Towards a Revision of the Single Parent Rule in Real-Time Java, Maintaining the RTSJ Programming Model*

M. Teresa Higuera-Toledano

Facultad Informática, Universidad Complutense de Madrid, Ciudad Universitaria, 28040 Madrid Spain
Email: mthiguer@dacya.ucm.es

Abstract

The memory model used in the Real-Time Specification for Java (RTSJ) offers real-time guaranties to time-critical tasks by tuning of the Garbage Collector, providing as alternative to the garbage collected heap immortal and scoped memory regions. In order to avoid cycles among objects allocated within different scoped memory regions, the single parent rule guarantees that once a 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. This rule introduces race carrier condition in RTSJ program execution. This paper addresses two solutions to this problem that are compliant with the RTSJ programming model.

Keywords: Real-time Java, Scoped-regions, Single parent rule, Illegal assignments, Garbage collection, Write-barriers.

1. Introduction

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. However, there has been extensive research work in the area of making garbage collection compliant with real-time requirements. Main results relate to offering garbage collection techniques that enable bounding the latency caused by the execution of garbage collection.

A classical technique is incremental garbage collection [1], which enables the interleaved execution of the garbage collector with the application. The generational garbage collection strategy enables minimizing the overhead caused by garbage collection. However, this not optimize for worst-case time latency that is the goal of collectors for real-time systems.

Another technique is region-based memory allocation [8], which enables grouping related objects within a region. Commonly, regions are used explicitly in the program code (see Figure 1).

```
1:  void f ()
2:  {
3:      region r = newregion();
4:      for (i = 0; i < 10; i++) {
5:          int *x = ralloc (r, (i+1)*sizeof(int));
6:          .....
7:      }
8:      deleteregion(r);
9:  }
```

Figure1. An example of region-based allocation.

Note that the two above collection strategies are complementary: incremental garbage collection may be used within some regions in order to limit their size, while the use of regions allows reducing the runtime overhead due to garbage collection. Application of the above strategies has been studied in the context of Java, which is in particular highlighted by the *Real-time Specification for Java* (RTSJ) [9] which introduces the concept of scoped memory regions to Java.

RTSJ introduces memory regions and allows the implementation of real-time compliant garbage collectors to be run within regions except within those associated with hard timing constraints. RTSJ memory regions have different properties in terms of both the object lifetimes and the object allocation/de-allocation timing guarantees. Particularly, *immortal memory* regions are never garbage collected, and *scoped memory* regions are collected when there is not a real-time thread using the memory region. The garbage collector within the heap must scan all objects allocated within immortal or scoped memory regions for references to any object within the heap in order to preserve the integrity of the heap.

* This research was supported by Consejería de Educación de Comunidad de Madrid, Fondo Europeo de Desarrollo Regional (FEDER) and Fondo Social Europeo (FSE), through BIOGRIDNET Research Program S-0505/TIC/000101, and by Ministerio de Educación y Ciencia, through the research grant TIC2003-01321.

Because scoped regions can be reclaimed at any time, objects within a region with a longer lifetime are not allowed to create a reference to an object within another region with a potentially shorter lifetime. An RTSJ implementation must enforce these scope checks before executing an assignment. A possible solution is to perform these checks dynamically, each time a reference is stored in the memory (i.e., by using *write barriers*).

We have organized the paper as follows: we first present an in depth description of the RTSJ memory model (Section 2). We analyze the scoped cycle exception, which throws when entering a scoped region that is jet on the scope tree (i.e., the region has been entered in a different order, then it have another parent) (Section 3). Then, we present a solution avoiding the scope cycle exception when the new parent is not in the same scope stack (Section 4). We provide an outline of the state of the art and related work (Section 5). Finally a summary of our contribution conclude this paper (Section 6).

