Real-Time Java Scoped Memory: Design Patterns and Semantics

F. Pizlo J. M. Fox D. Holmes[†] J. Vitek
Purdue University, [†]DLTeCH

Abstract

*One of the most powerful features of the Real-Time Specification for Java (RTSJ) is the new memory management model based on scoped memory areas. This model allows programmers to ensure timely reclamation of memory and predictable performance, at the cost of an unfamiliar programming model. We report on experience using and implementing scoped memory areas. Our contribution is twofold: (*i*) we give an informal introduction to the semantics of the scope management rules of the RTSJ, (*ii*) we present a number of design patterns for effectively using the scoped memory area API.*

1. Introduction

The Real-Time Specification for Java (RTSJ) [2] holds promise to play a key role in the construction of large scale real-time systems in type-safe high-level programming languages. The benefits of Java in mission critical systems are currently being evaluated in a number of companies and labs such as Boeing [8] and JPL [7]. As of this writing a high-quality commercial implementation of the specification has been released by Timesys [5], and a number of research projects are working on open source implementations as well as alternative proposals [1, 3, 6].While the RTSJ API is rich in features, the one that is most likely to affect how real-time Java programs are written is the RTSJ's new memory management model based on scoped memory areas. This paper sets out to do the following:

- Introduce the semantics of scoped memory management informally in sufficient details for non-experts to appreciate some of the subtleties the model.
- Introduce design patterns and programming idioms suited for constructing RTSJ programs.

These observations are based on our experience implementing Ovm, a RTSJ-compliant virtual machine as well as writing RTSJ programs and benchmarks.

An important concern when using a language such as Java to implement a real-time system is the interaction of automatic memory management with real-time tasks. While garbage collection (GC) frees the programmer from the burden of tracking memory usage, it introduces uncertainty because the exact point in time at which a collection occurs is unknown, moreover none of the traditional GC algorithms provide accurate time bounds; and real-time GC algorithm still incurs running time and memory overheads that are unacceptable for some resource constrained systems. The approach advocated by the RTSJ is conceptually simple: (*a*) scoped memory areas are regions of memory not subject to garbage collection, (*b*) threads, called `NoHeapRealTimeThreads`, that do not interact with the heap and thus can not interfere with, or be interfered by, the garbage collector. These abstractions are sufficient to protect hard real-time tasks from experiencing GC related jitter.

The paper is organized as follows. Section 2 introduces the RTSJ scoped memory interface. The presentation focuses on the parts of the API relevant to the design patterns presented later. Section 3 introduces design patterns and programming idioms for RTSJ programming. These patterns are intended to provide the basis for a disciplined Real-time Java programming style. Section 4 introduces an example RTSJ program that simulates an aircraft collision detector. Section 5 concludes.

2. RTSJ Scoped Memory

Scoped memory areas were designed to meet two equally important performance requirements, namely, provide predictable allocation and deallocation performance, and ensure that hard real-time threads need not block when memory is being reclaimed by the VM. Practical consideration also constrained the design space, as RTSJ programs must coexist with soft real-time ones and legacy Java codes (both of which may experience GC pauses).

Scoped memory is similar in principle to the familiar notion of stack-based allocation. A memory scope provides a pool of memory that can be used to allocate objects. As with data within a stack frame, individual objects allocated in a scope can not be deallocated. Instead, an entire memory scope can be torn down at the same time. Each scope can be entered by multiple threads of control which will allocate objects from the same pool and communicate by shared

variables. The physical memory associated with scope can be reused as soon as all threads exit that scope.

Scopes can be nested; the nesting relation is defined by the thread entry order. When a real-time thread executing in some original memory scope enters another scope, we say that the original scope becomes the parent of the newly entered one. Since multiple threads can be executing within the same scope at any given time, a scope can be the parent of multiple other scopes. The scope hierarchy thus forms a tree (or a cactus stack) rather than a simple stack.

The RTSJ defines two distinguished scopes, represented by the Java classes `ImmortalMemory` and `HeapMemory`, respectively, for objects with unbounded lifetimes and objects that must be garbage collected. Moreover, two new kinds of threads are introduced: `RealtimeThread` and `NoHeapRealtimeThread`, both are real-time threads with the difference that the latter is protected from GC pauses. Pauses can come about for two reasons either because the GC is invoked during the execution of a real-time thread as a side effect of memory allocation, or a real-time thread is released while the GC is running and must wait until all data structures are in a consistent state (this time is defined to be the GC preemption latency in the RTSJ). `NoHeapRealtimeThread`s are immune to both problems as they run at a higher priority than the GC and can not access heap allocated objects. Thus no heap threads do not cause heap allocation and do not require heap allocated objects to be in a consistent state.

The RTSJ allows references across scopes. But as Java is a safe language it forbids the existence of dangling references. Every reference must always be a valid reference to a live object or null. To maintain safety two rules are enforced:

- Because scoped memory areas can be shared, a reference counting technique is used to ensure that the objects in them are only reclaimed after all threads have finished using the memory area.
- Because a scoped memory area could be reclaimed at any time, it is not permitted for a memory area with a longer lifetime to hold a reference to an object allocated in a memory area with a shorter lifetime. This means that heap memory and immortal memory cannot hold references to objects allocated in scoped memory. Nor can one scoped memory area hold a reference to an object allocated in a lower (more deeply nested) memory area.

