

It is Time to Get Real with Real-Time: In Search of the Balance between Tools, Patterns and Aspects

Celina Gibbs, Yvonne Coady, Jan Vitek, Tian Zhao, James Noble, and Chris Andreae

Abstract

Increasing demands for real-time systems are vastly outstripping the ability for developers to robustly design, implement, compose, integrate, validate, and enforce real-time constraints. It is essential that the production of real-time systems take advantage of approaches that enable higher software productivity. Though many accidental complexities have proven to be time consuming and problematic – type errors, memory management, and steep learning curves – we believe a disciplined approach using tools, patterns, and aspects can help. But the question as to how to best strike a balance between these approaches remains.

This paper previews Scoped Types and Aspects for Real-Time Systems (STARS), an approach aiming to guide real-time software development. In this paper, the balance between tools, patterns and aspects in this programming model is explored, and tradeoffs associated with an early prototype are identified.

1 Introduction

The Real-Time Specification for Java (RTSJ) introduces abstractions through which developers must manage resources, such as non-garbage collected regions of memory [2]. The difficulty associated with managing the inherent complexity associated with these concerns ultimately compromises the development, maintenance and evolution of safety critical code bases and increases the likelihood of fatal memory access errors at runtime.

This paper considers key tradeoffs in the design of a programming model for real-time systems. The model integrates the RTSJ abstractions with tools for verification, a disciplined approach including patterns, and language features from AOP. *Scoped Types and Aspects for Real-Time Systems* (STARS), offers a programming environment we believe that is conducive to modern software development for real-time systems. Building on the work of Scoped Types [6], STARS both guides real-time development with a simplified Scoped Type discipline and provides much needed support for the modularization and verification of real-time constraints.

1.1 Background on RTSJ

RTSJ extends the Java memory management model to include dynamically checked region-based memory known as *scoped memory areas*. A scoped memory area is an allocation context, which provides a pool of memory for threads executing in it. Individual objects allocated in a scoped memory area cannot be deallocated. Instead, an entire scoped memory area can be collected as soon as all threads exit that area.

The RTSJ defines two predefined areas for *immortal* and *heap* memory represented by the Java classes `ImmortalMemory` and `HeapMemory`, respectively, for objects with unbounded lifetimes and objects that must be garbage collected. Scoped memory areas can be nested to form a dynamic, tree-shaped hierarchy, where child memory areas have strictly shorter lifetimes than their parent. Though this structure can be well defined in terms of design, it can be easily overlooked in an implementation, resulting in a dangling reference. Since a scoped memory area could be reclaimed at any time, dynamically enforced safety rules must include checks to ensure a memory area with a longer lifetime does not hold a reference to an object allocated in a memory area with a shorter lifetime.

1.2 Related Work: Scoped Types

Scoped types are one of the latest developments in the general area of type systems for controlled sharing of references [6]. The key insight of the scoped type work was the necessity to make the scope structure of the program explicit in order to have a tractable verification procedure. STARS builds on the contribution of scoped types and proposes that 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.

This paper investigates some of the tradeoffs involved in the design and implementation of the STARS programming model. The core decisions involve striking the right balance between aspects, patterns and tools in the development of real-time applications.

2 The STARS Programming Model

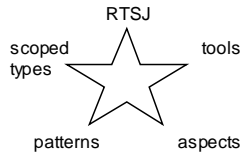


Figure 1: Elements of the STARS programming model.

STARS guides the design and implementation of real-time systems with a well defined, explicit programming model. Key elements of the model are overviewed here. Figure 1 shows that STARS consists of five elements: RTSJ, Scoped Types, aspects, patterns and tools.

When Scoped Types is coupled with a cleanly defined programming discipline, it provides static constraints and declarative specification. Patterns and aspects can provide controlled and explicit memory management. Tools of course can enhance each of these features as well as provide additional infrastructural support, such as static verification.