2. The RTSJ memory model

The `MemoryArea` abstract class supports the region paradigm in RTSJ through the three following kinds of regions: (*i*) immortal memory, supported by the `ImmortalPhysicalMemory` and the `ImmortalMemory` classes, that contains objects whose life ends only when the JVM terminates; (*ii*) (nested) scoped memory, supported by the `ScopedMemory` abstract class, that enables grouping objects having well-defined lifetimes and that may either offer temporal guarantees (i.e., supported by the `LTMemory` and `LTPhysicalMemory` classes) or not (i.e., supported by the `VTMemory` and `VTPhysicalMemory` classes) on the time taken to create objects; and (*iii*) the conventional heap, supported by the `HeapMemory` class. In the following, we study how these memory regions are used by a real-time application.

2.1. The memory model behavior

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.

2.3. The scoped region parentage relation

Some of semantics and requirements that RTSJ establishes across classes supporting memory regions [9] 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. Every push of a scoped memory region on a scope stack requires checking the single parent rule; this enforces the invariant that every scoped memory area has no more than one parent. The parent of a memory region (i.e., a memory area) is identified by the following rules (for a stack that grows up) [9], [2]:

- *"If the memory area is not currently on any scope stack, it has no parent."*
- *"If the memory area is the outermost (lowest) scoped memory area on any scope stack, its parent is the primordial scope."*
- *"For all other scoped memory areas, the parent is the first scoped memory area outside it on the scope stack."*
- *"Except for the primordial scope, which represents heap, immortal and immortal physical memory, only scoped memory areas are visible to the single parent rule."*

"The operational effect of the single parent rule is that when a scoped memory area has a parent, the only legal change to that value is to "no parent". Thus an ordering imposed by the first assignments of parents of a series of nested scoped memory areas is the only nesting order allowed until control leaves the scopes; then a new nesting order is possible. Thus a schedulable object attempting to enter a scope can only do so by entering in the established nesting order."

In RTSJ, the single parent rule is enforced effectively considering a tree with the primordial scope (i.e., the heap, immortal, and immortal physical memory) at its root, and other nodes corresponding to every scoped memory region that is currently on any real-time thread's scope stack. Each scoped region has a reference to its parent memory region, `ma.parent`. The parent reference may indicate a specific scoped memory area, no parent, or the primordial parent.

The code of Figure 2 shows the guidelines that RTSJ gives to maintain the scope tree and ensure that push operations on the scope stack of a real-time thread do not violate the current RTSJ single parent rule. The `ma.parent` is set to the correct parent or to `noParent`, `t.scopeStack` is the scope stack of the current real-time thread, `findFirstScope` is a convenience function that looks down the scope stack for the next entry that is a reference to an instance of `ScopedMemoryArea`.

```
if ma is scoped
      parent = findFirstScope(t.scopeStack)
      if ma.parent == noParent
            ma.parent = parent
      else if ma.parent != parent
            throw ScopedCycleException
      else
            t.scopeStack.push(ma)

findFirstScope(t.scopeStack) {
      for s = top of scope stack to bottom of scope stack
            if s is an instance of scopedMemory
                  return s
   return primordial scope
```

Figure 2. The RTSJ single parent rule.

2.2. Scoped region collection

A safe region implementation requires that a region get 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 zero. 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 finalizes execute to completion.

The reference-counter of a scoped memory region is increased when entering a new scope through the `enter()` method, when creating a `RealtimeThread` object using the scoped region, or when opening an inner scope (i.e., the reference count of all scoped regions on the current scope stack must be increased). And 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 it's `enter()` method (i.e., the reference count of all scoped regions on the current stack must be decreased).

When the reference-counter of a scoped 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 single parent rule can result in race carrier conditions, which gives a non deterministic behaviour to RTSJ programs.

