# Memory Management Based on Method Invocation in RTSJ

Jagun Kwon and Andy Wellings

Real-Time Systems Research Group
Department of Computer Science
University of York
York, England
{jagun, andy}@cs.york.ac.uk

**Abstract.** In this paper, we present a memory management model for the Ravenscar-Java profile. Because of the complexity and run-time overheads in verifying the proper use of the RTSJ's scoped memory, it is unfavourable in the area of high integrity systems where any unpredictability must be cast out. Our approach maps one anonymous memory area to a user-specifiable method by means of our Java 1.5 annotation types. This straightforward model eliminates single parent rule checks and simplifies other run-time checks that are the main cause of unpredictability and overheads. In fact, it also makes the programmer's job easier since he/she does not have to worry about creating and maintaining memory areas. All the annotated methods will be automatically converted by a transformer into an RTSJ/Ravenscar-Java compliant version. The semantics of the RTSJ remains the same, meaning that any program in our model when transformed is also a legal RTSJ program. Our key contribution is the definition of a predictable memory model and guidelines that will reduce/eliminate run-time overheads. A bonus to this is a less complicated programming model.

## 1 Introduction

Programmers do not want to be distracted by avoidable system-level activities, such as memory allocation/de-allocation. The RTSJ has added the notion of scoped memory that is intended for a more predictable execution of real-time Java programs. This can become an unnecessary burden to the programmer as well as to the underlying virtual machine. Every time an object in a memory area is assigned to a different memory area, that assignment has to be checked at run-time to prevent any dangling references. Programs are also more difficult to analyse with the additional features since, for example, memory areas can be nested and thus illegal nestings may result.

Moreover, programs can become inefficient in terms of their use of memory. Because now memory management is up to the programmer, he/she may wrongly decide to keep some unused memory areas active for a prolonged time, in which other components may be starving for more memory space. Portability and maintainability of RTSJ programs can also become problematic. On the whole, the new memory management model can introduce more programming errors and reduce the run-time performance of the whole system. However, we still want our programs to be predictable, i.e., we would like to know when memory will be allocated/de-allocated, and the overheads of such operations.

As a countermeasure to this problem, we propose here a programming utility that makes use of the original RTSJ's scoped memory to iron out some of the aforementioned concerns of Java in real-time systems. Novel to this utility is a memory model that matches one anonymous memory area to one user-decidable registered method. Programmers need no knowledge about what and how memory areas are used. The utility will automatically create a memory area for a registered method and execute that method as well as all other *non-registered* subsequent method calls that the method invokes in the memory area. Only when the registered method returns, the memory area will be reclaimed and freed (this acts as if each registered method has its own stack for all objects it creates). For this reason, the single-parent rule of the RTSJ no longer needs to be checked.

Programmers have the freedom to fine-tune memory usage of the program by executing certain methods in a new memory area, others in an existing one (i.e., caller's memory). For example, a small and infrequently invoked method can choose not to have its own memory area, but to depend on its caller's memory, so that the overhead of creating and finalizing a memory area is kept to a minimum at the expense of some additional memory space in the caller's memory area.

Assignment checks can also be reduced or completely eliminated if the programmer follows our guidelines presented in this paper. With an extensive analysis, however, such guidelines can be relieved. An analysis tool may be developed such that it checks escapement of references through method calls, local objects, and returned objects. References to outside objects[1] can be passed to the current method as parameters to that method, or as internal objects defined within the enclosing object that the method belongs to. Class objects or any other shared objects can also be assigned with a local reference. All these cases can be checked by syntactically analysing the logic of each method. Yet, this can be greatly simplified if our guidelines are in position. As outlined above, our key contribution is the definition of a predictable memory model and guidelines that will reduce/eliminate run-time overheads. A bonus to this is a less complicated programming model.

This paper is structured as follows: Section 2 will briefly review the Ravenscar-Java profile and issues related to the RTSJ. Next, we will introduce our proposed approach in detail in Section 3. This section will cover the key features, our annotations and an example. Issues on the implementation and design of our tool will be discussed in Section 4, before we briefly review some related works. Conclusions will be drawn at the end.