Conservatively speaking, these rules require that every memory access be checked to ensure that it does not violate the rules. Combined with the heap-access restrictions of no heap threads, this imposes some overhead at run-time.

Fig. 1 is a valid scope structure composed of two memory scopes, immortal memory and heap memory. Notice the

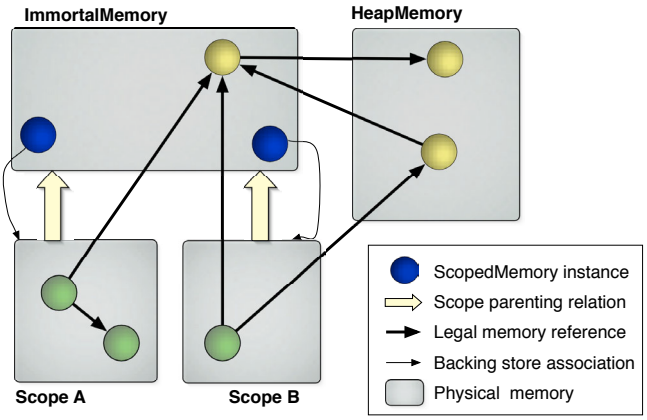

Figure 1. *Two scoped memory areas parented in immortal memory. Heavy arrows represent allowed reference patterns. While any scope is allowed to refer into the heap, a* `NoHeapRealtimeThread` *is not allowed to read those references. These constraints are implemented by read/write barriers are runtime.*

distinction between the instance of the `ScopedMemory` classes (Java objects) and the memory they denote. We show the `ScopedMemory` instance allocated within a parent scope holding a pointer to the start of the backing store used to allocate objects within that scope. The location of scoped memory instances is not directly related to their position in the scope hierarchy.

2.1. RTSJ Memory Management API

We discuss the key methods and classes that are relevant to scoped memory in the RTSJ API[1]. The `MemoryArea` class is the abstract parent of all classes representing memory. Its subclasses include `HeapMemory` and `ImmortalMemory`, both of which have a singleton instance obtained by invoking the `instance()` method. All memory area classes implement the `enter()` and `executeInArea()` methods which permit application code to execute within the allocation context of the chosen memory area. Furthermore, the `getMemoryArea()` method lets one obtain the allocation context of an object – an instance of a subclass of `MemoryArea`. Finally, all memory areas support methods to reflectively allocate objects.
`ScopedMemory` is the abstract parent class of all scoped memory classes, and it is subclassed by `LTMemory` and `VTMemory`. Allocation in an `LTMemory` area is guaranteed to take time linear in the size of the allocated object,

1 We omit discussing the `SizeEstimator` class used to estimate the size required for objects of a given class, as well as classes dealing with physical and raw memory.

while allocation times in a VTMemory may vary. Memory scopes support methods for setting and retrieving a portal object, setPortal() and getPortal(), which is used for communication between threads. getReferenceCount() returns the number of threads that have entered but not yet exited the scope. The join() method can be used to notify a thread when the reference count of the target scope reaches zero.

The only classes allowed to enter scoped memory are the two real-time thread classes. The RealtimeThread class provides a method that returns the current memory area for the current thread, getCurrentMemoryArea(), as well as a method to get the number of areas that are on the current thread's active scope stack, getMemoryAreaStackDepth(), and to obtain a reference to one of those areas, getOuterMemoryArea(). A NoHeapRealtimeThread must be associated with scoped memory or immortal memory. It can never allocate an object in the heap and is forbidden from reading or writing any reference to an object that is heap allocated.

2.2. Scoped Memory Areas Semantics

The definition of memory area given in [2] has been amended several times and the meaning of some of the finer points is still being debated. This section details the semantics of scoped memory and introduces the concepts of *scope stack* and *active scope stack*.

Reference semantics. The RTSJ memory model enforces runtime constraints on the use of object references. Fig. 1 shows all valid references patterns for a given scope hierarchy. As the figure suggests references into immortal memory and the heap are always permitted and any scope can refer into its parent scope (and transitively to all parent scopes). Attempting to create any other cross-scope references will result in a runtime exception. These invariants can be shown to be sufficient to ensure memory safety. Additionally, NoHeapRealtimeThreads are prohibited from manipulating heap references, even though it is perfectly legal for heap references to be stored in any scope. This last rule is needed to ensure that a NoHeapRealtimeThread can preempt the garbage collector.

Parented versus Unparented scopes. Every scoped memory area starts out *unparented*. An unparented scope has a reference count of zero, contains no data, and is not occupied by any threads. When entered by a real-time thread, an unparented scope becomes parented. A parented scope has a non-zero reference count, may contain data, and must be occupied by at least one thread. When a parented scope's reference count drops to zero, it becomes unparented again and objects in that scope can be finalized and deallocated.