Static verification is important in order to save costs. For example, in a comparison between (1) a plain Java implementation, (2) an RTSJ implementation and (3) a Scoped Type implementation with static checking, on average the plain Java performs the best (due to the fact that it does not have to incur the cost of dynamic scope checks and does not have code to manage, enter and reclaim scoped memory areas). However, results show interference by the data reporting thread and garbage collection¹. The RTSJ version is more predictable but markedly, slower. The Scoped version retains the predictability of the RTSJ version but is faster due to the elimination of most runtime checks. The following subsections overview the current Scoped Typed model for memory scopes, and the static constraints that can be established based on that model.

2.1 Modeling memory scopes

STARS looks to leverage the abstractions proposed in the Scoped Types discipline to explicitly enforce as well as modularly support reasoning about the RTSJ.

We start by considering a key simplifying feature of Scoped Types. Rather than relying on RTSJ's implicit, dynamic notion of allocation context, i.e. the last entered memory area by the current thread, we need to enforce an explicit lexically-scoped discipline which guarantees that the relative location of any object is obvious from the program text. Equally as important, we must establish a simple, clear, and *static* memory

scope hierarchy in the program's code. That is, developers need a clear structural view of the memory area hierarchy of an application.

In an attempt to model the Scoped Types abstractions a restructuring of the programs package structure is required. Essentially, we equate Java packages to memory scopes. Nested packages model nested scopes. Because real-time code needs to coexist with standard Java code, we require all real-time code to be in packages nested within a package for *immortal* memory called *imm* — all the instances of classes in this package are in permanent, are not garbage collected, but are accessible from the standard Java system. Then, classes to be in scoped memory reside in scopes nested inside *imm*, where the static package nesting reflects the dynamic scope nesting at runtime.

Further, Scoped Types categorizes every class that will execute in immortal memory as either a *gate* or a *scoped class*. A scoped class is assigned to the memory scope it executes in, whereas a gate class turns scopes into first class entities, facilitating the ability for threads to enter and exit nested scopes explicitly. Each memory scope and consequently each package has just one gate class and all references from parent to child scope must proceed through it. Each gate class is therefore associated with a thread of execution and a memory scope that execution and further allocations will occur in explicitly dictated by the package it resides in.

Intuitively, this model enforces the invariant that every instance of gate class maps to a uniquely scoped memory area. Furthermore, every instance of a class defined in a scoped package P is allocated in the memory area of a gate defined in P . Operationally, whenever a method of a gate is invoked, the allocation context is switched to the scope associated with that gate. Objects allocated within a scoped package are allowed to refer to 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. At runtime, there is one scoped memory area (immortal memory) corresponding to package *imm*, and then as many scoped memory areas nested inside it as there are instances of the gate classes defined in *imm*'s immediate subpackages (and so on, down through other packages nested more deeply within *imm*).

2.2 Static Constraints

We now outline the rules that ensure static correctness of STARS programs. In the following we assume that a scoped package contains at least one gate class and zero or more scoped classes. The descendant relation on package is a partial order on packages.

¹ This cannot occur in the RTSJ version as the real-time thread cannot be interrupted by a lower priority thread.

Rule 1. An expression of a type T defined in scoped package P is visible to classes defined in P or to class defined in some package P' that is a descendant of P . An expression of a gate type G defined in package P is visible to classes defined in P' where P is a direct descendant of P' .

This first rule encodes the essence of the RTSJ access rules. Scoped packages are those that are descendants of the *imm* package. *Gate* classes are treated differently as they are handles used from a parent scope to access a memory area. They must be accessible (visible) to the code defined in the parent scope.

Rule 2. An expression of type T defined in scoped package P can be widened to type T' iff T' is defined in P .

Rule 2 is used to enforce structural properties on the object graph. By preventing types to be cast to arbitrary supertypes (in particular `Object`), it is possible to verify Rule 1 statically.

Rule 3. An method invocation of method m on an expression of scoped (gate) type T is valid iff type T implements m .

