Analyzing the Memory Management Semantic and Requirements of the Real-time Specification of Java JSR-0000001 *

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 proposes a mechanism based on a scope three containing all scope-stacks in the system and a reference-counter collector. In order to avoid reference cycles among regions on the scope-stack, RTSJ defines the single parent rule. The given algorithms to maintain the scope-stack structure are not compliant with the defined parentage relation. More over, the suggested algorithms to maintain the single parent rule makes the application behaviour non-deterministic. This paper provides an indepth analytical investigation of the RTSJ requirements effecting the RTSJ defined parentage relation, and propose alternative approaches to avoid the indeterminism problem.

1. Introduction

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 needs research when considering the extension of Java for real-time. 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 is still problems to use this technique in time-critical systems. An alternative to the classical Garbage Collector 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 [8], 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. 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 allocates 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 (see Figure 1).

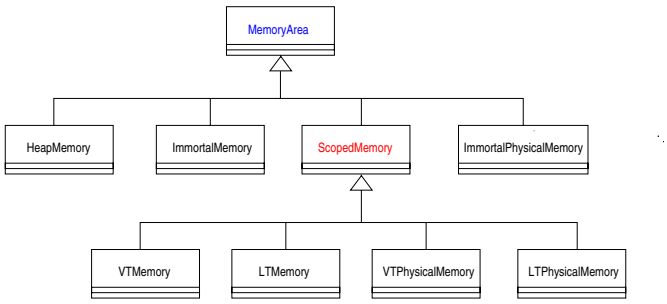


Figure 1. The `MemoryArea` hierarchy in RTSJ.

In order to define the semantic across the classes supporting memory regions, RTSJ [8] gives a list of 16 requirements (see Figure 2). From our point of view, the *(3.)* requirement that introduces a scope stack structure supporting the nested scoped region and a parentage relation, are in conflict with the requirements *(5.)* that establish another parentage relation. In this paper we review the RTSJ memory management semantic and requirements for nested scoped memory regions, we study and analyze this coherence problem and propose alternative approaches.

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

Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing
0-7695-2561-X/06 $20.00 © 2006 IEEE

Requirements *(1.)* and *(2.)*
relate respectively to the execution time taken for both object allocation within memory regions and object constructors.

Requirement *(3.)*
relates about a stack-based structure of enclosing scopes.

Requirement *(4.)*
establishes the memory region where an object is allocated.

Requirement *(5.)*
establishes a parentage relation among scoped regions.

Requirements *(6.)* and *(7.)*
establish that a scoped region have exactly zero or one.

Requirements *(8.)* and *(9.)*
establish the lifetime of objects allocated within scoped regions parent.

Requirement *(10.)*
establishes the lifetime objects allocated within the immortal region.

Requirements *(11.)* and *(14.)*
relate to critical tasks (i.e., `NoHeapRealtimeThread` instances).

Requirements *(12.)* and *(13.)*
stablish that there are only one object instance of both the heap and immortal memory.

Requirements *(15.)* and *(16.)*
impose some assignments rules to avoid dangling pointers.

Figure 2. Memory Requirements of RTSJ.

1.1. Background

This subsection relates to requirements (4.)*,* (8.)*,* (9.)*,* (10.)*,* (12.)*,* (13.)*,* (15.)*, and* (16.)[1].

In the RTSJ memory model, there is only one object instance of the heap and the immortal region in the system (requirement *(12.)* and *(13.)*), 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 (requirements *(8.)*, *(9.)*, and *(13.)*). Objects allocated within an immortal region live until the end of the application, and are never collected (requirements *(10.)*).

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 changes 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 (requirement *(4.)*). 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 (requirement *(15.)* and *(16.)*),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 checked when 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; that is, to introduce a code checking for dangling pointers when creating an assignment.

1.2. Paper organization

In this paper we review the RTSJ memory management semantic and requirements for nested scoped regions, considering and analyzing coherence among RTSJ memory management requirements and race carrier conditions, and we give some guidelines for several alternative approaches in other to solve both problems (Section 2). Finally, we offer some conclusions (Section 3).

2. Analyzing requirements for nested regions

This section shows that requirements (3.) *and* (5.) *are incompatible.*

The requirement *(3.)* associates to every real-time thread a stack that keeps track of the scoped region that can be accessed by the real-time thread. And also establishes the suggested algorithms for maintaining the scope structure. It is formulated as follows (see [8], pag. 72):

[1] Requirements *(3.)*, *(5.)*, *(6.)*, and *(7.)* are studied in the rest of this paper. And requirements *(1.)*, *(2.)*, *(11.)*, and *(14.)* are not treated in this paper

"The structure of enclosing scopes is accessible through a set of methods on `RealtimeThread`*...The algorithms for maintaining the scope structure are given in Maintaining the Scope Stack".*

"The Single Parent Rule" section of the RTSJ-document gives the algorithms to implement the four operations affecting the scope stack (see [8], pag. 73-75): *(i)* the `enter()` method in the `MemoryArea` class, *(ii)* the construction of a new `RealtimeThread` object, *(iii)* the `executeInArea()` method in the `MemoryArea` class, and *(iv)* the `newInstance()` method in the `MemoryArea` class. The suggested algorithms for the four above operations 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. The suggested RTSJ algorithm that perform this test have a time complexity of *O(n)*. This parentage relation is given on the "Maintaining the Scope Stack" section of the RTSJ-document, that can be formulated as follows (see [8], pag. 75):

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

Note that it is possible for a scoped region to have several parents along its live. The reference-counter of a scoped region is incremented/decremented each time a real-time thread enters/exits the region or an inner one (see [8], pag. 76). When the reference-counter of a scope region is zero, a new nesting (parent) for the region will be possible.

