope region is zero, a new nesting (parent) for the region will be possible.

Note that it is possible for a scoped region to have several parents along its live, which results in an unfamiliar programming model. But, the problem hence is that the RTSJ suggested implementation of the single parent rule can result in race carrier conditions, which gives an non deterministic behaviour to RTSJ programs.

vspace0.5cm

### 4.2. Race Carrier Conditions

The single parent rule and the parentage relation among scoped regions makes non deterministic the behaviour of the RTSJ programs. As an example, consider the code of Figure 3 that creates two scoped regions: *A* (line 8) and *B* (line 9), and two real-time threads $\tau_1$ and $\tau_2$ (line ). Where the real-time thread $\tau_1$ enters regions in the following order: *A* and *B* (line 10) , whereas $\tau_2$ enters regions as follows: *B* and *A* (line 11) . We found different behaviorus when executing this program depending on race carriers:

- If $\tau_1$ enters *A* and *B* before $\tau_2$ enters *B*, $\tau_2$ violates the single parent rule raising the `ScopedCycleException()` exception.
- But, if $\tau_2$ enters *B* and *A* before $\tau_1$ enters *A*, when $\tau_1$ tries to enter *A*, it violates the single parent rule and raises raising the `ScopedCycleException()` exception.

[1]Illegal assignments are pointers from a non-scoped MR (i.e., heap or an immortal MR) to a scoped one, or from a scoped region to a non-outer scoped region. Pointers to a non-scoped region are always allowed.

```
import javax.realtime.*;
public class MyProgram {

    LTMemory A;
    LTMemory B;

    public static void main(String[] arg) {
        A = new LTMemory(1024,1024);
        B = new LTMemory(1024,1024);
        MyTask T1 = new MyTask(A, B);
        MyTask T2 = new MyTask(B, A);
    }
}

public class MyTask {

    Runnable r1 = new Runnable() {
        public void run() {
            mr1.enter(r2);
        }
    };

    Runnable r2 = new Runnable() {
        public void run() {
            Thread.sleep(1000);
            mr2.enter(r3);
        }
    };

    Runnable r3 = new Runnable() {
        public void run() {
        .... // do some stuff
    }
};

    public MyTask(LTMemory mr1, LTMemory mr2) {
        RealtimeThread rt = new RealtimeThread(
                null, null, null, null, null, r1);
        rt.start();
    };
}
```

**Figure 3. Code of tasks $\tau_1$ and $\tau_1$ tasks.**

Let us suppose that $\tau_1$ and $\tau_2$ have entered respectively the *A* and *B* regions and both stay there for a while. In this situation, the application has two different behaviours:

- When $\tau_1$ tries to enter the *B* scoped region violates the single parent rule.
- When $\tau_2$ tries to enter the *A* scoped region violates the single parent rule.

Then the `ScopedCycleException()` exception throws by four different conditions. As consequence the programmer must deal with four executions errors, which makes this code hard to debug it. More over, the single parent rule is not violated and the application gives the correct result in the following cases:

- $\tau_1$ enters *A* and *B*, and exits both regions before $\tau_2$ enters *B* and *A*.
- $\tau_2$ enters *B* and *A*, and exits both regions before $\tau_1$ enters *A* and *B*.
- $\tau_1$ enters *A* and *B*, $\tau_1$ exits *B* before $\tau_2$ enters it, and $\tau_1$ exits *A* before $\tau_2$ tries to enter it.
- $\tau_2$ enters *B* and *A*, $\tau_2$ exits *A* before $\tau_1$ enters it, and $\tau_2$ exits *B* before $\tau_1$ tries to enter it.

Note that each of this execution cases alternate two parentage relations: *A* is parent of *B* while the $\tau_1$ execution, and *B* is parent of *A* while the $\tau_2$ execution. Then, assignments form objects allocated within *B* to objects within *A* are allowed when the executing real-time thread is $\tau_1$, and are illegal when the executing real-time thread is $\tau_2$. And assignments form objects allocated within *A* to objects within *B* are allowed when the executing real-time thread is $\tau_2$, and are illegal when the executing real-time thread is $\tau_1$. This situation that must be taken into account by the RTSJ programmer, makes difficult and tedious the programming task.

## 5. Studying Alternative Approaches

The RTSJ parentage relation is not trivial: there are orphans regions and the parent of a region can change along its life, which results in an unfamiliar programming model. More over the same program can have several results depending on race carrier conditions.