Rule 3 prevents reference leaks, 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 4. The constructor of class C defined in scope package P can only be invoked by a method defined in P .

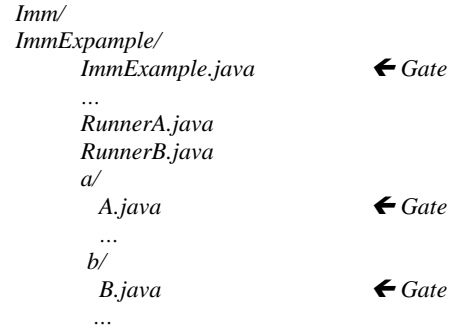
Rule 4 prevents a subpackage from invoking `new` on a class defined in a parent package. To do this, programmers should provide a factory method in the parent package.

Rule 5. A class C defined in a scoped package P may not have static reference fields.

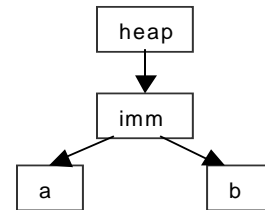
Rule 5 prevents objects of classes defined in the same scoped package (but with different gates at runtime) from communicating via static variables. This can result in dangling references as the gates have disjoint lifetimes.

2.3 Sample Scoped Types Structure

To get a sense of what the structure a scoped type program looks like, we overview an example here. The package structure includes *ImmExample* at the top level, and two subpackages *a* and *b*. The classes in this simple example are *Many*, *RunnerA* and *RunnerB*.



Which corresponds to the following scoped memory area structure, where the subpackages *a* and *b* are child scopes at the same level, and *ImmExample*, *A* and *B* are gates:



3 The STARS Prototype

A STARS prototype is currently being developed. Three key elements of the STARS model considered here are the tool that verifies the static constraints, patterns and idioms to manipulate scopes, and finally, an aspect-oriented translation that takes the scoped-type Java code and weaves in the necessary low-level real-time calls to execute the code on a real-time virtual machine. The following subsections consider these in turn, with an emphasis on the aspect-oriented issues. The software system used to demonstrate the STARS approach is modeling a real-time *collision detector* (or CD) consisting of two threads: (1) a real-time thread which periodically acquires data on position of aircraft from simulated sensors, and (2) a second thread that is low priority non-real-time thread responsible for updating the display. The system must detect collisions before they happen. The number of planes, airports, and nature of flight restrictions are all variables to the system.

The CD algorithm and base implementation was written by Filip Pizlo and Jason Fox using libraries written by Ben Titzer. The system is about 25K LOC and contains a mixture of plain Java and real-time Java. To provide a proof-of-concept for our proposal, we refactored the CD to abide by the previously described static constraints. The refactoring was done in three

stages. First, we designed a scope structure for the program based on the `ScopedMemory` areas used in the CD. Second, we redistributed classes between packages so that the Scoped CD package structure matches the scope structure. Third, we removed or replaced explicit RTSJ memory management idioms with equivalent constructs of our model.

3.1 Tools: Checking the Scoped Types Discipline

We must ensure that only programs following the Scoped Types discipline are accepted by STARS. The JavaCop “pluggable types” checker [1] verifies Scoped Types as a pluggable type. The key idea is that pluggable types layer a new static system over an existing language, and JavaCop can then check syntax-directed rules at compile time. JavaCop made it possible to leverage package structure in order to construct and check a relatively high-level description of the Scoped Type definitions.

3.2 Patterns and Idioms

RTSJ programmers have adopted a number of programming idioms to manipulate scopes. After changing the structure of the original CD, we needed to convert these idioms into corresponding idioms that abide by the rules established in Section 2.2. 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 metaobjects explicitly. In short, the patterns included two of those identified in [4] (*scoped run loop* and *multiscoped object*).

3.3 Aspect-Oriented Memory Management

Though the design of memory management in a real-time system may be clear, its implementation typically is not because it is inherently tangled throughout the code. For this reason we chose an aspect-oriented approach for modularizing scope management in a STARS program. This part of STARS is implemented using a subset of aspect-oriented programming extensions provided by AspectJ [5, 3].

