A High Integrity Profile for Memory Safe Programming in Real-time Java

Abstract

The Real-time Specification for Java (RTSJ) has been designed to cover a large spectrum of real-time applications, to achieve this goal the specification must cater to different real-time programming styles. This generality is essential for acceptance of Java by the industry but it also means that there are many error modes that application developers must deal with. The memory subsystem of the RTSJ is one particular area where the RTSJ's generality creates complexity. This complexity is a problem in high integrity systems as it can be the source of errors, and runtime overheads. The contribution of this paper is a new high integrity profile for memory safe programming in Real-time Java. This profile is notable in the sense that it does not restrict expressiveness of RTSJ programs, yet it guarantees that no memory-related programming errors will occur at runtime. The profile is machine checkable, and simple enough that errors can be readily corrected. While other profile have been put forward, this proposal is the first to have been evaluated on actual deployed software. We report on the use of our profile in a real-time CORBA server which has been used in an avionics application. The results are encouraging as we have been able to refactor the CORBA server relatively easily. The profile allowed to express all of the idioms present in the original system, but without any possibility of errors. Our refactoring effort also uncovered errors and resulted in an executable running 10% faster than the original.

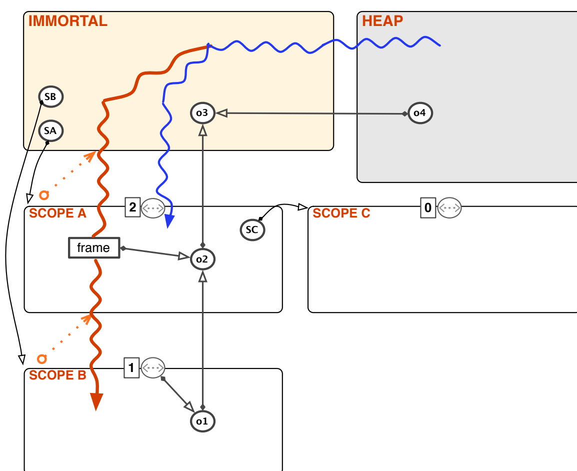
1 Introduction

The Real-Time Specification for Java (RTSJ) [3] is being used to construct large-scale Distributed Realtime Embedded (DRE) systems [17, 19]. The key benefits of the RTSJ are: first, that it allows programmers to write real-time programs in a type-safe language, thus reducing many opportunities for catastrophic failures; and second, that it allows hard-, soft- and non-real-time codes to interoperate in the same execution environment. This is becoming increasingly important as multi-million line DRE systems are being developed in Java, e.g. for avionics, shipboard computing and simulation. The success of these projects hinges on the RTSJ's ability to combine plain Java components with real-time ones.

The expressiveness of the RTSJ does have disadvantages. Any given application, which uses only the subset of the RTSJ relevant for its particular real-time requirements, will have to deal with the full range of error modes allowed by the RTSJ. For hard real-time *high integrity* software systems it is essential to minimize the likelihood of catastrophic runtime errors as these may lead to, e.g., loss of life. Wellings et.al. have identified a number of error prone features of the RTSJ and proposed a profile for the development of high integrity software that follows the guidelines of the U.S. Nuclear Regulatory Commission (NRC) [10]. This profile is called Ravenscar-Java [15] and is inspired by the SPARK subset [1] of the Ada language and the earlier Ravenscar Ada profile [6].

In this work we focus on the memory subsystem of the RTSJ and propose a new high integrity profile for memory safe programming. At the onset our goal was to provide: (1) a machine checkable profile that guarantees that no memory error will ever occur at runtime, and (2) a profile that does not unduly restrict the expressiveness of RTSJ programs. We will demonstrate that both goals have been achieved. The resulting profile is compatible with Ravenscar-Java, it simply replaces the portions of Ravenscar dealing with the memory model. In fact, we have defined a superset of Ravenscar, as we have stricter correctness guarantees (our model statically enforces the absence of runtime errors and is provably correct) and allows more programs than Ravenscar.

The RTSJ adopts a mixed-mode memory model in which garbage collection is used for non-real time activities, and manually allocated regions are used for real-time tasks. The interaction of these two memory management disciplines causes significant complexity and has the potential to cause runtime memory errors. In more detail, the above mentioned manually allocated regions are called *scoped memory areas* (or scopes). Scopes provide memory to threads executing within them, this memory is reclaimed when there are no more threads in the scope. Scopes have been specified to enforce one of the key properties of Java, namely, type-safety. This boils down to ensuring that can never be a



1. Scoped memory areas (A, B, and C) are represented by special meta-objects, instances of ScopedMemory, allocated in another scope (SA, SB, and SC).
2. Scope-allocated objects can refer to objects in the ancestor scopes or the heap. Objects in the heap (o4) can refer to objects in immortal memory.
3. Scopes have a reference count that denotes the number of threads currently active. If the count drops to 0, the scope is reclaimed.
4. Objects in the scope can only be referenced from (a) local variables of an active thread, (b) fields of objects that can refer to the scope, and (c) the scope's portal.
5. Scopes can be parented (A, B), or unparented (C). A scope is unparented if it has not been entered by a thread. A scope can be entered by several threads which communicate by shared variables.

Figure 1: *Scoped Memory Areas.*

pointer to a deleted object. The RTSJ uses runtime checks at every reference assignment to ensure that this property is respected. This means that any reference assignment, e.g. a simple statement such as `obj.f = x`, can cause an exception to be thrown at runtime. In our experience, this makes writing RTSJ code unnecessarily complex and is, in general, impossible to check statically. Figure 1 summarizes the scoped memory areas.