Dynamic scope hierarchy. Scoped memory areas are related by a transitive *parent* relation. We say that a scope S1 is the *direct parent* of another scope S2 if a real-time thread entered S2 from S1. Transitively any parent of S1 is a parent of S2. In the RTSJ, the runtime scope structure is constrained by the *single parent rule* which states that a scope can only have, at any given time, a single parent scope. The parent is established when the scope is first entered and remains for the lifetime of the objects allocated in the scope (while getReferenceCount() is non-zero). While a scope can only have a single parent, it is perfectly legal for it to have multiple descendants. After all threads exit a scope, the physical memory associated with the instance of ScopedMemory is automatically zeroed out (when getReferenceCount() == 0), so that when re-entered, the scope's memory pool is available again. Re-entrant scopes are the norm in RTSJ programs.

Primordial scope. The dynamic scope hierarchy is rooted at the *primordial scope*, which is made up of HeapMemory and ImmortalMemory. Although heap and immortal memory areas have different semantics, references between them are freely allowed and both have the property of always existing; hence, it makes sense to combine them into a single primordial scope when discussing scoped memory semantics.

Scope stacks and current scope. The scope stack of each thread contains the list of all scopes entered by the thread (either via enter() or executeInArea()) in the order in which they were entered. A scope stack will contain multiple occurrences of the same scope in the case executeInArea() has been used to enter a parent scope, or when the primordial scope is entered recursively. A thread's *current scope* is the last element pushed on the stack. A thread's *parenting scope* is the last non-primordial scope pushed on the stack, or the primordial scope if no other scopes have been pushed. The role of the scope stack is to keep track of the execution state of each thread, while the parent relationship records the allowed inter-scope operations. Entering a scope entails checking that the parenting scope is either the parent of the scope about to be entered or that the scope about to be entered has no parent.

Active scope stack. A thread's active stack is a subset of the scope stack. While the scope stack grows upon executeInArea(), the active stack is unwound up to the scope that executeInArea() is applied to. The active stack is visible to the user via getMemoryAreaStackDepth() and getOuterMemoryArea(). It can be easily observed that the active stack consists of the transitive parents of the parenting scope.

Fig. 2 illustrates a valid sequence of operations on a thread's scope stack. We now discuss some of the main operations in the RTSJ API and their interpretation.

Enter. The enter() method's behavior depends on the type of scope that it is applied to. If used on the primordial scope (either HeapMemory or ImmortalMemory),

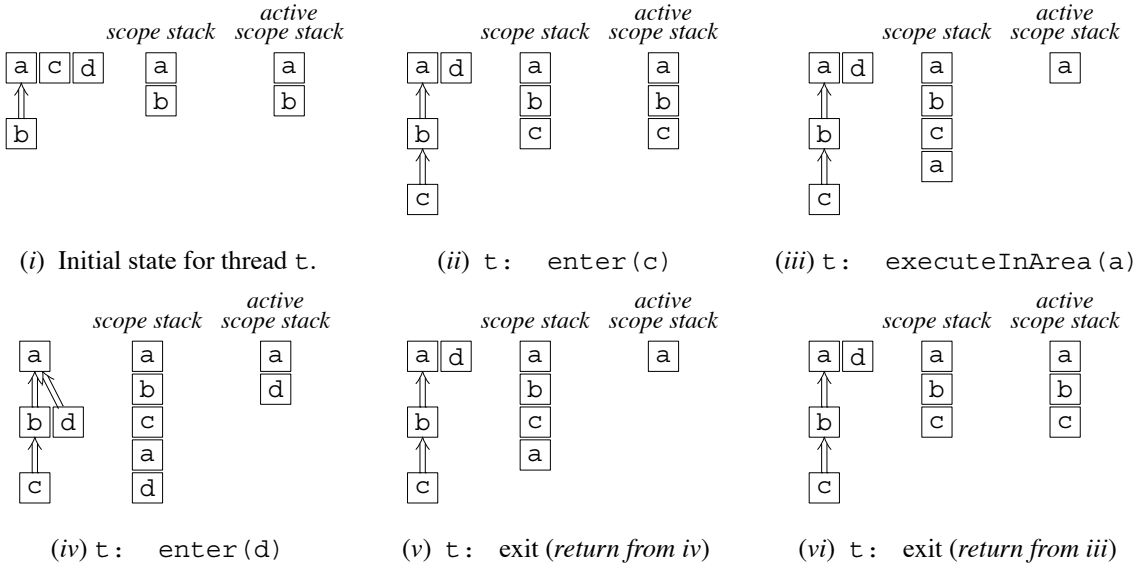

Figure 2. *The figures illustrate the parent relation between scopes and both the scope stack and active scope stack of thread* t. *Notice that entering in* c *and* d *sets their parents to* b *and* a *respectively, which are the current scopes of* t *at instants (*i*) and (*iii*). When* t *exits* d*, the scope's parent is unset. The active stack is obtained by the transitive closure of the parent's of the current scope.*

this method causes the primordial scope to be pushed on the scope stack. Otherwise, this method behaves as follows: First, it checks that the scope about to be entered is either unparented, or that its parent is the parenting scope. The scope is pushed on the thread's scope stack and the active stack, current scope, and parenting scope are updated. The reference count of the entered scope is increased by one.