After the program has been statically verified, aspects are composed with the plain Java base-level application. The aspects weave necessary elements of the RTSJ API into the system, and invoke some virtual machine specific extensions to ensure efficient management. This translation (and the aspects) depend critically upon the program following the Scoped Type discipline. If the rules are broken, the resulting program will no longer obey the RTSJ scoped memory discipline. As a result, either the program will fail at runtime with just the kind of an exception we aim to

prevent; or worse, if running on a virtual machine that omits runtime checks, fail in some unchecked manner.

Memory management aspects in STARS can be largely generated from information provided by the declarative specification for gates and scoped packages. For each gate class, the aspect introduces two fields: `memory` and `thread`, for the memory area in which the gate executes and the thread to which the gate is bound. Though these introductions can be automatically generated by having all gates implement the following interface:

```
public interface Gate {
    private MemoryArea memoryArea;
    private Thread thread;
}
```

the programmer must specifically customize some functionality, as the RTSJ memory hierarchy provides many choices for memory areas.

The `MemoryArea` class is the abstract parent of all classes representing memory. Its subclasses include `HeapMemory`, `ImmortalMemory`, and `ScopedMemory`. Both `HeapMemory` and `ImmortalMemory` have a singleton instance obtained by invoking the `instance()` method. The `ScopedMemory` class is also abstract and its subclasses `LTMemory` and `VTMemory` provide linear time and variable time allocation of objects in scoped memory areas respectively.

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.

For an example of the ways in which the precise functionality must be customized, in one case memory may be set simply as immortal without a bound thread:

```
this.memory = ImmortalMemory.instance();
```

whereas another gate may require both a memory area and a real-time thread:

```
this.memory =
    new LTMemory(Constants.MEMSIZE,
                 Constants.MEMSIZE);
this.thread =
    new RealtimeThread(
        new PriorityParameters(Constants.priority),
        null, null, null, null, this);
```

In order to efficiently manage the setting and getting of appropriate scoped memory allocation context during program execution, the STARS infrastructure relies upon two virtual machine methods customized for this purpose:

```

VM.setAllocForThread(ScopedMemoryArea memory);
ScopedMemoryArea VM.getAllocForThread();

```

More specifically, before any object is created (i.e., before all calls to *new*) or before any call to a gate method from a parent, the allocation context must be set accordingly. Subsequently, within any object initializer, the right context must be used. It is important to note that this is the only added VM support needed for STARS, and its use is localized within one STARS memory management aspect. The STARS infrastructure does provide a class *STARS* with some helper methods (such as *waitForNextPeriod* and *start()*) available for general purpose use.

In order to provide a high-level example of a memory management aspect of STARS, consider the sample code in Figure 2. In lines 6-9 the declare parents construct is used to identify gate classes and to extend the specialized thread class. The new fields required for each Gate class, memory and thread, are introduced starting on line 11. The around advice beginning on line 15 simply demonstrates the kind of functionality that the aspect can associate with all calls to *new* within the *detector* package. This code simply shows the new call proceeding to create an instance of a new object, and if that object is a gate, the memory field of the gate is set accordingly. As previously mentioned, this code relating to the specifics of the scoped memory area would need to be customized (not automatically generated) on a per gate basis.

```

1 import javax.realtime.*;
2
3 public aspect ScopeAspect {
4
5 // gate class
6 declare parents: App implements Gate;
7
8 // thread concerns
9 declare parents: App extends NoHeapRealtimeThread;
10
11 // fields for gate context
12 private ScopedMemoryArea Gate.memory;
13 private Thread Gate.thread;
14
15 // functionality associated with all calls to new
16 Object around(Object o):
17     call(detector.*.new(..) && this(o) {
18         Object newObj = proceed(o);
19         if (newObj instanceof Gate) {
20             Gate tmp = (Gate)newObj;
21             tmp.memory = new LTMemory(124000, 124000);
22         }
23     }
24 }

```

Figure 2. Sample memory management aspect.