This paper builds on the work of Zhao *et. al.* [20] to propose a practical profile. The key insight of the Zhao *et. al.* paper was that it is necessary to make the scope structure of the program explicit in order to have a tractable verification procedure. In essence, every time the programmer writes an allocation expression of the form `new Object()`, it should be possible to know statically (i.e. at verification time) where the object fits in the scope structure of the program. It is not essential to know which particular scope it will be allocated it, but rather one should know the object's relationship with other objects in the scope hierarchy. This ensures that when an assignment expression, e.g. `obj.f = new F()`, is encountered it is possible to guarantee that the left-hand side is allocated in the current scope or a scope that has strictly longer lifetime.

The contributions of this paper are thus the definition of a profile that extends Zhao's [20] and that we have completed a full empirical evaluation of our approach (related models either went through partial validation [20], or have not reported validation results [15]). Unlike Zhao [20], we do not require extension or modification to the RTSJ, indeed our profile is defined so as to be able to run on a standard RTSJ VM. Our validation experiments were executed on the reference RTSJ VM implementation. We describe a detailed case study: the refactoring of the RTZen Object Request Broker (ORB) [12], a real-time CORBA ORB originally written with the RTSJ. Finally, we give a series of programming idioms that programmers can use to manipulate scopes when programming with our profile. These idioms are significantly simpler and more straightforward than the comparable code that would be required by the RTSJ.

2 Related Work

BeeBee and Rinard reported on the first implemented the RTSJ memory management extensions in [2]. They found it "close to impossible" to develop error-free realtime Java programs without some help from debugging tools or static analysis.

The difficulty of programming with the RTSJ has motivated Kwon, Wellings and King to propose Ravenscar-Java [15], a high-integrity profile for real-time Java based on earlier work for Ada [6]. The authors point out that while Java is a better programming language for high-integrity system than C, there are some features that are error prone. The goal of the profile is thus to define a subset of the RTSJ that can decrease the likelihood of catastrophic programming error in mission critical systems. Ravenscar mandates a simplified computational model. Applications

will be split in two phases: an initialization phase in which data structures, scopes, and threads are created and have initial values assigned to them, and a mission phase in which the real-time logic is invoked. All memory areas are created in the initialization phase and reside in immortal memory, in other words, the scope hierarchy is flat. While Ravenscar simplifies the scope structure, it does not prevent memory access violation. A similar design was also advocated by Puschner [18].

In [14], Kwon and Wellings propose another approach for a simpler RTSJ memory management model. In that work they associate scoped memory areas with methods transparently. Thus avoiding the need for explicit manipulation of memory areas. Their approach is elegant and has the potential for catching many common error by static analysis of the code. But they cannot guarantee the absence of memory violation (in general the problem is undecidable). Furthermore, their scopes cannot be multi-threaded and we see no obvious way to handle the RTSJ idioms of Sec. 5.4.

Scoped types are one of the latest development in the general area of type systems for controlled sharing of references. The goal of previous works such as Ownership Types [7] and Islands [11] was to restrict the scope of references in object-oriented programs for to enable modular reasoning. The idea of using these techniques for safety of region-based memory management was first discussed by Boyapati et al. [4]. But, unlike this proposal, their work required changes to the Java syntax and explicit type annotations. The approach proposed here is lighter, and matches more directly standard RTSJ idioms. Research in type-safe memory management for Cyclone, a dialect of C, has shown that it is possible to prevent dangling pointers even in low-level codes [9]. The RTSJ is more challenging than Cyclone as scopes can be accessed concurrently and are first-class values.

3 A High Integrity Scoped Memory Profile

The goal of the proposed high integrity profile for safe memory management is to ensure that memory management errors will not occur during the execution of mission critical Java programs. The RTSJ specifies that runtime exceptions should be generated in the following three cases: (*a*) To prevent dangling references, an exception is generated if a reference to a scope-allocated object is ever assigned to the field of a longer lived object, i.e. an object allocated in a scope which disjoint lifetime. (*b*) To prevent interference from the garbage collector, an error is generated if a hard real-time task (NoHeapRealtimeThread) attempts to read a field of a heap-allocated object. (*c*) Finally, an exception is thrown if allowing a thread to enter a scope would cause that scope to have to distinct parents (referred to as the *single parent rule*). The model proposed guarantees *at compile-time* that none of these errors will occur. Correctness of our proposal can be shown by reduction to the type system presented by Zhao *et. al.* [20] which covered some of the core features of the profile. A simple static analysis tool has been implemented to check the rules presented below

and will report any errors at verification time. It should be noted that no changes are required to the Java development environment or virtual machine. The analysis tool is run on the bytecode before the application is run. It can be executed stand alone or, potentially, as part of a development environment such as Eclipse. Any program abiding by the rules of the profile is also a valid RTSJ program.

3.1 Programming Model

The proposed profile has a simple and intuitive underlying programming model. Rather than relying on RTSJ implicit notion of allocation context, i.e. the last entered scope by the current thread, we enforce an explicit lexical discipline which guarantees that the relative location of any object is obvious from the program text. This is achieved by equating Java `package`s to memory scopes. The package hierarchy is used to specify the scope parenting relation. All instances of classes defined within the same package will be allocated together. A subpackage defines a subscope. This simplifies reasoning about RTSJ programs as two objects can only be allocated in the same scope if their classes are defined in the same package.

Fig. 2 illustrates the basic feature of the model. At runtime every scope is represented by a `ScopeGate` object. Gates are allocated in the parent scope and are the only object allowed to have references into the subscope. Whenever application code calls a method on a gate, the allocation context is switched to the scope associated to that gate for the duration of the method. Objects allocated within a scoped package are allowed to refer objects defined in a parent package (just as in the RTSJ objects allocated in a scope are allowed to refer to a parent scope). But as expected the converse is forbidden.

The profile does not restrict non real-time Java codes, and especially the standard libraries. Plain Java objects are allocated in the (garbage collected) heap and can be left as is. This is important to support backward compatibility with legacy codes.