ExecuteInArea. The `executeInArea()` method has as precondition that the scope about to be entered must already be a parent (within the scope hierarchy) of the current scope (this is equivalent to saying that it is on the active scope stack). The scope is pushed on the scope stack, its reference count is increased by one. The active stack is unwound so that the entered scope is the new top. The current scope is updated.

Exit. Leaving a scope occurs when a thread returns from an `enter` or `executeInArea` method (either normally or with an exception). An exit action pops the top of scope stack. The reference count of the current scope is decreased by one. If the current scope's reference count is zero, the scope is cleared and it's parent is undefined. The current scope is updated to the new top of the stack, the active stack is rewound.

GetMemoryAreaStackDepth. Return the number of element on the active scope stack.

GetOuterMemoryArea. This method returns the *i*th element in the active scope stack.

3. RTSJ Design Patterns

In this section, we describe several idioms and design patterns involving scoped memory in an effort to simplify the design of Real-Time Java programs. All methods and fields in the code fragments are public unless specified otherwise.

3.1. The Scoped Run Loop Pattern

Many real-time applications are structured around a top-level run loop which repeatedly invokes application specific code. Typically, almost all of the memory allocated in one iteration of the run loop should be recycled and made available to the loop next time around. A prototypical run loop:

```
void runLoop() {
    while ( true ) {
        . . . wait for input
        . . . some behavior
    }
}
```

In a *scoped run loop*, the loop body is extracted into a `Runnable`. The loop itself is modified to allocate a scoped memory area and repeatedly execute the body of the method within that scope. The resulting code will take advantage of memory area's predictable performance and will not have to rely on the garbage collector. At each iteration of the loop the memory area will be entered and exited again, thus reclaiming any temporary objects created by the loop's body.

```
class RunLoopIteration implements Runnable {
    void run() {
        ...wait for input
        ...some behavior  } }

void runLoop() {
    memory = new LTMemory( init_sz, max_sz);
    runLoop = new RunLoopIteration();
    while ( true ) memory.enter( runLoop); }
```

When converting plain Java code to Real-Time Java, we found that using *scoped run loop* was straightforward. However, certain constraints are placed on the loop logic. For example, preserving state from one iteration to the next requires additional code, as will be presented in Section 3.3. Furthermore, the logic may not read from, nor write to, static reference fields as these fields are allocated within immortal memory and therefore cannot be set to refer to objects allocated in scoped memory. Thus, for example, the typical implementation of singleton pattern [4] is likely to cause runtime errors.

3.2. The Encapsulated Method Pattern

The ideal scenario for scoped memory is a method which allocates intermediate results which can be discarded when that method returns. Scoped memory helps in two ways: first, allocation is performed in linear time, and second, deallocation is implicit. Certain computational tasks are particularly well-suited for being executed in a scope. *Sinks*: methods whose result set is not returned to the program, but relayed to an external entity, such as a file, a network socket, or a specialized hardware device. *Pure functions*: methods that do not modify external state.

The easiest way of scoping a method is to have the body implemented in a static inner class[2] that implements `Runnable`. The method instantiates the runnable and enters into a, likely preallocated, scope passing the runnable as a parameter. The following code shows an `EncapsulatedMethod` as a static inner class. Any arguments needed by the method should be represented by private fields of the class and initialized by the constructor.

```
static class EncapsulatedMethod
             implements Runnable {
    EncapsulatedMethod( arguments ) {
        ...initialization...  }
    void run() {
        ...method body...  } }
```

The method can be invoked by calling `enter` or `executeInArea`. For example, the following executes the method in the memory area of another object:

```
area = MemoryArea.getMemoryArea( object);
area.executeInArea(
    new EncapsulatedMethod( arguments ));
```

We assume that the only memory the method accesses is given to it in the form of parameters. The parameters may be objects of arbitrary complexity[3]. However, the parameter objects cannot be made to reference any memory allocated inside the `EncapsulatedMethod`. Also, returning a newly allocated structure object or array would violate RTSJ reference semantics. Therefore, an encapsulated method has two options for returning results to the caller: either return a primitive, or return nothing and modify some of its parameter objects *in situ*. Two restrictions remain: the caller must have a good upper bound on the size of the result, and the callee may not allocate any additional memory for storing the result. These constraints, especially the latter, may prove too restrictive. An alternative is to perform the allocation of result objects reflectively in a parent scope. The following code fragment is an example of a function returning an array of `size` elements which can be used in its parent scope.

```
area = ImmortalMemory.instance();
result = (int[]) area.newArray(
    int[].class, size);
...
return result;
```

The runtime cost and cumbersome syntax of reflective allocation is often sufficient to limit use of this idiom.

3.3. The Multi-scoped Object Pattern

While safety advocates against crossing scope boundaries, it is sometimes necessary for an object to span several scopes. This maybe done for convenience, for example, to initialize long-lived objects for the lifetime of the application, or to preserve state across scope lifetimes. Or, as described in Section 3.2, to store results of an encapsulated computation. A *multi-scoped object* is thus an instance of a class accessed from different memory scopes. The key issues here are ensuring that memory allocation performed *within* multi-scoped object is done in the correct allocation context, and that arguments to methods do not cause illegal access errors.