3. The Scoped Cycle Exception

The single parent rule and the parentage relation among scoped regions make nondeterministic the behaviour of the RTSJ programs. As an example, consider two real-time threads T1 and T2. Where the real-time thread T1 enters regions in the following order: A and B, whereas T2 enters regions as follows: B and A. The single parent rule is not violated and the application gives the correct result in the following cases:

- T1 enters A and B, and exits both regions before T2 enters B and A.
- T2 enters B and A, and exits both regions before T1 enters A and B.
- T1 enters A and B, T1 exits B before T2 enters it, and T1 exits A before T2 tries to enter it.
- T2 enters B and A, T2 exits A before T1 enters it, and T2 exits B before T1 tries to enter it.

But the application raises the scoped cycle exception for four different cases. Then, we found another four several different behaviours when executing this program:

- If T1enters A and B before T2 enters B, T2 violates the single parent rule raising the `ScopedCycleException()` exception.
- But, if T2 enters B and A before T1 enters A, when T1 tries to enter A, it violates the single parent rule and raises the `ScopedCycleException()` exception.

Let us suppose that T1 and T2 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 T1 tries to enter the B scoped region, it violates the single parent rule.
- When T2 tries to enter the A scoped region, it violates the single parent rule.

Then the ScopedCycleException() throws by four different conditions, which makes difficult and tedious the programming task. Note that each of theses execution

cases alternate two parentage relations: A is parent of B while T1 executes, and B is the parent of A while T2 executes. Assignments form objects allocated within B to objects within A are allowed when the executing real-time thread is T1, and are illegal when the executing real-time thread is T2. Since T1 and T2 alternate their execution in concurrency, it can produce data race conditions.

4. Avoiding the Scope Cycle Exception

In order to avoid this problem and to optimize the RTSJ memory model, we simplify the parentage relation among scoped memory regions making it local to a task. This solution simplifies also the algorithms implementing this rule, but at the cost to complicate the scoped region garbage collection algorithm.

4.1. Allowing independent cycles

In this subsection, we do not consider the single parent rule. Taken into account the previous example, where the real-time thread T1 enters regions in the following order: A and B, whereas T2 enters regions as follows: B and A. We found a correct behaviour when executing this program:

- If T1 enters both regions A and B, and then T2 enters B, the single parent rule is not violated. Only references made for the T1 real-time thread from objects within B to objects allocates within A are allowed. Note that the reference count of A is 1 (i.e., it is used by the T1 real-time thread) and the reference-count of B is 2 (i.e., it is used by both the T1 and T2 real-time threads).
- If T2 enters both regions B and A, and then T1 enters A, the single parent rule is not violated. Only references made for the T2 real-time thread from objects within A to objects allocates within B are allowed. Note that the reference-count of A is 2 (i.e., it is used by both the T1 and T2 real-time threads) and the reference-count of B is 1 (i.e., it is used by the T1 real-time thread).

In this solution, illegal references are known before to run the program, and there are not race carrier conditions. If T1 has entered both regions A and B, then T2 enters also both regions B and A. Only references made for the T1 real-time thread from objects within B to objects allocates within A, and references made for the T2 real-time thread from objects within A to objects allocates within B are allowed. Note that the reference-count of both A and B are 2.

- If T1 exits B, the reference-count of B decreases to 1, and only references made for the T2 real-time thread from objects within A to objects allocates within B are allowed. Since T2 must exit A before to exit B, it is sure that B will be collected after collecting A. Then, there are not dangling pointers.
- If T2 exits A, the reference counter of A decreases to 1, and only references made for the T1 real-time thread from objects within B to objects allocates within A are allowed. Note that T1 must exit B before to exit A, as consequence there are not potential dangling pointers.

Considering the above situation, T1/T2 exits A/B, which decreases its counter to zero and can be collected.

4.2. Making local the single parent rule

As another example, we consider a real-time thread T1 entering the A and B scoped regions in the following order: A, B, and A. In this case, both types of references thus from objects allocated within A to objects within B (i.e., X.f = Y), and thus form objects allocated within B to objects within A (i.e., Y.f = X) are allowed. The reference count of A has been incremented two times (i.e., its value is 2); whereas the reference count of B has been incremented only one time.