Another source of indeterminism is the stack: the introduced overhead by the algorithms exploring the stack is high and unpredictable. Each time a real-time thread is created/destroyed, enters/exits a region, or executes the `executeInArea()` or `newInstance()` method, requires the execution of a stack-based algorithm. Real-time applications require putting boundaries on the execution time of some piece of code. Since the depth of the stack associated with the real-time threads of an application are only known at runtime, to estimate the average write barrier overhead, we must limit the number of nested scoped levels that an application can hold. More over, the algorithms checking for illegal assignments are stack-based, which makes the overhead that write barriers introduce unpredictable. This unpredictability can make it impossible to establish bounds for the time taken by service requests in distributed real-time Java solutions [1]

In this section, we try to avoid these problems by proposing two alternative solutions: one of then avoids the single parent rule while the other one avoids the scope stack.

### 5.1. Removing the Scope Stack

In order to avoid race carrier conditions, we propose to change the parentage relation among MRs as follows:

*"The parent of a scoped memory area is the memory area in which the object representing the scoped memory area is allocated"*

Note that the parent of a scoped region is assigned when creating the region and does not change along the live of the region. As consequence there are not orphan regions, nor *"adopted"* regions by several times.

Consider the code of Figure 3, where the *A* and *B* scoped MRs are created within the heap. That means , the heap is the parent of both scoped regions *A* and *B*. As different that occurs in RTSJ, pointers from objects within *A* to objects within *B* and vice-versa are not allowed. Note that with this parentage relation, illegal references are known before to run the program and there are not race carrier conditions.

Let us give another example, consider two scoped regions: *A* and *B*, which have been created in the following way: the *A* region has been created within the heap, the *B* region has been created within the *A* region. Then, the creation of the *A* and *B* scoped regions gives the following parentage relation: the region *A* is the parent of *B*. Let us further consider two real-time threads $\tau_1$ and $\tau_2$, we suppose that the real-time thread $\tau_1$ has entered *A*, and $\tau_2$ has entered *B*. Suppose that $\tau_1$ enters *B* or $\tau_2$ enters *A*, at different than those that occurs in RTSJ, the single parent rule is not violated.

Instead of throwing the `ScopedCycleException()`, we have the following situation: The scoped stack associated to the real-time thread $\tau_1$ includes the *A* and *B* scoped regions. The scoped stack associated to the real-time thread $\tau_2$ includes only the *A* scoped regions. Then, even if $\tau_2$ has entered *B* before entering *A*, references from objects allocated within *A* to objects allocated within *B* are dangling pointers, as consequence they are not allowed. And references from objects allocated within *B* to objects allocated within *A* are allowed. Regarding assignment rules, we found no problem for pointers from *B* to *A* created as consequence of the $\tau_2$ execution. This situation is stable independently of the real-time thread that makes the reference.

Using this program model we can use the name of the regions in order to check for illegal assignment (e.g., we call respectively **A** and **AB** to the adobe *A* and *B* regions, and **ABC**, **ABD** .... to new regions created within the **AB** region). Then, it is not required a stack supporting the scoped regions that the real-time thread can hold, which simplifies the implementation of write barriers and the operations affecting the stack. By avoiding the stack, the overhead is significantly reduced, and programs are time-predictable.

An inconvenience of this approach is that it is less expressive than RTSJ. For example, two regions are created within region *A*, then cross references between both are disallowed. Consider a real-time thread $\tau_1$ that enters into scoped area *A* and creates both *AB* and *AC*. Then, $\tau_1$ enters into scoped area *AB* and next into *AC*. Then, only references from objects allocated within *AB* or *AC* to objects within *A* are allowed. Note that it is not possible for $\tau_1$, nor for other real-time threads, to create a reference from an object within *AB* to an object within *AC*, and vice-versa; even if $\tau_1$ must exit the area *AC* before to exit the area *AB*. Another inconvenience is that this approach requires redefining the RTSJ parentage relation [10].

### 5.2. Removing the Single Parent Rule

Race conditions carriers and non-deterministic behaviour are great problems for real-time systems, a solution approach consists to avoid the single parent rule allowing cycles references among scoped regions. Since region cycles including the heap increase considerably the complexity of the collector within the heap. And that a region cycle that incudes the immortal region becomes immortal. In order to avoid dangling pointers from objects within the heap or the immortal region to objects within a scoped region, we maintain the RTSJ assignment rules. Then, there are not cycles references among objects within scoped regions and objects within the heap or an immortalregion.

This approach requires the modifications of the reference-counter collector for scoped regions. Then, we introduce a new data structure for each scoped memory object *S*, consisting in a list taken into account all scoped regions that must be collected before to collect it (i.e., all *S* inner scoped regions). Note that by collecting regions, problems associated with reference counting collectors are solved: the space to store reference counters is minimal, and cyclic structures can be collected because they are known. By using the list of inner regions, we know all cycle structures among scoped regions on the system at a given instant. Then, we can detect whether the scoped regions that compound a cycle are not in use by any thread, and in this case they can be collected.