---

2 Java's translation scheme for *non-static* inner classes introduces a reference to the enclosing instance. In a `NoHeapRealtimeThread` an access to a heap-allocated outer object results in an exception.

3 For a `NoHeapRealtimeThread` the parameters must be allocated within a scoped region or immortal memory. A common idiom for using no-heap threads is to have a (plain) realtime thread enter a scoped region and then create the no-heap thread.

The structure of a multi-scoped object is shown in Fig. 3. The class members are split between *scope safe* and *scope unsafe* methods and fields. Primitive fields are always considered scope safe and can thus be public. Reference fields must be manipulated with care. They are thus declared private and encapsulated by accessor methods. Methods that are written to work correctly from within any scope are deemed to be safe. Methods that expect to be in a particular scope or expect that their arguments be allocated within particular scopes are unsafe. As a convention, unsafe methods are marked by declaring an unchecked exception `ScopeUnsafe`. The constructor of the class runs, by definition, in the object's allocation context and thus may assign to all fields.

```
class MultiscopedObject {
    // primitives are scope safe
    int prim;
    private Object ref;
    MultiscopedObject( ...) {
        // allocate and init from original scope
        ref = ...
    }
    // scope safe methods
    Object s1( ...) {}
    // scope unsafe methods
    Object u1( ...) throws ScopeUnsafe {}
}
```

Figure 3. *Multi-scoped Object*

Many classes can be adapted to fit this pattern with a modicum of changes. The simple case for a multi-scoped object is when the object either keeps no state, or when the object does not need to modify its state. A good example here is an `InputStream` object that does no buffering and therefore only needs to keep a handle to some operating system resource. In such a case, the object requires no modification to be used from multiple scopes. It is possible that an object needs to modify its state regularly; but if the object only manipulates primitive values, then it requires no modification either.

However, if the multi-scope object needs to perform allocation, then some changes to its code are necessary to make the object work properly from multiple scopes. Consider a `AircraftData` object that has a hashtable, and has an `update()` method that performs some computation and then updates the table by calling `put()`.

```
class AircraftData {
    private Hashtable my_hash = new Hashtable();
    void update(...) throws ScopeUnsafe {
        my_hash.put( key, val ); }
}
```

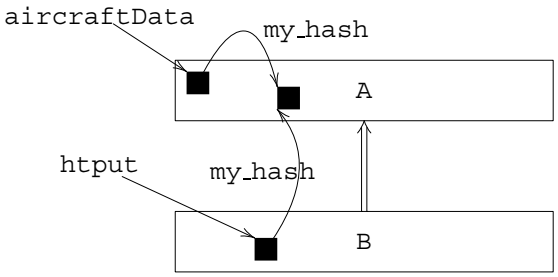

Figure 4. *The multi-scoped object* `aircraftData` *was allocated in scope* `A`*, and is being used from scope* `B`*. Invoking the* `update()` *method on it causes a* `HTPut` *instance to be allocated within* `B`*. This object will then be set to execute in the scope* `A` *to perform the modification to the hashtable referred to by variable* `my_hash`*.*

Assume that `AircraftData` is allocated in scope `A`. Naturally, `my_hash` would also be allocated in `A`, so the reference would be legal. What would happen if `update()` was called from some other scoped `B`, where `B` is a child of `A`? An error would be thrown inside `put()` because of an attempt to make an illegal reference, either due to allocation of a hashtable bucket, or if one of `key` or `val` were allocated in a child scope of `A`.

If `update()` is intended to be used from another scope, then the following changes are needed. First, the `put()` method must be executed in scoped memory area `A`. Second, we must make sure that the objects referenced by `val` and `key` were allocated in `A` as well. The `AircraftData` class can be thus rewritten as follows. (Fig. 4 illustrates this.)

```
class AircraftData {
    private Hashtable my_hash =
        new Hashtable();
    class HTPut implements Runnable {
        Object val_, key_;
        HTPut( Object key, Object val) {
            val_ = val; key_ = key;}
        void run() {
            my_hash.put( key_, val_); }
    }
    void update(...) {
        MemoryArea.getMemoryArea( this).
            executeInArea(
                new HTPut( key, val)); }
}
```

The call to `getMemoryArea()` returns the allocation context of the `AircraftData` object, namely `A`. The `put()` call then is executed within that allocation context. This is achieved by creating an encapsulated method `HTPut` and calling `executeInArea()` to switch allocation context.

Of course the new version of `AircraftData` will still fail if `key` or `val` are allocated in scoped memory area `B`. A simple idiom for ensuring that the objects are indeed allocated within `A` is to perform a deep copy of the data structures. This can be done, for example, in the body of `run()`:

```
void run() {
   my_hash.put( key_.copy(), v_.copy()); }
```

This example highlights a potential problem with multi-scoped object. More and more of the original scope, in this case `A`, will be populated with each call to methods like `update()`. For this pattern to be effective, the programmer must make sure that either no allocation is performed within the original scope, or ensure that memory of the original scope can be reclaimed before it is exhausted.