## 2 Brief Review of the RTSJ and Ravenscar-Java Profile

### 2.1 Ravenscar-Java Profile

In recent years, there has been a major international activity, initiated by Sun, to address the limitations of Java for real-time and embedded systems. The *Real-Time Specification for Java* (RTSJ) [2] attempts to minimise any modification to the

[1] By outside objects, we mean objects that are not created within the current method and memory area pair.

original Java semantics and yet to define many additional classes that must be implemented in a supporting virtual machine. The goal is to provide a predictable and expressive real-time environment. This, however, ironically leads to a language and run-time system that are complex to implement and have high overheads at run-time (Sources of run-time overhead include interactions between the garbage collector and real-time threads, assignment rule/single-parent rule checks for objects in different memory areas, and asynchronous operations). Software produced in this framework is also difficult to analyse with all the complex features, such as the asynchronous transfer of control (ATC), dynamic class loading, and scoped memory areas.

Following the philosophy of the Ravenscar profile for Ada [4], we have proposed a high integrity profile for real-time Java (called Ravenscar-Java [8]) along the lines of the set of software guidelines produced by the U.S. Nuclear Regulatory Commission (NRC) [6]. This restricted programming model (or a subset of Java and RTSJ) offers a more reliable and predictable programming environment by preventing or restricting the use of language features with high overheads and complex semantics. Hence, programs become more analysable in terms of timing and safety and, ultimately, become more dependable. The profile is intended for use within single processor systems.

The computational model of the profile defines two execution phases, i.e. *initialisation* and *mission phase*. In the initialisation phase of an application, all necessary threads and memory objects are created by a special thread *Initializer*, whereas in the mission phase the application is executed and multithreading is allowed based on the imposed scheduling policy. There are several new classes that will enable safer construction of Java programs (for example, *Initializer*, *PeriodicThread*, and *SporadicEventHandler*), and the use of some existing classes in Java and RTSJ is restricted or simplified due to their problematic features in static analysis. For instance, the use of any class loader is not permitted in the mission phase, and the size of a scoped memory area, once set, cannot be altered. Objects that do not need reclaiming should be allocated in the immortal memory (thus in the initialization phase). For further restrictions, see [8].

## 2.2 RTSJ's Memory Model

There is one major concern with the use of the RTSJ's memory model in high integrity applications, that is, the runtime overheads of checking the single parent and assignment rules are often unpredictable and thus undesirable. A number of approaches have been proposed to reduce the overheads of dynamic checks, but we claim in this paper that such expensive operations can be completely eliminated by a few well-defined programming guidelines and a program transformer or virtual machine support.

## 2.3 Relationship to the Profile

The work presented in this paper should be regarded as an optional extension to the Ravenscar-Java Profile. In particular, it relaxes one of the rules, i.e., "no nested scoped memory areas" constraint.

## 3 Scoped Memory Method Invocation

### 3.1 Motivation

Because in Java every object is allocated in the heap and reclaimed by the garbage collector, one does not have to care about allocating and freeing memory space for new objects. However, its unpredictability and possible interferences with real-time threads means that it is not a convincing option for hard real-time applications.
The RTSJ has added the notion of scoped memory to solve this problem. Due to the vast amount of additional features, however, a very complex programming model can be created and, especially, memory areas can be utilized in an erratic way. That means the virtual machine must check for misuses of memory areas at run-time, i.e., single-parent rule and assignment rule checks. The overheads incurred by such operations are high and unpredictable, and extremely undesirable in resource-sensitive applications. This may also hinder portability, maintainability, and analysability of RTSJ programs.

### 3.2 The Idea in a Nutshell

In many imperative and procedural languages, programs are organized into blocks of code that are called *functions*, *procedures,* or *methods* in the case of Java. Each method has a functional purpose, such that it takes an input, processes it, and returns a result. A chain of method calls is constructed at runtime that can be represented as a stack, as shown below. In most cases, parameters that are passed to a method are only read and used in the production of the result. Each method has its own stack to store parameters, immediate results, and any local/temporary variables. Any other objects are allocated in the heap in Java by default.

This model can be extended to take on board scoped memory in a way that the program becomes simpler to analyse and run-time checks are minimized. A method can be associated with an anonymous memory area created by the run-time, and that memory area is entered whenever that method is invoked. All objects will be created within the current memory area, and the area will be reclaimed when the method returns, in the same way as the method stack is allocated and freed.