Note that by allowing cycles among regions on the scoped stack (i.e., by removing the RTSJ single parent rule), the assignment rules and the stack-based algorithm can themselves to enforce safety pointers. Then, we remove the `ScopedCycleException()` exception. Note that scoped region cycles are now allowed.

Allowing scope cycles in RTSJ programs becomes along the simplification of the programming model, because the programmer must not take into account the race carrier conditions. An advantage of this approaches its expressiveness, which is higher than in RTSJ. Let us consider the above example, where regions *B* and *C* are both created within region *A*. Consider further that real-time thread has been entered the regions in the following order: *A*, *B*, *C* and *B*. Then, both types of references are allowed: references from objects within *B* to objects within *C*, and references from objects within *C* to objects within *B*.

## 6. Conclusions

The RTSJ memory model presents several problems, among them *(i)* race carrier conditions on program execution, *(ii)* an unfamiliar programming model that makes it difficult and tedious for the programmer, and*(iii)* a non-time-predictable execution programs. In order to solve these problems we give two alternative solutions for the RTSJ memory model. While one of them redefines the simple parent rule and avoid the scope stack, the other removes the single parent rule and maintains the scoped stack. A comparison of these solutions, taken as reference RTSJ, is summarized in Table 6, where we use **+**, **=**, and **-** to respectively mean that the corresponding issue has improved, is similar in RTSJ, and has deteriorated.

| FEATURE / SOLUTION | Race Carrier | Programming Model | Time Overhead | Time Predictable | Expressivity Power |
|---|---|---|---|---|---|
| Redefining the SPR | + | + | + | + | - |
| Removing the SPR | + | + | - | = | + |

**Table 1. Comparison of proposed solutions**

Both solutions avoids the race carrier conditions and the `ScopedCycleException()` exception. In the solution based on redefining the single parent rule, scoped memory regions are patented at creation time which allows us to eliminate the scope stack, and introduces great advantages: *(i)* the single parent rule becomes trivially true, which simplifies the semantic of scoped memory, *(ii)* checking for all ancestors every time a real-time thread change its allocation contest are avoided, and *(iii)* illegal assignment checks are name-based, which reduces highly the time overhead and makes programs time-predictable. Note that time determinism is fundamental in real-time systems.

Taken into account static analysis based solutions to enforce the assignment rules [4], we argue that: since regions are parented at creation time, instead at entering they, the program model is simpler than the RTSJ one, and the scope tree created by the application execution becomes more static. Then, this model simplifies the scope inference algorithm. More over, since write barrier becomes time-predictable, critical tasks are tolerant with the dynamic enforcement of assignment rules.

The solution based on removing the single parent rule introduces also great advantages: *(i)* there are not single parent rule, which simplifies the semantic of scoped memory, then *(ii)* checking for all ancestors every time a real-time thread change its allocation contest are avoided, but *(iii)* it requires stack-based illegal assignment checks, which makes program execution time unpredictable. Also this solution increases the RTSJ program expressivity power, because it allows scope cycles, which results in a more flexible use of scoped regions by the programmer. The drawback of this solution becomes along the garbage collector of scoped regions, which increases highly the complexity of the RTSJ scoped region collector, as well its time overhead. We are now studying this solution.

## References

[1] A. Corsaro and R.K. Cytron. *Efficient Reference Checks for Real-time Java*. ACM SIGPLAN LCTES, 2003.

[2] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004. http://www.rtj.org.

[3] D. Gay and A. Aiken. *Memory Management with Explicit Regions*. PLDI ACM SIGPLAN, 1998.

[4] K. Palacz and J. Vitek. *Java Subtype Tests in Real Time*. ECOOP, 2003.

[5] M.T. Higuera. *Towards an Understanding of the Behavior of the Single Parent Rule*. IEEE RTAS, 2005.

[6] M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. *Region-based Memory Management for Real-time Java*. IEEE ISORC, 2001.

[7] M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. Memory Management for Real-time Java: an Efficient Solution using Hardware Support. *Real Time Systems journal*, 2004.

[8] P.C. Dibble. *Real-Time Java Platform Programming*. Prentice -Hall, 2002. http://www.rtj.org.

[9] The Real-Time for Java Expert Group. ADDISON-WESLEY, 2000. http://www.rtj.org.

[10] The Real-Time for Java Expert Group. Real-Time Specification for Java. Technical report, RTJEG, 2002. http://www.rtj.org.

[11] W.S. Beebe and M. Rinard. *An Implementation of Scoped Memory for Real-Time Java*. EMSOFT, 2001.