The Scoped Memory Profile does impact the structure of Real-time Java programs. By giving an additional meaning to the `package` construct, we, *de facto* extend the language. This form of overloading of language constructs has the same rationale as the definition of the RTSJ itself, namely to extend a language without changing its syntax, compiler, or intermediate format. As for the architectural changes, this discipline imposes a different kind of functional decomposition on programs. Rather than grouping classes on the basis of some logical criteria, we group them by lifetime and function. In our experience, this decomposition is natural as RTSJ programmers must think in terms of scopes and locations in their design. Thus it is not surprising to see that classes that end up allocated in the same scope are closely coupled, and grouping them in the same package is not shocking. Also this is, arguably, a small price to pay for the associated static guarantees.

The remainder of this section is devoted to presenting the programming rules that are necessary to ensure memory

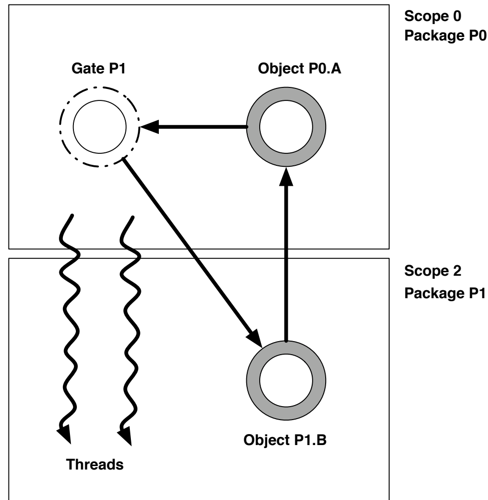


Figure 2: *Programming Model. Each runtime scope has a corresponding Java package. Objects defined in a package are always allocated in the corresponding scope with the exception of the scope's Gate which is allocated in the parent scope. All legal reference patterns are shown.*

safety.

3.1.1 Scoped Package

A package is a *scoped package* if it contains one class definition which extends the ScopeGate class. Packages that do not contain ScopeGate classes are referred to as *raw packages*. We define a distinguished Java package named imm for immortal memory. All classes defined in this package are never reclaimed.

$\mathcal{C}1$ — A scoped package must be a subpackage of imm *or of another scoped package.*

$\mathcal{C}2$ — Classes in a scoped package may not define static variables of object type.

Rule $\mathcal{C}1$ is needed to prevent nonsensical package definitions, such as having a scoped package hang-off a raw package or a raw package be a subpackage of scoped package. Rule $\mathcal{C}2$ is essential to prevent two gates of the same class from communicating via static variables (this can result in dangling references as the gates have disjoint lifetimes). The fact that a package can only have one parent package trivially ensure that the RTSJ single parent rule will hold.

$\mathcal{C}3$ — An argument to method which has been annotated as @borrowed*cannot be assigned to a field or local variable, or passed as argument to another method unless the corresponding position is also annotated as* @borrowed*. Type widening of borrowed argument is dissalowed.*

Borrowed arguments are used to loan objects to parent scopes while ensuring that the parent cannot retain a reference to this object. This allows for temporary relaxation of the scoping discipline (a feature needed in our

application). For a formal treatment of the safety of borrowing see [5].

3.1.2 Scoped Classes

A class defined within a scoped package is termed a *scoped class*. Some restrictions apply to these classes.

$\mathcal{C}4$ — A scoped class is visible only to classes in the same package or subpackages. A gate class is visible only to classes defined in the immediate super-package.

$\mathcal{C}5$ — An expression of a Scoped class type can be widened only to another class type defined in the same package.

$\mathcal{C}6$ — Methods invoked on a variable of Scoped class type must be defined in the class' defining package.

$\mathcal{C}7$ — A Scoped class may not define a `finalize` *method.*

Rule $\mathcal{C}4$ ensures that scoped classes defined in a package are accessible only to the classes defined in that package and its subpackages, while gate classes are only accessible to classes defined in their parent packages. In other words, scoped classes are *not* allowed to access classes in subpackages (other than gates). These constraints ensure that a package's gate classes form an encapsulation boundary for classes outside that package: scoped classes, and classes in subpackages are inside that encapsulation boundary. More importantly, they ensure that objects allocated in one scope may never have outgoing inferences to objects allocated in inner scopes, and thus that illegal assignment errors can never happen. Rule $\mathcal{C}5$ prevents type confusion, i.e. casting a Scoped class type to `Object`. Rule $\mathcal{C}6$ prevents a more subtle form of reference leak, within an inherited method the receiver (i.e. `this`) is implicitly cast to the method's defining class – this could lead to a leak if the method is defined in another package.

Rule $\mathcal{C}7$ is important for predictability. The RTSJ allows for finalization of objects within a scoped memory area. While this is consistent with Java there are several problems with finalization. First it is not clear which thread should perform finalization, the logical choice is the last thread in scope. But if this is a `NoHeapRealtimeThread` (which is not allowed to read heap allocated object) and some of the objects in the scope were created by a simple real-time thread, a memory access error may occur. Conversely, if a real-time thread (but not a `NoHeapRealtimeThread`) is used, the finalization procedure may block for GC. Even if a solution is found for this problem (the RTSJ does not provide one) then there is still the issue that finalization will introduce a degree of unpredictability (the last thread out must clean up for all threads in that scope). As a result we choose to forbid finalization and instead rely on the gate `onReclamation()` method.

3.1.3 Scope Gates

A scope gate, or gate, is an object that reifies scoped memory areas. Gates replace `ScopedMemory` objects (as well as the concept of `Portal`) in RTSJ. At runtime, there is a one-to-one mapping between runtime instances of a gate class

and scoped memory areas. Each gate is associated to a an instance of LTMemory, all objects allocated while executing within a method of a scoped class are allocated in that scope. The gate itself is allocated in the parent scope. Thus the gate object is the only object that can legally be stored in the field of a class defined in the parent package. Thus gates have a special status, as they do not reside in the same memory scope as other classes of the same package, yet they can refer to instances of these classes.