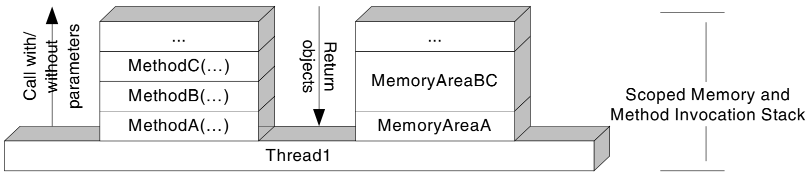

**Fig. 3.1.** Scoped Memory and Method Invocation Stack

Programmers can have the freedom to fine-tune memory usage of the program by executing certain methods in a new memory area, others in an existing memory area (i.e., caller's memory area). For example, a small and infrequently invoked method can choose not to have its own memory area, but depend on its caller's memory, so that the overheads of creating and finalizing a memory area is kept to a minimum at the expense of some additional memory space in the caller's memory area (see Figure 3.1 above).

In order to associate a method to a memory area, we use annotation types that take advantage of the Java 1.5's annotation facility [10] (See Figure 3.2 below for an illustration). Memory areas are created anonymously, so that programmers cannot gain a direct access to them. This prevents misuses of memory areas; especially, the single parent rule check is no longer required at run-time by the virtual machine. Illegal assignments can also be checked using a static Escape analysis technique [1]. We will explore these matters further in section 3.4 and 3.5.

```
…
    @ScopedMemoryMethod(   size = 10000,
        isAssignmentCheckRequired = false,
        allocateCalleesReturnedObject = true,
        allocateCalleesExceptionObject = true
        reuseMA = false)
    public anObject complexCalculation(int param) {
        @ReturnedObject(type = anObject)
        anObject rtn = new anObject(…);
        …
        for (int j = 0; j>param; j++) {
            rtn.result = method1(j));
        }
        return rtn;
      }
```

**Fig. 3.2.** Annotating a method to associate with an anonymous memory area

Objects that are to be returned to the caller can be allocated in the caller's memory area, if such objects are identified and annotated in advance [2] (see `@ReturnedObject` annotation in the above example). That way, when the current method returns, the returned object need not be copied back to the caller's memory area, saving valuable computation time.

## 3.3 Key Features

In this model, threads have no knowledge about memory areas (except the immortal memory). Each method that has an associated memory area acts as if every object created in that method is stored in the method's local stack, and they are freed after the method returns. Below are the key features of our model.

- One to one matching between a scoped memory area (MA in short hereafter) and a user-specifiable method. There is no thread-oriented memory area stack.
- An MA object can be reused after the method returns or simply be reclaimed. Programmers can specify this requirement in the annotation. When the same

[2] Programmers have the knowledge on which object is to be returned to the caller method. Such objects can be annotated, so that they will be allocated within the caller's memory area. This saves copying objects between memory areas. But, of course, the size of a returnable object must be bound.

method is invoked next time, it will be given the same anonymous MA if its reuse flag is set to true; otherwise, the MA object itself will be reclaimed.

- A method can have its own MA to use, or choose to be dependent on its outer/caller method's MA. E.g., an infrequently called method can choose not to have its own memory area.
- MAs are all independent of each other and invisible at the source code level. No MAs will be entered by more than one thread/method at the same time. In fact, threads do not know which MAs they enter (i.e., they are invisible and anonymous!). For this reason, the single-parent rule in RTSJ is no longer required.
- Programs can either be transformed into a RTSJ compliant version that utilizes scoped memory, or a virtual machine can be developed, so that it takes care of creation, allocation, and reclamation of MAs at run-time.
- Referencing objects in the same method (this MA) does not cause any concern.
- Any references to objects created in the caller methods (thus outside this MA) is allowed. Such references can be passed as parameters to the method or as local objects of the enclosing object that the method belongs to. However, references to objects in a *callee* method are not allowed. This must be analysed.
- Illegal references can be checked by a method oriented analysis tool that incorporates the escape analysis.
- Objects in the immortal memory can always be referenced, whereas those in the heap cannot. This is to prevent any interactions between the garbage collector and real-time threads.

### 3.4 Single Parent Rule Checks

In the RTSJ, it must be checked that a thread does not enter a scoped memory area that is already active. In other words, if a thread has entered memory area A, B, and C in sequence, it cannot enter A again and allocate some objects that can refer to objects created in B and C, because the lifetime of A is in fact longer than B and C, and dangling references can be created.

In our model, threads simply cannot obtain references to memory areas, and all memory areas are anonymous. Even if a thread wants to enter the same memory area twice, there is no way it can do that! All the memory areas will be created internally by the virtual machine when a registered method is invoked. This will completely eliminate runtime overheads for checking the single parent rule.