Taken into account the *(5.)* requirement, it is not possible for a scoped region to have more than one parent. The requirement is formulated as follows (see [8], pag. 72):

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

Then, the RTSJ suggested algorithms to maintain the scope stack are not compliant with the parentage relation defined by the *(5.)* requirement. Also, the parentage relation derived from the *(3.)* requirement is considerably more complex than the defined by the *(5.)* requirement. Some given approach as [9] and [5] use a stack-based algorithm to determine illegal assignments. Moreover, the new version of RTSJ (i.e, the JSR-282), the RTSJ reference implementation, and the guidelines given by some members of the RTSJ to write programs in real-time Java (e.g., [6] [2]) do not complies with the requirement *(5.)*. Another problem, is that the requirement *(3.)* together with the single parent rule introduce race carrier conditions, as we can see in the following subsection.

3. The parentage relation

In this section, we tray to ask the following question: "Given a scoped region, what memory region is its nested outer scoped (i.e., its parent)?"

Taken into account the illegal references rules, this is a essential question to determine dangling pointers. The single parent rule establishes 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 problem hence is that the RTSJ suggested implementation of the single parent rule is not compliant with is definition. In this section, we analyze and study both type of requirements, those establishing the parentage relation and those suggesting the implementation support for scoped regions.

3.1. Analyzing the *(3.)* requirement

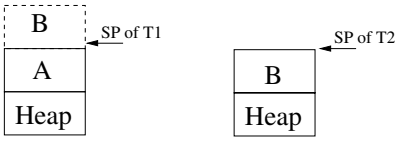
This subsection avoids the requirements (5.)*, and shows that the requirements* (3.)*,* (6.)*, and* (7.) *give a not good approach*[2].

Consider two scoped regions: *A* and *B*, and two real-time threads τ_1 and τ_2. Where the real-time thread τ_1 enters regions in the following order: *A* and *B*, whereas τ_2 enters regions as follows: *B* and *A*. Let us suppose that τ_1 and τ_2 have entered respectively the *A* and *B* regions and both stay there for a while. In this situation, the application have two different behaviours:

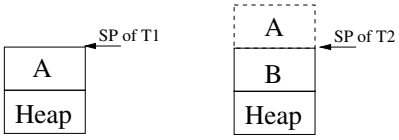
- When τ_1 tries to enter the *B* scoped region violates the single parent rule, raising the `ScopedCycleException()` exception (see Figure 3.a).
- When τ_2 tries to enter the *A* scoped region violates the single parent rule, raising the `ScopedCycleException()` exception (see Figure 3.b).

More over:

- If τ_1 enters *A* and *B* before τ_2 enters *B*, then it is τ_2 which can violate the single parent rule (see Figure 4.a).
- But, if τ_2 enters *B* and *A* before τ_1 enters *A*, when τ_1 tries to enter *A*, violates the single parent rule (see Figure 4.b).

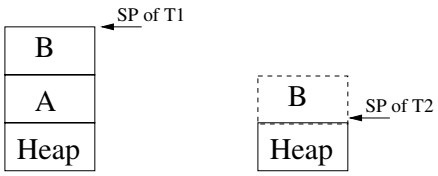


a. τ_1 violates the single parent rule.

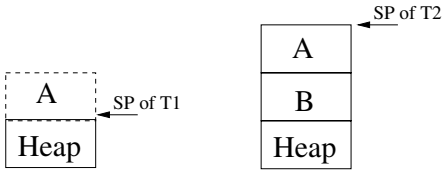


b. τ_2 violates the single parent rule.

Figure 3. Example of race carrier conditions



a. τ_2 violates the single parent rule.



b. τ_1 violates the single parent rule.

Figure 4. Example of race carrier conditions

The single parent rule is not violated and the application gives the correct result in the following cases:

- τ_1 enters *A* and *B*, and exits both regions before τ_2 enters *B* and *A*.
- τ_2 enters *B* and *A*, and exits both regions before τ_1 enters *A* and *B*.
- τ_1 enters *A* and *B*, τ_1 exits *B* before τ_2 enters it, and τ_1 exits *A* before τ_2 tries to enter it.
- τ_2 enters *B* and *A*, τ_2 exits *A* before τ_1 enters it, and τ_2 exits *B* before τ_1 tries to enter it.

The application can give the correct results or raising and exception depending on *carrier conditions*. Also the exception throws in four different cases, which makes the application program hard-debugged.