3.4. Portal Object Idioms

If an object must be shared across multiple threads then each thread must know how to obtain a reference to that object. There are typically two choices for how that reference is obtained: (*i*) It is stored in a well-known location (such as a static field of a well-known class) (*ii*) It is passed explicitly using some form of inter-thread communication (the simplest case of which, is to pass it as a parameter when constructing the thread). If the shared object must be allocated in a scope, then using a well-known static field is not an option, as static fields live in immortal memory. An alternative is to use *portal object*s.

The `ScopedMemory.setPortal( obj)` method sets the portal object to be `obj`. The portal object can only refer to an object allocated in the scope or an outer scope. The current portal object, if any, is obtained from the `getPortal()` method. However, to invoke this method, a thread must be active within the scope. When a scope is reclaimed the portal is cleared.

Initialization of the portal poses a problem: one must decide which thread sets it and when? The answer depends on how the portal is to be used. If one thread is a designated creator, and others are simply clients, then the clients should synchronize with the creator. If the first thread in the scope is supposed to setup the portal, then all threads must synchronize to establish that they are indeed first and that they should set the portal. The rub is that synchronization requires a shared object, and we are in the process of trying to set up a shared object! Again, we have two choices for choosing a lock: explicit communication or a well-known object. If explicit communication is possible then we probably do not need to be using the portal to begin with or we would communicate the portal object directly. The well-known object could be any object allocated in immortal memory or in a parent scope. For example, the creator of the scope might define a static field for this purpose:

```
static final Object portalLock = new Object();
```

But, there is already a shared object being used by all the threads in the scoped memory area: the scoped memory area itself. So a simple solution is to synchronize on the scope.

```
synchronized( ma ) {
   shared = (SharedObject) ma.getPortal();
   if ( shared == null )
      ma.setPortal( new SharedObject(...));
}
```

Note that making `getPortal()` and `setPortal()` synchronized methods would not suffice to correctly coordinate the actions of the threads as they have to atomically query if the portal object exists and create it if it does not. If the threads are not explicitly aware of which scoped memory area they are using, they can find out by invoking `getCurrentMemoryArea()`.

While using the scope object for synchronization seems simple and convenient it should be used with caution. If the application structure required users of the portal to wait until it was created, then you might be tempted to use the monitor methods `wait()` and `notifyAll()` to enforce this. The danger with this is that the `ScopedMemory` class may internally use these methods to implement the `join()` and `joinAndEnter()` methods, and so application use of them would lead to incorrect behavior.

3.5. The Wedge Thread Pattern

While the automatic reclamation of scoped memory upon exit of the last thread is often desirable, it is sometimes needed to prevent reclamation. For instance, if thread `t` stores an object reference in a portal (intended for thread `s`) and wishes to exit the scope before thread `s` enters and calls `getPortal()`. A normal exit of the scope by thread `t` could reclaim the scope's memory (if thread `s` has not yet entered the scope) and clear the portal.

The solution is to use a *wedge thread* to keep the scope alive. A wedge is a real time thread which will enter the scope and block, waiting for a signal to exit the scope. To do this, we first create a simple `ExitCondition` class. A high-level layout of this class is shown in next. The `WedgeThread` class inherits from `RealtimeThread` and will be initialized so that it has a higher priority than the thread that will start the wedge.

```
class ExitCondition {

   synchronized void waitForExit()
                throws InterruptedException;
   synchronized void signalExit();
}

class WedgeThread extends RealtimeThread {

   WedgeThread() {
      super(...);
   }
```

```
   void run() {
      exit = new ExitCondition();
      getCurrentMemoryArea().
         setPortal( exit);
      try {
         exit.waitForExit();
      } catch ( InterruptedException _ ) {
         return; }
}  }
```

A thread wishing to keep a scoped memory area alive after the enter method completes may then create a new instance of `WedgeThread` and start it. Later, a thread wishing to deallocate the scoped memory after all working threads exit would call `signalExit` on the `ExitCondition`. The exit condition object may be retrieved using `getPortal`.

start the wedge to keep the current allocation context alive

```
wedge = new WedgeThread();
wedge.start();
```

stopping the wedge destroy the scope after all other threads exit

```
exit = (ExitCondition)
   getCurrentMemoryArea().getPortal();
exit.signalExit();
```

A cleaner approach to keeping a scope alive would require additional RTSJ API support. For example, the `ScopedMemory` class could give the user access to its reference count via `retain()` and `release()` methods. This way, a thread wishing to keep a scope alive after enter completes would only need to call `retain()`. Later, a call to `release()` would be used to indicate that the scope should be deallocated as soon as all threads exit.

3.6. The Handoff Pattern

Many real-time tasks deal with filtering large amounts of data. The characteristic of such tasks is that data is first processed, then a, usually smaller, summary is retained while the main data area is reclaimed. This suggest an architecture in which one thread (usually a `NoHeapRealtimeThread`) will perform data acquisition and initial treatment, then this thread will handoff data to a second thread, not necessarily a `NoHeapRealtimeThread`, which will extract a summary and, perhaps, communicate results to other parts of the system, such as, e.g., a user interface.