Fig. 3 gives the interface of the parent class of all gates. Every gate is associated to a different instance of LTMemory, but this object is hidden from view and only used internally by the implementation. By convention we require these instances of LTMemory to be allocated in immortal memory (this is because allocating them in the heap complicates the implementation of the RTSJ VM). The ScopeGate methods include an explicit reset() method that is used to reclaim the contents of a scope. The advantage of reset() over the default reclamation-on-exit policy of the RTSJ is that it avoids the need of using the wedge thread design pattern to keep a scope alive (see Sec. **??**). reset() is blocking and only takes effect when no threads are active within the gate. The method onReclamation() is called when the last live thread exits a gate – the subclass of ScopeGate is free to add finalization code by overriding that method to provide a finalization hook (see Fig. 3). The active() method returns the count of active threads within the gate.

Scope gate methods can be annotated with the Java 5.0 annotations @reclaim and @force. The meaning of @reclaim is that the reset() method should be invoked after the method returns as soon as the gate is inactive. @force means that thread that invokes this method will have exclusive access to the scope. Exceptions will be thrown in all other thread currently active in the gate. Combining @force and @reclaim ensure that a thread will execute within a clean scope.

3.1.4 Intrinsics

Some basic types must be available in all scopes, we refer to these as intrinsics. In the proposed profile we support arrays, string buffers and strings as intrinsics. They can be allocated within any package – but will be prevented from

```
abstract class ScopeGate {
    protected ScopeGate(LTMemory memory);
    public final void reset();
    public void onReclamation();
    public final int activeThreads();
}

Annotations:
    @reclaim,  @force,  @borrowed, @scopesafe
```

Figure 3: *The ScopeGate API.*

being transferred or referenced across package boundaries.

$\mathcal{C}8$ — Intrinsic class types are restricted to package-scoped (and private) fields and methods and cannot be widened to non-intrinsic class types.

$\mathcal{C}9$ — Exceptions thrown from a scoped package must be allocated in the immortal package.

$\mathcal{C}10$ — The only native methods that are allowed in a scoped package are ones annotated @scopesafe.

$\mathcal{C}11$ — Reflective calls are disallowed within a scoped package.

Rule $\mathcal{C}8$ ensure that intrinsics will never leak across packages. This restriction can be loosened by either wrapping the intrinsic in a scoped class (which can be shared with subpackages) or by using the @borrowed annotation. Rule $\mathcal{C}9$ ensures that exception object do not leak references to scope allocated objects. Rule $\mathcal{C}10$ is needed because many of the core Java method are native and cannot be checked, this gives an escape hatch to the rules. Reflection is problematic as the method invoked is not known statically, Rule $\mathcal{C}11$ simply forbids reflection. This seem adequate in the context of high integrity systems.

3.1.5 Allocation Contexts

The allocation context of thread is tied to gates in the following fashion. When a thread invokes a method of a gate, the allocation context is switched to memory area associated with the gate. Similarly, when a method of any class residing in a different scoped package is invoked, the allocation context is switched to the memory area in which the receiver object was allocated. In this way, the Scoped Types system ensures objects are instantiated into scopes corresponding to their classes' packages

$\mathcal{C}12$ — An object constructor can only be invoked in its defining package.

Rule $\mathcal{C}12$ prevents a subpackage from invoking new on a class defined in a parent package. To do this programmers should provide a factory method in the parent package.

The methods of the RTSJ scope memory areas classes must not be used by application code as they would change the allocation context in unpredictable ways.

3.1.6 Heap Access

We use a special scoped package imm.heap to hold pointers into the heap. This scoped package is special in that it can hold references to objects defined in raw packages.

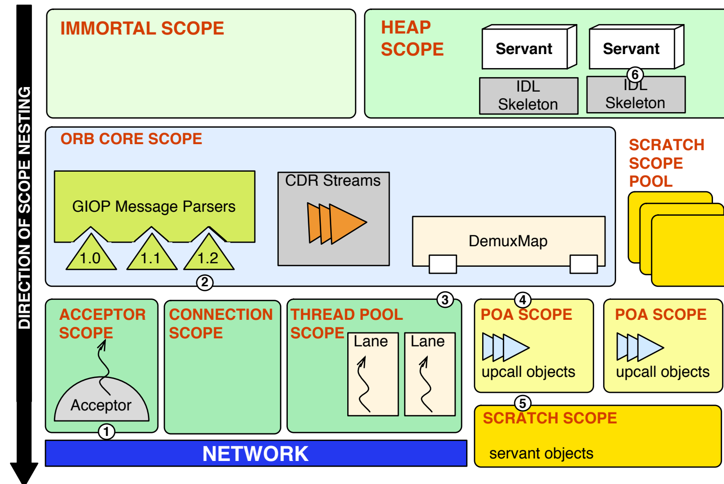
4 RTZen: A Real-Time Java ORB

The RTZen open-source ORB [12] is the first implementation of Real-Time CORBA that does not rely on C++ for predictable performance and quality of service, but instead was designed from first principles to use the Real-Time Specification for Java. Fig. 4 gives an overview of the core of the RTZen ORB [13], including connection management, data transfer, demultiplexing and concurrency control. Fig. 4 also shows how the ORB is organized into a number of RTSJ memory scopes. Under the RTSJ, the code of the ORB must explicitly manage these scopes, entering or executing code in the correct scope at the correct time. Any accidental incorrect use of scopes is likely to breach the RTSJ's constraints, crashing the ORB.