Another source of indeterminism is the scope stack. Since real-time applications require putting boundaries on the execution time of some piece of code, and the depth of the scoped region stack associated with the real-time threads of an application are only known at runtime; the overhead introduced by write barriers checking the assignment rules is unpredictable. In order to fix a maximum boundary or to estimate the average write barrier overhead, we must limit the number of nested scoped levels that an application can hold. This unpredictability can make it impossible to establish bounds for the time taken by service requests in a distributed real-time Java solution [1]. Then, this parentage relation among scoped regions presents several problems:

(i) It requires 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 checking for the single parent rule each time a real-time thread is created/destroyed, enters/exits a region, or executes the `executeInArea()` or `newInstance()` method.

(iii) It introduces higher overhead: the algorithms checking for the single parent rule and illegal references are stack-based, which have a time complexity of *O(n)*.

(iv) It is not time-predictable, thus the introduced overhead must be bounded.

*(v)*It presents race carrier conditions, which gives a non-deterministic behavior, that is contradictory with real-time systems [4].

(vi) It does not compliant with the *(5.)* RTSJ.

3.2. Avoiding race carrier conditions

In this subsection, we considerer the requirements (5.)*,* (6.)*, and* (7.)*, avoiding the* (3.) *requirement.*

Note that the requirement *(5.)* establishes the parent of a scoped region at creation time, and does not change along the live of the region. As consequence there are not *"orphan"* regions or regions *"adopted"* several times. Consider two scoped regions: *A* and *B*, which have been created as follows, 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 heap is the parent of *A*, and the

[2]Requirements *(3.)* and *(5.)* are not compatible

region *A* is the parent of *B*. Let us further consider two real-time threads τ_1 and τ_2, we suppose that the real-time thread τ_1 has entered *A*, and τ_2 has entered *B* (see Figure 5).

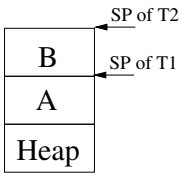
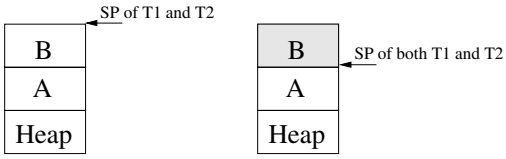


Figure 5. Scope-stack of τ_1 and τ_2.

Let us further suppose either of the following cases:

- τ_1 enters *B*, at different than those that occur in RTSJ, the single parent rule is not violated, and the scoped stack associated to both the real-time thread τ_1 and τ_2 includes the *A* and *B* scoped regions (see Figure 6.a). Then pointers from objects allocated in *A* to objects allocated in *B* are dangling pointers.
- τ_2 enters *A*. Then, The scoped stack associated to the real-time thread τ_2 includes only the *A* scoped regions. Then, even if τ_2 has entered *B* before entering *A*, pointers from objects allocated in *A* to objects allocated in *B* are dangling pointers, as consequence they are not allowed (see Figure 6.b).



a. τ_1 *enters* B. *b.* τ_2 *enters* A.

Figure 6. Scope-stack of τ_1 and τ_2.

Regarding assignment rules, we found no problem for pointers from *B* to *A* created as consequence of the τ_2 execution. Then, the following situation is stable independently of the real-time thread who makes the reference: only pointers into the heap or an immortal region, or pointers from *B* to *A* are allowed (i.e., pointers from objects allocated in *A* to objects allocated in *B* are dangling pointers, as well pointers from the heap or an immortal region to *A* and *B*).

We consider another situation: the real-time thread τ_1 enters into scoped region *A* creates *B* and *C*. Then, τ_1 enters into scoped regions *B* and *C*. Then, only references from objects allocated within *B* or *C* to objects within *A* are allowed. Note that it is not possible for τ_1, or for other real-time threads, to create a reference from an object within *B* to an object within *C*, and vice-versa from an object within *B* to an object within *C*; even if τ_1 must exit the region *C* before to exit the region *B*.

4. Conclusions

In this paper, we have analyzed the RTSJ specification, given an indepth study of the memory management requirements. We show that there are requirement that need a revision because there are some incoherencies among them. The *(5.)* requirement establishes that scoped memory regions are patented at creation time. However the guidelines given by the *(3.)* requirement to support and maintain the parentage relation, explicitly violates the *(5.)* requirement, which implicitly establishes a different parentage relation. In the suggested implementation algorithms, scoped memory regions are patented when a real-time thread changes the active memory region (i.e., each time a real-time thread is created/destroyed or enters/exits a region). This parentage relation together with requirement *(6.)* and *(7.)* result in a non-deterministic application behaviour. In order to solve this problem, we give two alternative approaches: one of them complies with the simple parent rule and avoid the scope stack; the other complies with the scoped stack, avoiding the single parent rule. The memory model of the former approach is less flexible than the RTSJ suggested implementation. By opposite, the memory model of the second approach is more flexible than the RTSJ suggested implementation, allowing scoped region cycles. We are investigating now these approaches.

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

[5] 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.

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

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

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

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