### 3.5 Assignment Violation Checks

Assignment checks fail if and only if the following condition is satisfied in our model.

*"A reference to an object within the current MA-registered method is assigned to another object that resides in a MA-registered* ***caller****."*

Assignment Violation Condition

More specifically, there are the following two cases.

**Case 1.** Direct Assignment of a method-local object to outside objects. It is fine if the enclosing object and the method share the same memory area; there is no need for any check. But if the method has entered a new memory area, then this is an illegal operation and a dangling reference can result when the method returns. The following example illustrates this point.

```
public class aClass {
    private anObject outsider;
    @ScopedMemoryMethod( … )
    public anObject aMethod(int param) {
        anObject insider = new anObject();
        outsider = insider;    // *** Assignment violation ***
        …
      }

    public anObject methodCaller(int param) {
        outsider = aMethod(param);
        …
    }
}
```

**Fig. 3.3.** Direct Assignment of a method-local object to an outside object

As shown above in *aMethod()*, when *insider* is assigned to *outsider*, an assignment violation occurs. This can be avoided if *outsider* is declared as *final*, so that the reference cannot be modified in methods. Alternatively, the method can create a local copy of the object, use it, and return it as a *ReturnedObject* mentioned earlier. If annotated by the *ReturnedObject* type, that object will be created in the parent MA. This means that the returned object shares the same memory scope as the object-local object. However, this is only possible when the parent method (the one that invokes *aMethod* in the first place, i.e., *methodCaller*()) and the object *outsider* are in the same scope.

**Case 2.** Indirect Assignment/Escapement as a parameter to a method call. If this *callee* method does not have its own memory area, then assignments are allowed unless that method invokes another one with a memory area associated with it, and passes the object to that method. The passed object and its members must only be read or '*used*' by all the callees, but not any member objects to be 're-*defined*' or assigned a new local reference by invoking a member method – this is an assignment violation. All subsequent method calls must be checked until the object in question is not passed any more to other methods and objects. Therefore, if an object can change any of its member objects' references by invoking a method, that method must share the same memory area as that of the enclosing object. Otherwise, an illegal reference can be created.

In short, it is best to declare any escapable objects as *final*. However, this seems too restrictive in some cases. Therefore, a static escape analysis can be performed to identify only those that are indeed assigned with references to method-local objects in different memory areas. The Static Single Assignment (SSA) form [5] is useful here since a program in that form can reveal whether a variable or object is *defined* (or assigned to a new value). So, first we need a graph of method invocations by conducting an escape analysis, and the SSA form representation of code for each of

```
public class aClass {
    private anObject outsider;

    @ScopedMemoryMethod( … )
    public void aMethod(anObject param) {
       anotherMethod(param);
         …
       }

    @ScopedMemoryMethod( … )
    public void anotherMethod(anObject param) {
        anotherObject x = new anotherObject();
        …
        param.xSet(x);       // *** Assignment violation ***
    }
}
```

**Fig. 3.4.** Indirect Assignment/Escapement as a parameter to a method call

the methods. Going through each method, a tool can examine whether the parameter's assignment counter, as well as the parameter's member objects' counters, ever becomes greater than 0. If the counter goes up to 1, it means that the parameter or some member object is modified and the assignment will raise an exception at runtime. Therefore, a method-level static escape analysis can identify illegal assignments statically.

## 3.6 Dealing with Returned Objects

There are situations where an object must be returned to a distant caller in a series of method invocations. When the methods do not share a memory area, such a returned object has to be relayed back to the original caller's memory area and its reference to become available for the original caller method. When the chain of methods is long, and there are more than two memory areas involved, overheads can be high due to copying the returned object from area to area. In order to resolve this, we propose an additional parameter that can specify where to store a returned object from the beginning of a method call (this parameter is added in the **@ScopedMemoryMethod** annotation; see the next section for details).