A complete description of the RTZen architecture is outside the scope of this paper. To give some idea of the complexity of programming with RTSJ scoped memories, consider the scope management required to dispatch a GIOP (generic inter-orb protocol) message to an application code for processing. Processing begins in a *connection scope*. Upon reception of a connection request, an acceptor registers it and selects a real-time thread to handle messages for that connection. The thread's event loop waits for data from the client. Once an incoming message is detected, a BufferManager (located in the *ORB Core scope* which contains persistent data structures) is contacted to obtain the proper marshalling buffer. Then the request is demultiplexed to find the proper Portable Object Adapter (POA). POA will invoke the application code stub that services the request. Once a POA has been located, a thread will be selected, this is done be entering the *thread pool scope* and finding a NoHeapRealtimeThread with a priority matching that of the message. The handler thread then enters the POA scope and finds an available ScopedMemory for processing the request. RTZen keeps a pool of scopes for that purpose. When a request has to be handled, a scope is removed from the pool, entered for the duration of the request processing and exited once the request has been processed. This scheme guarantees that all objects that were created while demarshalling and processing the request can be reclaimed. The last action performed by the handler thread is to get the POA to invoke the application logic for that request.

5 Refactoring RT-Zen with the Scope Memory Profile

To provide a proof-of-concept for our proposal, we refactored RTZen (the new version is called Scoped Zen) to abide by the above described rules. The refactoring was done in four stages. First, we designed the scope structure for Scoped Zen, based on the scopes used in RTZen. Second, we moved classes amongst packages so that Scoped Zen's package structure matched the scope structure. Third we tightened access mode and specialized the type signature of RT classes. Finally, we removed or replaced explicit RTSJ memory management idioms with equivalent constructs of our model.



① An Acceptor accepts a new connection request and creates a *Connection Scope* where a transport NoHeapRealtimeThread waits for an incoming request.
② A buffer is acquired and the appropriate GIOP Message Parser is queried to decode the request header; the request is demultiplexed to obtain the target POA. The *POA Scope* associated with that adapter is used and the internal request data is initialized.
③ The *Thread Pool Scope* is entered. A NHRT thread is selected from one of the priority lanes in the thread pool. Control is transfered to the selected thread which enters the *POA Scope*.
④ The POAImpl corresponding to the request is invoked; the servant is located; a new scratch scope is acquired from a scope pool. The current thread enter()s the *Scratch Scope*.
⑤ The IDL skeleton is invoked. ⑥ The servant code serves the request.

Figure 4: *RTZen Server ORB Architecture: Control flow path of an incoming request.*

5.1 Step 1: Extracting the Scoped Memory Architecture

RTZen consists of approximately 179,000 lines of Java code. This is a significant amount of code to refactor. Much of this code — such as the library of org.omg CORBA interfaces — is not real-time and does not use the features of

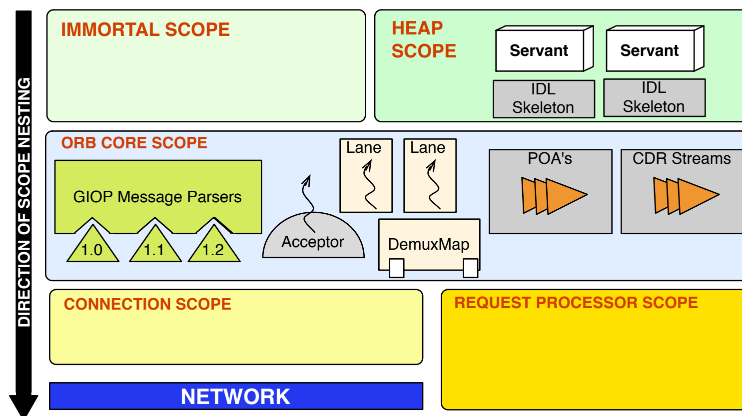


Figure 5: *Scoped Zen Server Scoped Memory Architecture.*

the RTSJ. We therefore began by identifying those parts of the complete system that was used within the real-time environment. For RTZen, this was about 10% of the total, containing approximately 185 instantiable real classes (as opposed to abstract classes and interfaces) and about $18,000$ lines of code. As we needed to refactor only the real-time portion of the code, this is a key reduction of scale — although refactoring $18,000$ lines of RTSJ code remained a significant challenge.

After identifying the real-time core of RTZen, we then analysed its scope structure (that is, we gathered the information presented earlier in Fig. 4 and discussed in Sec. 4). To do this, we annotated RTZen constructors to log the scopes in which they were created, using reflexive features of the RTSJ to identify the scopes. This analysis showed that all of the scopes involved in the internal implementation of the ORB — the POA, acceptors, and thread pool — have the same lifetime as the ORB scope. This implied that there was no need to separate them: all these scopes could be combined into the main ORB scope. Fig. 5 shows the resulting architecture. The server side of Scoped Zen consists of three scopes: the core ORB scope, the connection scope, and a scope to handle request processing. The client side of Scoped Zen contains one more additional scope: the request waiter. The latter is used by the client side to execute the methods of org.omg.CORBA.Object hiding the internal CORBA mechanism of accessing the server where the actual servant is located.

5.2 Step 2: Refactoring Classes into Scoped Packages

We require the classes of objects that will be allocated within a particular scope to be declared within the package corresponding to that scope. Having established Scoped Zen's structure, we then had to relocate classes into the correct packages, adjusting their definitions where necessary, and establish the necessary gate classes to give access to those scopes.

Our scope analysis also allowed us to eliminate a large number of RTZen classes whose only functionality was managing the proper execution of methods in the right scopes at the right time. Additional analysis allowed us to eliminate dead classes. This left us with approximately 140 classes that would be needed in Scoped Zen, and a map (like Fig. 5 but including all classes) showing which classes had to be allocated inside which scope. These classes then had to be moved into the packages representing their scopes. We moved scoped classes into one of four new packages (three for the server side of Scoped Zen and one for the client side), all subpackages of scope, corresponding to scopes in our architectural design. Fig. 6 illustrates the way we moved classes between packages (and discarded some others). The left-hand column of the table shows the number of classes in each package of RTZen. The right-hand column shows the same for Scoped Zen, in particular the four imm.* packages containing the Scoped Types.