Since T1 must exits, the regions in the following order: A, B, and A. If now T1 exits A (i.e., the scope stack of T1 is compound by AB) there are objects within the A region that have references to objects within the B region (i.e., X.f = Y). These references are dangling pointers, because T1 must exit B before to exit A, at this moment the object Y is collected. RTSJ present also the `executeInArea()` method, which allows to change the current active region to an entering before region.

In order to avoid problems with cycles among regions on the same scope stack, we redefine the single parent rule in local scope (i.e., the scope stack). Then, we consider the parentage relation independently for the scope stack of each real-time thread. Then the parent of a scoped memory region is identified by the following rule (for a stack that grows up):

"*If a scoped area is not in use, it has no parent. For all other scoped objects, the parent is the nearest scope on* ***the current entered scoped scope stack****. A scoped area has exactly zero or one parent* ***for each scope stack*** *(i.e., it can appear only one time on a given scope stack."*

Note that with this definition of the single parent rule, a scoped region can have several different parents at the same time, but only a parent for each real-time thread (i.e., a scoped region can appear only one time in each scope stack).

Considering our previous example, where the T1 real-time thread enters the A and B scoped regions, and the real-time thread T2 enters the B and A sopped regions (i.e., the scope stack of T1 is AB, and the scope stack of T2 is BA); we found that at this moment, both scoped regions have two different parents:

- The parent of scoped region A is the primordial area on the scope stack of the T1 real-time thread, and the region B on the scope stack of T2.
- The parent of scoped region B is the region A on the scope stack of T1, and the primordial area on the scope stack of T2.

However, since both scoped regions A an B have only one parent for each real-time thread, the new single parent rule is not violated. Then, the behavior of T1 and T2 execution does not depend on the order that these real-time threads gains the scoped region resources (i.e., there are not race carrier conditions).

In this solution, the single parent rule can be enforced effectively considering only the scope stack of the active real-time thread. This solution does not require that each scoped region have a reference to its parent memory region (i.e., ma.parent). More over, considering the RTSJ tree with the primordial scope (i.e., the heap, immortal, and immortal physical memory) at its root, a scoped region can have several parents, but only one for each branch of the tree (i.e. for each scope stack).

The code of figure 11 shows the our suggested implementation to maintain the scope tree and ensures that push operations on the scope stack of a real-time thread do not violate the proposed single parent rule.

```
if ma is scoped
      for s = top of scope stack to bottom of scope stack
            if s == ma
                  throw ScopedCycleException
            else
                  t.scopeStack.push(ma)
```

Figure 11. The suggested single parent rule.

Note that this algorithm requires an exploration of the scope stack, having a complexity of O(n), where n is the depth of the real-time thread's scoped stack.

4.3. Avoiding dangling pointers

By making local the single parent rule, we allow cycle references across memory regions on different stack. Let us consider the execution on the code given in Figure 3 taken into account our proposed model (i.e., the single parent rule is considered local). As different that occurs in RTSJ where the `ScopedCycleException()` raises, we can found the following situation (see Figure 9): Task T1 can create pointers from objects within B to objects within A, whereas task T2 can create pointers from objects within B to objects within A. Than means, it is possible to have cycles references. Then, both regions A and B must be collected at the same time.

This solution requires be careful with the scoped region collection: When the reference-counter of a scoped region is zero, the scoped region is garbage collectable only if it is no part of a region cycle. The code of Figure 12 shows the guidelines to maintain the data structure `ma.cycle`, which contains the set of memory regions that must be collected before to collect the scope region `ma`. Instead of the scope stack associated with each task, here we consider the general order that scoped regions have been enter by all the tasks in the system. Note that in the RTSJ VM tasks are executed in concurrency not in parallel.

```
if ma is scoped begin
    for s = top of global stack to bottom of scope stack
          if s == ma
              for t = s downto top of globalstack
                    if t == is an instance of scopedMemory
                          add t to the ma.cycle set

    t.globalStack.push(ma)
endif
```

Figure 12. Detection of local scoped region cycles.