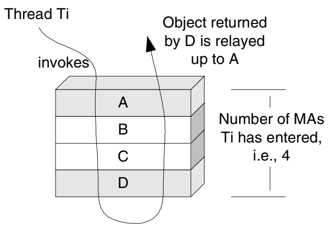
Thread Ti
invokes
Object returned by D is relayed up to A
A
B
C
D
Number of MAs Ti has entered, i.e., 4

**Fig. 3.5.** Returned objects can be stored in one of parents' memory areas

As an illustration, assume that a thread has invoked a number of methods, say A, B, C, and D, which all have their own memory areas (see Figure 3.5 above). The returned object from D can be relayed back to A, and in which case that object can be stored in A's memory area from the very beginning of D. This means that the object will not be copied back to C, B, and then A at run-time, reducing the overheads of copying objects. All returned objects must be annotated with **`@ReturnedObject`** to take advantage of this feature. Assignment checks are still required since a returned object can acquire a reference to an outside object, e.g., an object in D, C or B. Since the returned object is identified by the programmer, it can be checked by our tool that such objects are not allocated with any method local references.

This solution takes care of conditional invocations of methods. When there is an 'if' statement and two different sets of methods to call with different requirements on returned objects, we have a way of specifying that a returned object from one conditional branch will be stored in which method's memory area. Consider Figure 3.6 for a basic mode of operations.

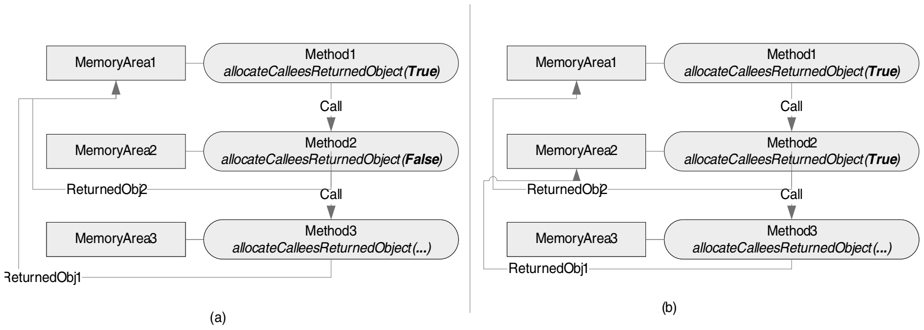
MemoryArea1
Method1
allocateCalleesReturnedObject(True)
Call
MemoryArea2
Method2
allocateCalleesReturnedObject(False)
ReturnedObj2
Call
MemoryArea3
Method3
allocateCalleesReturnedObject(...)
ReturnedObj1
(a)
MemoryArea1
Method1
allocateCalleesReturnedObject(True)
Call
MemoryArea2
Method2
allocateCalleesReturnedObject(True)
ReturnedObj2
Call
MemoryArea3
Method3
allocateCalleesReturnedObject(...)
ReturnedObj1
(b)

**Fig. 3.6.** Different ways of dealing with returned objects

The new attribute *allocateCalleesReturnedObject* is by default set to true (b), meaning that any returned object from a callee will be stored in the direct caller method's memory area. If it is set to false (a), any returned object will not be stored here, but in a memory area up in the chain where this attribute is set to true.

An interesting problem occurs, however, if both branches invoke a number of different methods, and eventually one common method that returns an object, as illustrated in Figure 3.7 below. When one branch requires the returned object to be stored in somewhere other than the memory area that the original conditional branch instruction belongs to, an appropriate *allocateCalleesReturnedObject* attribute can be set to *true* or *false*, depending on where we want the object to be stored. In case 1 in Figure 3.7, the returned object will be created in MA1 from the beginning, whereas in case 2, it will be Method5's memory area that the object will be created and stored.

The size of every returned object must be bound in order for our model to work. It becomes impossible to analyse and specify the size of a memory area if an arbitrarily sized object can be returned. Returned objects must be annotated, so that the transformer or an analyser can check the type and the caller memory area's size. It also has to be checked that a returned object does not take any method local references with it. This can cause an assignment check violation as explained in the previous section.