5.3 Step 3: Access Mode Adaption and Type Specialization

Once the package structure of the program has been created, we must make sure that field and class visibility are fixed to provide proper access to the members in the new structure. Final fields must be refactored into additional instance fields. Finally, since the integrity rules prevent widening of scoped classes to non-scoped classes, it is necessary to specialize types of fields and arguments to methods. For instance, if a method takes an argument of a non-scoped type, such as Object, the method's signature must be modified to become more specific and to refer to the exact scoped type required. This refactoring is rather tedious as the type specialization often percolates across the class hierarchy. But the refactoring does not change the structure of the code and can be done in a few hours.

5.4 Step 4: Refactoring Common Idioms

RTSJ programmers have adopted or developed a number of programming idioms to manipulate scopes. After changing the structure of RTZen, we need to convert these idioms into corresponding idioms that abide by our rules. In almost every case, the resulting code was simpler and more general, because it could directly manipulate standard Java objects rather than having to create and manage special RTSJ scope meta-objects explicitly.

RTZen packages	classes per package	**Scoped Zen packages**	classes per package
zen.orb	38	zen.orb	16
zen.orb.any	2	–	
zen.orb.any.monolithic	1	–	
zen.orb.dynany	11	–	
zen.orb.giop	6	zen.org.giop	4
zen.orb.giop.IOP	3	zen.orb.giop.IOP	3
zen.orb.giop.type	5	zen.orb.giop.type	5
zen.orb.giop.v1_0	9	zen.orb.giop.v1_0	9
zen.orb.giop.v1_1	5	zen.orb.giop.v1_1	5
zen.orb.giop.v1_2	4	zen.orb.giop.v1_2	4
zen.orb.policies	13	zen.orb.policies	9
zen.orb.resolvers	2	–	
zen.orb.transport	11	zen.orb.transport	3
zen.orb.transport.iiop	4	zen.orb.transport.iiop	1
zen.poa	16	zen.poa	3
zen.poa.mechanism	27	zen.poa.mechanism	19
zen.poa.policy	7	zen.poa.policy	7
zen.util	21	zen.util	11
		imm.orb	45
		imm.orb.connection	7
		imm.orb.requestprocessor	10
		imm.requestwaiter	3

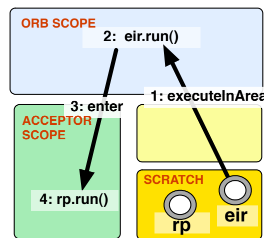
Figure 6: *Package structure of RTZen (left) and Scoped Zen (right)*

5.4.1 Sibling Scope Invocation

A common problem in RTSJ is for a thread executing in one scope to invoke a method within the allocation context of a sibling scope. Since RTSJ disallows a thread to directly enter a sibling scope another solution is needed. The *Execute in Runnable* idiom (or EIR) is a widely used idiom which precisely addresses this problem. Fig. 7 illustrates this idiom with a slightly simplified example from Zen. The left-hand side shows the generic EIR class. The right-hand side shows a fragment of the code processing incoming messages. Once a request has been received and processed by the user, the reply must be sent from the acceptor scope. To do this, a new EIR and a new processor are created. The EIR first changes allocation context to the parent scope (orb), and then enters the acceptors scope and executes the code that will send the reply (ProcessorRunner.run(), not shown here).

```
class ExecuteInRunnable {
    Runnable action;
    MemoryArea area;
    void init(Runnable r,MemoryArea a)
        { action = r; area = a; }
    void run() { area.enter( action ); }
}
```

(a) The Zen ExecuteInRunnable class



(b) Scope Hierarchy

```
eir = new ExecuteInRunnable();
rp = new ProcessorRunner(reply.getBuffer());
eir.init(rp, req.getScope());
orb.orbScope.executeInArea(eir);
```

(b) A typical use of EIR

Figure 7: *The Execute In Runnable Idiom. This idiom is widely used in Zen. This example is taken from request processing code. At runtime the code (1) enters the common parent, (2) executes the run() method of the EIR, (3) then enters the acceptor scope, and finally (4) executes the application logic contained in the run() method of ProcessorRunner.*

The profile requires refactoring this code so that method calls can be delegated from one scope's gate to another, effectively following the Law of Demeter [16]. We thus avoid the need for ExecuteInRunnable objects and other runnables such as ProcessRunner. Fig. 8 illustrates the refactored version of the example. A new method has been added to a class (ORBImpl) defined in the parent package. The body of this method contains the contents of the ProcessorRunnable.run() method (in this case a single call to send(). The ProcessorRunnable class can be deleted and no EIR is needed.

```
package imm.orb;
  class ORBImpl ...
    imm.orb.acceptor.Transport transport;
    public void sendFromProcessor(WriteBuffer wb) {
        transport.send(wb);
    }
```

```
package imm.orb.requesprocessor;
  ...
  orb.sendFromProcessor(wb);
  ..
```

Figure 8: *Cross scope invocation using delegation.*

5.4.2 Thread Communication

In the RTSJ a thread entering a (new or existing) scope starts out without any reference to objects already allocated within that scope. It is often the case that different threads will enter the same scope and need to communicate by shared variables, or that a thread needs to store objects in a scope for future use. The RTSJ provides a single shared variable called a portal which can be used for this purpose.

```
class ProcessorRunner implements Runnable {
    WriteBuffer wb;     ProcessorRunner(WriteBuffer w) { wb = w; }
    public void run() {
        ScopedMemory mem = (ScopedMemory) RealtimeThread.getCurrentMemoryArea();
        ((Transport) mem.getPortal()).send(wb);
}   }
```

Figure 9: *Portals. In the RTSJ a portal is a shared variable for all threads within the same scope.*

Fig. 9 illustrates a typical use of portals. The ProcessorRunner class discussed earlier has a method which will be invoked within a scope different from the scope where the object was created. The code first obtain the current memory area, then obtains the portal object, which is expected to be an instance of Transport, then finally invokes the send() method.