The difficulty in translating this idea to real-time Java is that there is no easy way to hand off an entire scope. Assume that desired behavior is to receive data in scope `a`, summarize it in scope `b` and then free `a`. The obvious way to achieve this is to arrange for `b` to be the parent of `a`, thus allowing a thread running in `a` to store results in `b`. While this meets the requirements we outlined it has several serious drawbacks: (1) the size of memory needed for the summary must be known *before* the data is received in order to properly size scope `b`, (2) scope `b` will remain in use as long as all threads that need either `a` or `b` have not returned, (3) this approach is not appropriate for a longer sequence of handoffs, establishing a ten stage pipeline requires ten nested scopes, all pre-allocated with the right size.

The *data handoff* pattern circumvents these problems with a rather subtle use of scoped memory. In general, we set up a data handoff when two scopes `a` and `b` which are siblings (they share some parent scope `p`) need to directly exchange data. We assume that there is an instance of a class `Source` allocated in `a` and an instance of a class `Target` allocated in `b`, and that the handoff consists of repeatedly calling `read` on the source object and passing the return value as an argument to calls of `write` on the target. The handoff is performed by invoking `handoff()` with both instances.

```
class Source {
   Object read();
   boolean available(); }

class Target {
   void write( Object o); }

void handoff( Source src, Target tgt) {
   while ( src.available() )
      tgt.write( src.read());
}
```

To get this to work we will temporarily establish a reference across sibling scopes. This may appear to violate the RTSJ safety rules, but as we will explain below, it is actually safe. We assume that variable `mem_a`, `mem_b` and `mem_parent` refer to memory area objects for scopes `a`, `b` and their parent scope respectively, and that these are accessible. The code to perform the handoff will turn the handoff method into an encapsulated method with a twist. Rather than expecting the target as an argument it is obtained from a portal.

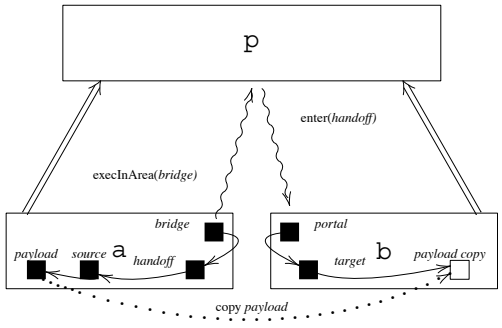

Figure 5. *The* `Target` *object copies the* `Source` *payload from scope* `a` *to sibling scope* `b`. *The reference pattern appears to break the RTSJ memory safety rules, but is actually safe due to the presence of the same thread in both areas.*

```
class Handoff implements Runnable {
    private Source src_;
    Handoff( Source src) { src_ = src; }
    void run() {
       mem = RealtimeThread
             .currentRealtimeThread()
             .getCurrentMemoryArea();
       tgt = (Target) mem.getPortal();
       handoff( src_, tgt);
    }
    void handoff( Source src, Target tgt) {
       ...
    }
```

Then from within a the following code will trigger handoff:

```
class Bridge implements Runnable {
    private handoff_;
    Bridge( Handoff handoff) {
       handoff_ = handoff; }
    void run() {
       mem_b.enter( handoff_); }
}
void doHandoff() {
    bridge = new Bridge( new Handoff( source));
    mem_parent.executeInArea( bridge);    }
```

The temporary cross-sibling scope reference is safe because no references are stored in scope b that point to scope a (a deep copy of the payload is stored in b and not a reference to the payload). It is safe to read a reference in a while in b because it is guaranteed that scope a will not disappear. The reason being that the current thread of execution has not yet exited scope a and cannot do so until it exits scope b. Thus any references that are read will be valid.

4. An Example RTSJ program

We describe a program that simulates an aircraft collision detection system in Real-Time Java. A collision detector is a single threaded hard real-time task which must be ready to receive a stream of aircrafts, identified by their call sign, along with their respective positions, and must determine if any of these aircraft are on a collision course. Deterministic running times are important as dropping a frame could result in failing to report a collision.

The algorithm is to be invoked repeatedly in a loop, checking for collisions on each iteration. The algorithm proceeds as follows. First, the next frame containing call signs and positions is retrieved. Next, the state table is used to create a list of motion objects representing the movement of aircraft from the previous frame. The algorithm has been parameterized by the collision detection strategy. The installed strategy is then queried to determine if a collision of aircraft has occurred by examining the list of motion objects created from the state table. To ensure that this thread can run with optimal efficiency, the algorithm runs as a `NoHeapRealtimeThread`. The `NoHeapRealtimeThread` is allocated in immortal memory and its arguments are copied into immortal memory. But it should be obvious that we do not want the entire collision detection system to run in the allocation context of immortal memory. Thus, the first action of the system is to create and enter a scoped memory. So the algorithm allocates this second memory scope before creating the detector object that will perform the search for collisions, and the table that holds the intermediate results. It then loops forever, entering the scoped memory area, and running the collision detection algorithm.