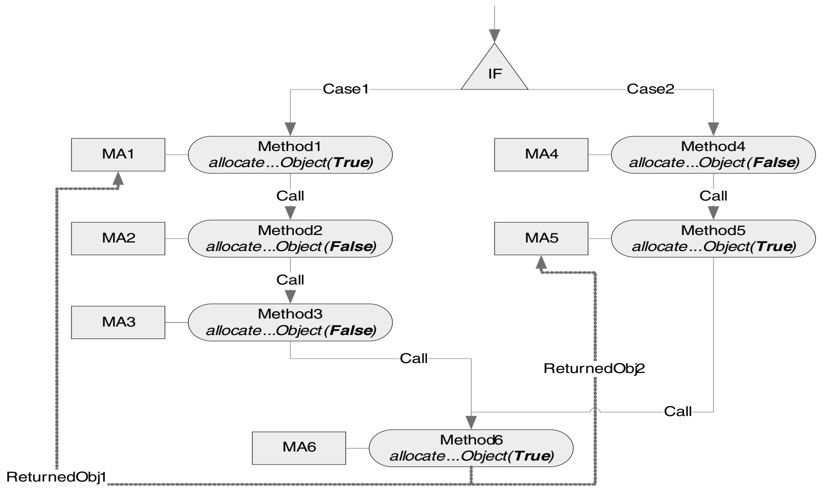
IF
Case1
Case2
MA1
Method1 allocate...Object(True)
MA4
Method4 allocate...Object(False)
Call
MA2
Method2 allocate...Object(False)
MA5
Method5 allocate...Object(True)
Call
MA3
Method3 allocate...Object(False)
Call
ReturnedObj2
Call
MA6
Method6 allocate...Object(True)
ReturnedObj1

**Fig. 3.7.** Dealing with returned objects: conditional branches with a common method call

In some cases, although an object is not syntactically returned to a caller (with a return statement), it can eventually have the same effect. If a MA-associated method needs to allocate an object in one of the callers' MAs, then that object should also be annotated with **@ReturnedObject** annotation. This can prevent illegal assignments.

## 3.7 Dealing with User Exceptions

Exceptions are represented as objects in Java. A new exception object is created at runtime when a particular exception is raised. Similar to returned objects in the previous section, exception objects should be treated in the same way, so that we can save copying exception objects between memory areas when exceptions can be propagated or relayed. We have added a new attribute in **`@ScopedMemoryMethod`** annotation type, *allocateCalleesExceptionObject*, to cater for such situations. Exception objects should also be annotated with **`@PropagatedException`** annotation to take advantage of this feature, which is defined in the next section.

An analyser or the transformer will have to check if an annotated exception is indeed handled in the method whose *allocateCalleesExceptionObject* is set to true.

## 3.8 Annotation Types and Programming Procedures

We make use of the annotation facility of Java 1.5. Our annotation types are declared in the following way and illustrated below with an example.

**ScopedMemoryMethod**
This annotation type is used to declare that the subject method will be invoked with an associated memory area. The size of a memory area must be specified; all other properties have default values.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.METHOD})
public @interface ScopedMemoryMethod {
        long size();                      // Required size of memory area
        boolean isAssignmentCheckRequired()           default true;
        boolean allocateCalleesReturnedObject()       default true;
        boolean allocateCalleesExceptionObject()      default false;
        Boolean reuseMA() default false;
                // Force to keep this MA object and reuse
}
```

We can tell the runtime that whether dynamic assignment checks are required or not. If we can be certain that such checks are not necessary by means of static analysis, then *isAssignmentCheckRequired()* field should be set to *false*.

As explained previously, it is also possible to avoid copying returned objects between scoped memory method calls. *allocateCalleesReturnedObject*() field is supplied to specify where to store a returned object in a chain of method calls with memory areas. If set to true, the returned object of any calls made by this method will be stored in the current method's memory area. This becomes useful when there are asymmetric method invocations with different requirements on where to keep returned objects.

**ReturnedObject**
This annotation is used to specify that a particular object is to be returned to the caller.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, LOCAL_VARIABLE})
public @interface ReturnedObject
{       Class type();  }              // Type of the returned object
```

**PropagatedException**
User-defined exception objects can also be pre-allocated in one of the parent memory areas.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({TYPE, FIELD, LOCAL_VARIABLE})
public @interface PropagatedException
{       Class type();  }              // Type of the exception object
```

**Example**
We present a trivial program illustrating the use of our annotations described above. In the program below, the *run*() method calls *complexCalculation*(), which then invokes *method1*(). Two memory areas are involved, and the returned object, *rtn*, will be created in *run*()'s memory area.