In our case, this complexity can be avoided by simply storing such shared variables in the fields of the scope gate object. This has the advantage that programmers can defined multiple shared variables and give them meaningful names and types.

5.4.3 Scope Lifetime

The lifetime of objects allocated within a scope is limited to the time one or more threads are active in the scope. When all threads exit, all objects, including the portal, are deallocated. This behavior is inconvenient for scopes that need to remain live for longer periods of time. A rather inconvenient (and wasteful) way to extend the lifetime of objects is to use a, so-called, *wedge thread*: a real-time thread waits within the scope, keeping it alive. A wedge thread is essentially inert, and is only used to keep the scope active. Fig. 10 illustrates an example of this idiom in RTZen. The contents of the scope can be deleted by invoking notify() on the orb instance.

```
class ORBImpl { ...
  public ORBImpl(...) {
    thread =
        new NoHeapRealtimeThread(...scope...new Wedge(scope));
```

```
class Wedge ...
    public void run() {
        ORBImpl orb = (ORBImpl) scope.getPortal();
        synchronized (orb) {
            ...
            orb.wait();
            ...
```

Figure 10: *A wedge thread used to keep a scope alive.*

With the profile, the default behavior for a scope is to retain objects between invocations of the methods of the scope's gate. Thus we delete all wedge threads during the refactoring. The default RTSJ behavior can be obtained by annotating methods of a gate with @reclaim. Otherwise an explicit deallocation can be triggered by calling reset(). One feature that is not directly supported by the RTSJ is the @force annotation, this is useful in cases where a high-priority thread must be guaranteed access to an empty scope, even if some low priority thread is still executing within it. The method throws an asynchronous exception in the low-priority thread and reclaims the objects within the scope.

5.4.4 Borrowed Objects

While the RTSJ prevents assignment of objects allocated in a subscope to fields of objects allocated in a parent scope, it is legal to assign references to objects allocated in a subscope to a local variable. This means that in certain cases code executing in a parent scope may manipulate objects allocated in a subscope (or even a sibling scope). This actually safe – it is a form of borrowing [5] – because the subscope is pinned by the current thread. This pattern has been named the bridge pattern as it can be used to establish temporary communication channels between scopes. Great care must be taken when doing this as it is very easy to confuse the allocation context of objects. In the profile, a restricted version of the bridge pattern is allowed. It is legal to hand out reference to any object to another method if the corresponding argument has been declared as @borrowed. Fig. 11 illustrates an example of borrowing in RTZen.

```
rh = new Handler();
...
reply = servant.invoke(rm.getOperation().toString(), rh);
```

(a)

```
ServantProxy invoke(@borrowed String operation,
                    @borrowed Handler handler);
```

(b)

Figure 11: *An example of borrowing. (a) RTZen code for invoking servant methods in which local objects escape the current scope. (b) In Scoped Zen, this idiom is statically checked thanks to the @borrowed annotation on the invoke method.*

5.4.5 Scope Pools

The last common real-time processing idiom used within RTZen is the use of a pool of scoped memory objects. As any memory-consuming operation should be executed within its own short-lived, private, scope, it is convenient to keep a pool of scope that can be used to that end. Fig. 12 illustrates a simple use of a scope pool.

```
scope = orb.getScopedRegion();
eir = getEIR();
eir.init(request, servant);
scope.enter(eir);
orb.releaseScope(scope);
```

Figure 12: *Scope pools are used to avoid repeated creation of scopes.*

In our case, scoped memory areas are associated with gates. Thus in the refactored version of Zen, we use different pools of gate objects, one pool per type of gate.

5.5 Example: Scoped Zen Request Dispatching

We now proceed with an example that illustrates request processing in the refactored version of Zen. Fig. 13 shows simplified code from our implementation. The classes defined in the orb package include ORBImpl which implements the key methods of the ORB, and ServantProxy which delegates call to the application specific servant. The immediate subpackage of imm.orb is called requestprocessor and it contains a gate class called Processor and a Handler class for construct reply messages. A number of other classes such as WriteBuffer, Request and CDROutStream are defined in the imm.orb package but not shown here.

```
package imm.orb;
    class ORBImpl ... {
        public void sendFromProcessor(WriteBuffer buffer) { transport.send(buffer); }

    class ServantProxy {
        public void invoke(@borrowed String op, @borrowed Handler hl);



package imm.org.requestprocessor;
    public class Processor extends ScopeGate {
        private imm.orb.ORBImpl orb;
        public Processor(ORBImpl o) { orb = o; }
        @reclaim public void send(Request rm, Servant servant) {
            ...
            CDROutStream reply = servant.invoke(rm.getOperation().toString(), new Handler());
            orb.sendFromProcessor(reply.getBuffer());
        }

    public class Handler {
        public CDROutStream createReply() {
            CDROutStream out = CDROutStream.instance();
            out.write_octet_array(magic, 0, 4);
            ...
            return out;
        }
```

Figure 13: *Request Dispatching in Scoped Zen.*

Each processor instance is associated with a different scoped memory area. The following code fragment creates a processor pool of size 2:

```
Processor[] procs = new Processor[]{
        new Processor(new LTMemory(...)),
        new Processor(new LTMemory(...)) } ;
```

A processor can then be used by simply calling the send method:

```
processor.send(request, servant);
```

This example is much simpler than the corresponding RTZen code as we have removed 5 runnable classes. While refactoring this part of the the RTZen code we have uncovered one error: a method invoked from the request processor scope was performing lazy initialization and trying to assign to a field of an object allocated within the ORB. Depending on invocation order, this could result in a runtime exception.