4 Tradeoffs

In its current form the STARS approach impacts the logical 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.

```

1 import javax.realtime.*;
2
3 public aspect ScopeAspect {
4 declare parents: Mem extends IMM
5 declare parents: Cdmem extends Mem
6 declare parents: Main implements IMM;
7
8 // classes to be created in Mem Scope
9 declare parents: (App || Detector ||
10                 StateTable || Aircraft ||
11                 Position) implements Mem;
12 // class to be created in CDMem Scope
13 declare parents: Frame implements CDMem;
14
15 // gate class
16 declare parents: App implements Gate;
17
18 // thread concerns
19 declare parents: App extends NoHeapRealtimeThread;
20
21 // fields for gate context
22 private ScopedMemoryArea Gate.memory;
23 private Thread Gate.thread;
24
25 // functionality associated with all calls to new
26 Object around(Object o): call(detector.*.new(..)
27     && this(o) {
28     Object newObj = proceed(o);
29     if (newObj instanceof Gate) {
30         Gate tmp = (Gate)newObj;
31         tmp.memory = new LTMemory(124000, 124000);
32     }
33     return newObj;
34 }

```

Figure 3. New memory management aspect, introducing memory hierarchy for scope management.

It can be argued that this decomposition is natural; 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 unrealistic. But from a first principles perspective, the importance of a logical package structure cannot be underestimated in application development, maintenance and evolution. Could we offset this impact by escalating tool support, aspects, or both, in order to still claim the associated static guarantees?

One alternative to this overloading of package structure is the use of empty interfaces that would behave as low level flags to model the Scoped Types abstractions. That is, empty interfaces corresponding to each of an application's memory scope hierarchy are introduced, with the hierarchal relationship between those memory scopes paralleled with the *extends* relationship shown in line 4 of Figure 3. Intuitively, this enforces the single parent rule for memory areas establishing a root memory scope of *immortal memory* (*imm*). Further we can associate each class with a specific memory area

illustrated in lines 9-13 of Figure 3, statically reflecting the dynamic scope hierarchy similar to the result of the package restructuring described in Section 2.1.

Arguably, this alternative to package restructuring does not explicitly force the same safety guarantees, such as the assignment of a class to a memory scope. Just as with the implicit requirement of non-garbage collected classes to be assigned to packages within the immortal memory package, the requirement for those classes to implement the corresponding interface can possibly be overlooked.

5 Conclusions and Future Work

It is essential that the production of real-time systems take advantage of approaches that enable higher software productivity. Through a combined approach of tools, patterns and aspects, STARS attempts to strike a balance necessary to combat real problems associated with real-time, such as type errors, memory management, and steep learning curves.

The question of how to strike a balance between tools, patterns and aspects is a matter we are currently exploring. Depending on the balance, aspects can be used solely to introduce low level RTSJ mechanisms, or their role could be more comprehensive, as part of tool support and also pattern implementations. One of the determining factors in this work will be the performance impact of aspects in RT applications. We are currently exploring these metrics.

6 References

1. Chris Andreae, Shane Markstrum, Todd Millstein, and James Noble. Practical pluggable types for java. Submitted, December 2005.
2. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.
3. Gregor Kiczales, Erik Hilsdale, Jim Hugunin, Mik Kersten, Jeffrey Palm, and William G. Griswold. An overview of AspectJ. *Lecture Notes in Computer Science*, 2072:327–355, 2001.
4. Filip Pizlo, Jason Fox, David Holmes, and Jan Vitek. Real-time java scoped memory: design patterns and semantics. In *Proceedings of the IEEE International Symposium on Object-Oriented Real-Time Distributed Computing*, May 2004.
5. AspectJ Project. IBM, <http://www.eclipse.org/aspectj/>.
6. 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.