The collision detector was converted from plain Java using the *scoped run loop* and *multi-scoped object* patterns. In this section, we will step through some of the changes that we made to the collision detector to make it work in Real-Time Java. The plain Java collision detector consists of a run loop that follows a straight-forward algorithm: receive a frame containing recent aircraft positions, lookup the last known positions of each aircraft, check if any aircraft trajectories intersect (indicating a collision), and report the results. Each of these stages requires allocation of temporary memory to store intermediate results.

4.1. Using Scoped Run Loop

Our goal is to make the collision detector work without using the garbage collected heap. Therefore, we need to run the appropriate portions of the collision detector inside a scoped memory. For this, we used the *scoped run loop* pattern. On the surface, this looks quite simple.

```
class Runner implements Runnable {

public void run() {
    LTMemory cdmem =
       new LTMemory(initsz, maxsz);
    Detector cd =
       new Detector(new StateTable());
    while (true)
       cdmem.enter(cd);
}  }
```

The `Runner`'s allocation context is `mem`. The loop inside the run method above simply invokes the `Detector` repeatedly inside `cdmem`, so that the memory used by the detector is reclaimed at the end of each iteration.

4.2. Using Multi-Scope Object

The `Detector` class, seen in Fig. 6, is simple. It instantiates two objects: the `StateTable`, responsi-

```
class Detector implements Runnable {

    private StateTable state;
    private TwoPhaseStrategy strategy;

    Detector(StateTable state) {
        this.state = state;
        this.strategy = new TwoPhaseStrategy(); }

    void run () {
        frame = receiveFrame();
        collisions = strategy.lookForCollisions(
                     state.createMotions(f));
        send collision information to be displayed ...
    }
}
```

```
class StateTable {

    private HashMap prevState = new HashMap();

    List createMotions(Frame f) {
        ret = new LinkedList();
        for ( iter = f.iterator(); iter.hasNext(); ) {
            new_pos = new Vector3d();
            plane = iter.next( new_pos);
            old_pos = (Vector3d) prevState.get( plane);
            if ( old_pos == null ) {
                mem = MemoryArea.getMemoryArea(this);
                mem.executeInArea(
                    new HashtablePutter( plane, new_pos));
            } else
                old_pos.set( new_pos);
            ret.add( new Motion( plane, new_pos));
        }
        return ret;
    }
```

Figure 6. *The* `Detector` *and* `StateTable` *classes*

ble for keeping track of recent aircraft positions, and the `TwoPhaseStrategy`, a stateless object that runs the actual collision detection algorithm.

Each time the detector's run method is called, a new `Frame` representing the latest aircraft positions is created by native code in the `receiveFrame()` method, the frame is passed to the `StateTable` where each new position is associated with the last known position of the same aircraft, and the `lookForCollisions` method delegates to several other objects to identify all collisions (Fig. 6). All of the temporary memory used to store frames and lists objects is reclaimed when `run()` completes. This introduces a problem: the state table must keep some information from the previous frame in order to process each new frame. To resolve this, we use the techniques described in the *multi-scoped object* pattern. The state table uses `executeInArea()` every time it needs to add new objects to its collection of known aircraft; in all other cases, it modifies objects in place without any additional allocation. `StateTable` allocates new memory in the outer area only when a new aircraft enters the simulation. At present, the collision detector identifies aircraft by their call sign. So, unless the simulation uses a completely new callusing for each new aircraft, we can be fairly certain that `StateTable` will not run out of memory, even when new aircraft are being introduced into the simulation.

5. Conclusions

Scoped memory is one of the cornerstones of the Real-Time Specification for Java. It presents language implementors with a number of challenges and exposes a rich and powerful interface to programmers. This paper has presented an informal semantics of the scope management and reference rules of the RTSJ. We have identified a number of non-obvious implementation constraints and pitfalls of RTSJ programming. Finally we have documented a number of design patterns that have proven useful during our own work with the RTSJ. The patterns introduced here are a starting point towards a programming discipline for building real-time systems in Java. But more work is needed in the direction of enforcing design time correctness rules as the prospect of sudden memory access violation in the middle of real-time tasks is not pleasant.

Acknowledgments: This work was supported by the DARPA PCES program and NSF CCR-0093282.

References

[1] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. *Emsoft - LNCS*, 2211, 2001.

[2] Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Java Series. Addison-Wesley, June 2000.

[3] Angelo Corsaro and Doug Schmidt. The design and performace of the jRate Real-Time Java implementation. In *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.

[4] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1994.

[5] Timesys Inc. jTime. 2003. `www.timesys.com`.

[6] S^3 Lab. The Ovm customizable virtual machine project. 2004. `www.ovmj.org`.

[7] NASA/JPL and Sun. Golden gate. 2003. `research.sun.com/projects/goldengate`.

[8] David Sharp. Real-time distributed object computing: Ready for mission-critical embedded system applications. In *Proceeding of the Third International Symposium on Distribtued-Objects and Applications (DOA'01)*, 2001.