6 Implementation

In order to run the Scoped Memory Profile on a standard RTSJ VM, a translator is requires to implement the implicit scope functionality provided by the gate objects; to ensure that objects allocated within gates remained until the gate was deleted or the scope reset; and to ensure objects were created within the correct scopes. This can be implemented by bytecode rewriting of the classes. For the purpose of our experiment we performed a hand translation.

A scope gate is allocated in a parent scope, but contains objects allocated within a child scope that existentially depends upon the gate object. This means we had to develop a translation that effectively allowed gate objects to occupy two RTSJ scopes simultaneously, and also to manage the RTSJ ScopedMemory meta-object that represented the scope. To solve this problem, we split each gate class into two RTSJ classes: an *outer gate* class and an *inner gate* class. The outer gate is allocated in the parent scope and stores the RTSJ scope meta-object for its inner scope, and a reference to the gate of its enclosing parent scope. The inner gate is then allocated within that inner scope, and stored in the scope's portal: objects stored in the (Scoped Types) gate object's fields are actually stored in fields of the (RTSJ) inner gate. When an outer gate is created it begins by constructing an instance of the child scope it will control access to, and then uses the Execute In Runnable idiom to create, initialise and install the actual inner gate object within that scope. The outer gate then acts as a Proxy [8] for the inner gate — it provides the full interface of the gate class, but implements those methods by using the RTSJ Execute In Runnable idiom to call the corresponding methods of the inner gate. In this way, program can call methods on the gate object as if they were normal Java methods, but they end up being executed in the correct scope context. Furthermore, because RTSJ prevents incoming references into inner scopes, code using the gate is unable to observe that the objects have been split in this fashion. The gate reset functionality is implemented by having the inner gate create a wedge thread to keep the scope from being reclaimed. This is essentially the same solution adopted by RTZen: our translation has the advantage that the code required to manage the thread is abstracted away from the programmer, who can simply use objects with similar data semantics to plain Java. The translation for method invocation is as follows. Methods defined in the same package are left intact. Methods defined in parent package require an explicit allocation context switch before invocation (essentially an executeInArea wrapper in RTSJ terms).

We have implemented a checker for the core scope visibility rules and run it on the hand-translated version of RT-Zen to validate the translation. We have also executed the translated version Zen on a stock VM and verified empirically that our refactoring was behavior preserving. Finally, since runtime memory checks are not needed in Scoped Zen, we modified the RTSJ VM to ignore these checks and ran the Zen with a variety of workloads. The results showed that removing memory checks improved performance by, on average, 10%.

7 Conclusion

This paper is the first step towards a comprehensive profile for high-integrity real-time Java systems. The rules proposed here ensure that an important category of runtime errors can be prevented statically. The profile is fully backwards compatible, it does not require changes to the development environment of virtual machine, and allows standard (non real-time) Java code to remain as is. We have performed a significant validation experiment by refactoring over 17'000 lines of RTSJ code. We expect the refactoring methodology described in this paper to be applicable to other systems.

References

[1] J. Barnes. *High integrity Ada: the SPARK approach*. Addison Wesley, 1997.

[2] William S. Beebee, Jr. and Martin Rinard. An implementation of scoped memory for real-time Java. In *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, 2001.

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

[4] Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, Jr., and Martin Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Proceedings of Conference on Programming Languages Design and Implementation*. ACM Press, 2003.

[5] John Boyland. Alias burying: Unique variables without destructive reads. *Software — Practice and Experience*, 31(6):533–553, May 2001.

[6] Alan Burns. The Ravenscar Profile. *ACM SIGADA Ada Letters*, 19(4):49–52, 1999.

[7] Dave Clarke and Sophia Drossopoulou. Ownership, encapsulation and the disjointness of type and effect. In *Proceedings of the 17th ACM conference on Object-oriented programming, systems, languages, and applications (OOPSLA02)*, 2002.

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

[9] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney. Region-based memory management in Cyclone. In *Proceedings of Conference on Programming Languages Design and Implementation*, pages 282–293, June 2002.

[10] H. Hetcht, M. Hecht, and S. Graff. Review guidelines for software languages for use in nuclear power plant systems. Technical Report NUREG/CR-6463, U.S. Nuclear Regulatory Commission, 1997.

[11] John Hogg. Islands: Aliasing protection in object-oriented languages. In *ACM Conference on Object-Oriented Programming Languages, Applications, Languages, and Systems (OOPSLA)*, volume 26, pages 271–285, New York, November 1991. ACM Press.

[12] Arvind Krishna, Douglas C. Schmidt, and Raymond Klefstad. Enchancing Real-time CORBA via Real-time Java features. In *International Conference on Distributed Computing Systems (ICDCS 2004)*, 2004.

[13] Arvind S. Krishna, Douglas C. Schmidt, Krishna Raman, and Raymond Klefstad. Optimizing the ORB core to enhance Real-time CORBA predictability and performance. In *Distributed Objects and Applications (DOA)*, 2003.

[14] Jagun Kwon and Andy Wellings. Memory management based on method invocation in RTSJ. In *OTM Workshops 2004, LNCS 3292, pp. 33–345*, 2004.

[15] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-Java: A high integrity profile for real-time Java. In *Joint ACM Java Grande/ISCOPE Conference*, November 2002.

[16] Karl J. Lieberherr. Controlling the complexity of software designs. In *IEEE International Conference on Software Engineering (ICSE)*, pages 2–11, 2004.

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

[18] Peter Puschner and Andy Wellings. A profile for high integrity real-time java programs. In *Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.

[19] 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*, 2001.

[20] Tian Zhao, James Noble, and Jan Vitek. Scoped Types for Realtime Java. In *International Real-Time Systems Symposium (RTSS 2004)*, Lisbon, Portugal, December 2004. IEEE.