```
public class Thread1 extends PeriodicThread {
   @ScopedMemoryMethod(size()=1000,isAssignmentCheckRequired()=false)
    public void run() {
        Complex x = new Complex(…);
        …
        try { Complex i = complexCalculation(x);
        } catch (Exception e) {   …   }
        DoSomething(i);
    }
    @ScopedMemoryMethod(size=10000, isAssignmentCheckRequired=false,
        allocateCalleesReturnedObject = false)
    public Complex complexCalculation(Complex param){
        @ReturnedObject(type() = Complex)
        Complex rtn = new Complex(…);
        …
        for (int j = 0; j>5; j++) {
            rtn.add(method1(j));
        }
    }

    // This method is dependent on complexCalculation's MA
    public Complex method1(int i) {…}
}
```

## 4 Implementation

There are two possible implementations of the memory model we presented in this paper. One implementation would be a transformer that takes a class file, read all the annotations, analyse them, and convert the code into a RTSJ compatible version. The other way is to build a high integrity virtual machine that is aware of our annotations, and provides all the appropriate operations. In the former case, we can still keep using a RTSJ compliant virtual machine since our model does not require any changes in the semantics. This transformation will involve wrapping up annotated methods with an *enter()* method of a memory area. That way, when the method returns, the memory area will be reclaimed automatically. Annotated returned objects can be created in one of the callers' memory areas using the RTSJ's methods of *MemoryArea*. At the present time, we are implementing a transformer that can also analyse and verify class files.

## 5 Related Work

There are a number of different approaches to removing or optimizing runtime checks for RTSJ, for example [3, 9]. It is impossible to list all of them here. Most of them analyse RTSJ programs to locate non-escaping objects, and allocate such objects in a method's stack or a memory region (see [1] for a comprehensive survey). Programmers can also specify stackable objects explicitly (see [7]).

## 6 Conclusions and Future Work

We have presented a memory management model for the Ravenscar-Java profile. Our approach maps one anonymous memory area to a user-specifiable method by means of our Java 1.5 annotation types. This straightforward model eliminates single parent rule checks and simplifies other run-time checks that are the main cause of unpredictability and overheads. In fact, it also makes the programmer's job easier since he/she does not have to worry about creating and maintaining memory areas.

All the annotated methods will be automatically converted by a memory allocator/transformer into a RTSJ/Ravenscar-Java compliant version. The semantics of the RTSJ remains the same, meaning that any program in our model when compiled is also a legal RTSJ program. Benefits of using our model are obvious, i.e., a more straightforward programming model, reduced overheads caused by run-time checks, and use of existing/proven RTSJ virtual machines to our advantage. We believe that the idea presented in this paper will help facilitate the use of Java in the area of high integrity systems in the future.

However, in order for our approach to work best, we need an analysis technique for determining the worst-case memory consumption for each method. This can become complex since objects can be created dynamically in Java. We hope that the restrictions in the Ravenscar-Java Profile will make this analysis easier. We will implement our transformer and evaluate the results in the future.

## References

[1] B. Blanchet, Escape Analysis for Java: Theory and Practice, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 25, Issue 6, November 2003.

[2] G. Bollella, et al, The Real-Time Specification for Java, http://www.rtj.org.

[3] C. Boyapati, et al, Ownership types for safe region-based memory management in real-time Java, Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation, San Diego, CA, USA, 2003.

[4] A. Burns, B. Dobbing, and G. Romanski, The Ravenscar Tasking Profile for High Integrity Real-Time Programs, In L. Asplund, editor, Proceedings of Ada-Europe 98, LNCS, Vol. 1411, pages 263-275, Springer-Verlag 1998.

[5] R. Cytron et al, Efficiently Computing Static Single Assignment Form and the Control Dependence Graph, ACM Transactions on Programming Languages and Systems (TOPLAS), Vol. 13, Issue 4, October 1991.

[6] H. Hetcht, et al, Review Guidelines for Software Languages for Use in Nuclear Power Plant Systems, NUREG/CR-6463, U.S. Nuclear Regulatory Commission, 1997.

[7] J Consortium, International J Consortium Specification: Real-Time Core Extensions, Revision 1.0.14, available at http://www.j-consortium.org.

[8] J. Kwon, A. Wellings, and S. King, Ravenscar-Java: A High Integrity Profile for Real-Time Java, to appear in the Concurrency and Computation: Practice and Experience Journal, Special Issue: ACM Java Grande-ISCOPE 2002 Conference, Wiley.

[9] F. Pizlo, et al, Real-Time Java Scoped Memory: Design Patterns and Semantics, Proceedings of the 7th IEEE Intl., Symposium on Object-Oriented Real-Time Distribute Computing (ISORC'04), Vienna, Austria, May 2004.

[10] Sun Microsystems, JSR 175: A Metadata Facility for the Java Programming Language, available at http://www.jcp.org/en/jsr/detail?id=175.