Note that this solution requires to modify the reference counter collector by introducing the recursive `collectCycle()` function (Figure 13), which detect if the scoped region that must be collected is part of a cycle, and in this case if it is possible to collect all the regions that compound the cycle.

```
if ma.cycle is empty
      make ma garbage collectable
else
      for s = first of ma.cycle to last of ma.cycle
            if s.referenceCount ==0
                  collectCycle(s)
```

Figure 13. The recursive `collectCycle()` function.

6. Related work

The main contribution of our approach is to introduce a modification on the single parent rule definition. This solution continues the work presented in [5], which shows how RTSJ programs suffer of race carrier conditions because the single parent rule. In order to solve race carrier conditions, in [6] we propose a stricter parentage relation based on the way that scoped regions are created, instead on the order that scoped regions are entered by the real-time threads, which allows us to eliminate the scope stack. In [7], we introduce another solution avoiding both the scope stack and the scoped cycle exception, which also removes both the assignment rules. These both solutions are time-predictable. Note than deterministic and time-predictable execution are important requirements in real-time systems. At different from the solution presented in this paper, both these solutions removes the scope stack and the `ScopedCycleException()`. But, at different neither of these solutions are compliant with the RTSJ applications.

7. Conclusions

To enforce the RTSJ imposed rules, a compliant JVM must check both the single parent rule on every attempt to enter a scoped memory region, and the assignment rules on every attempt to create a reference between objects belonging to different memory regions. Since, the single parent rule at is defined by RTSJ introduces race carrier conditions, it is imperative to found alternatives solutions avoiding this problem.

The solution that we present in this paper consists in a simplification of the RTSJ single parent rule, reducing its application from the global tree of scope stacks to the current real-time thread's scope stack. The major advantage of this solution is that it conserves the memory model semantic by avoiding data race problems.

Acknowledgements: This paper has taken into account some ideas of the student Laura Herraiz, Víctor Parra, and Gabriel Salafranca.

References

[1] H. Baker. The Treadmill: Real-Time Garbage Collection without Motion Sickness. In Proc. of the Workshop on Garbage Collection in Object-Oriented Systems. OOPSLA'91, 1991. Also appears as SIGPLAN Notices 27(3), pages 66-70, March 1992.

[2] P.C. Dibble. "Real-Time Java Platform Programming". Prentice Hall 2002.

[3] M.T. Higuera, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Java Embedded Real-Time Systems: An Overview of Existing Solutions". In Proc. of the 3th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC), IEEE, March 2000.

[4] M.T. Higuera-Toledano, V. Issarny, M. Banatre, G. Cabillic, J.P. Lesot, and F. Parain. "Region-based Memory Management for Real-time Java". In Proc. of the 4th International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC). IEEE 2001.

[5] M.T. Higuera-Toledano. "Studying the Behaviour of the Single Parent Rule in Real-Time Java". In Procof 2on Workshop on Real-time Java (JTRES), 2004.M.T. Higuera-Toledano. "Towards an Understanding of the Behaviour of the Single Parent Rule in the RTSJ Scoped Memory Model". In Proc. of the 10th IEEE Real-time and Embedded Technology and Applications Symposium (RTAS). IEEE 2004.

[6] M.T. Higuera-Toledano. "Towards an Analysis of Race Carrier Conditions in Real-time Java". The 14th International Workshop on Parallel and Distributed Real-Time Systems 2006. IEEE 2006

[7] D. Gay and A. Aiken. Memory Management Regions. In Proc. of the Conference of Programming Language Design and Implementation (PLDI), ACM SIGPLAN, June 1998.

[8] The Real-Time for Java Expert Group. "Real-Time Specification for Java". RTJEG 2005. http://www.rtsj.org

[9] Sun Microsystems. "KVM Technical Specification". Technical Report. Java Community Process, May 2000. http://java.sun.com.