Scoped Types and Aspects for Real-Time Java

Chris Andreae[3], Yvonne Coady[1], Celina Gibbs[1],
James Noble[3], Jan Vitek[4], Tian Zhao[2]

(1) University of Victoria, CA. (2) University of Wisconsin–Milwaukee, USA.
(3) Victoria University of Wellington, NZ. (4) Purdue University, USA.

Abstract. Real-time systems are notoriously difficult to design and implement, and, as many real-time problems are safety-critical, their solutions must be reliable as well as efficient and correct. While higher-level programming models (such as the Real-Time Specification for Java) permit real-time programmers to use language features that most programmers take for granted (objects, type checking, dynamic dispatch, and memory safety) the compromises required for real-time execution, especially concerning memory allocation, can create as many problems as they solve. This paper presents Scoped Types and Aspects for Real-Time Systems (STARS) a novel programming model for real-time systems. Scoped Types give programmers a clear model of their programs' memory use, and, being statically checkable, prevent the run-time memory errors that bedevil models such as RTSJ. Our Aspects build on Scoped Types guarantees so that Real-Time concerns can be completely separated from applications' base code. Adopting the integrated Scoped Types and Aspects approach can significantly improve both the quality and performance of a real-time Java systems, resulting in simpler systems that are reliable, efficient, and correct.

1 Introduction

The Real-Time Specification for Java (RTSJ) introduces abstractions for managing resources, such as non-garbage collected regions of memory [4]. For instance, in the RTSJ, a series of *scoped memory* classes let programmers manage memory explicitly: creating nested memory regions, allocating objects into those regions, and destroying regions when they are no longer needed. In a hard real-time system, programmers must use these classes, so that their programs can bypass Java's garbage collector and its associated predictability and performance penalties. But these abstractions are far from abstract. The RTSJ forces programmers to face more low-level details about the behaviour of their system than ever before — such as how scoped memory objects correspond to allocated regions, which objects are allocated in those regions, how those the regions are ordered — and then rewards any mistakes by throwing dynamic errors at runtime. The difficulty of managing the inherent complexity associated with real-time concerns ultimately compromises the development, maintenance and evolution of safety critical code bases and increases the likelihood of fatal errors at runtime.

This paper introduces Scoped Types and Aspects for Real-Time Systems (STARS), a novel approach for programming real-time systems that shields developers from many accidental complexities that have proven to be problematic in practice. Scoped Types use a program's package hierarchy to represent the structure of its memory use, making clear where objects are allocated and thus where they are accessible. Real-Time

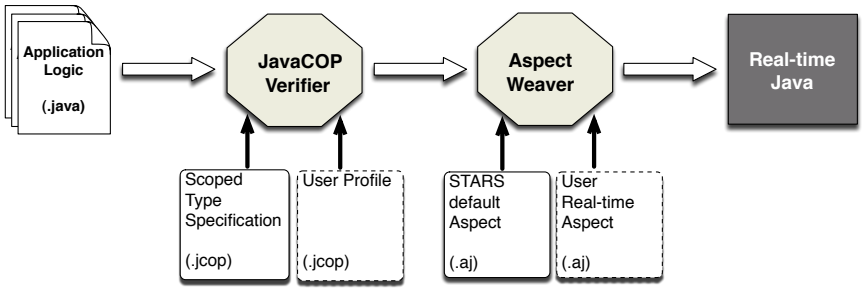


Fig. 1. *Overview of STARS. Application logic is written according to the Scoped Types discipline. The* JAVACOP *verifier uses scoped types rules (and possibly some user-defined application-specific constraints) to validate the program. Then, an aspect weaver combines the application logic with the real-time behaviour. The result is a real-time Java program that can be executed on any STARS-compliant virtual machine.*

Aspects then weave in allocation policies and implementation-dependent code — separating real-time concerns further from the base program. Finally, Scoped Types' correctness guarantees, combined with the Aspect-oriented implementation, removes the need for memory checks or garbage collection at runtime, increasing the resulting system's performace and reliability. Overall, STARS is a methodology that guides real-time development and provides much needed tool support for the verification and the modularization of real-time programs.

Fig. 1 illustrates the STARS methodology. Programmers start by writing application logic in Java with no calls to the RTSJ APIs. The code is then verified against a set of consistency rules — STARS provides a set of rules dealing with memory management; users may extend these rules with application-specific restrictions. If the program type checks, the aspects implementing the intended real-time semantics of the program can be woven into the code. The end result is a Real-time Java program which can be run in any real-time JVM which supports the STARS API.

The paper thus makes the following contributions:

1. **Scoped Types.** We use a lightweight pluggable type system to model hierarchical memory regions. Scoped Types is based on familiar Java concepts like packages, classes, and objects, can be explained with a few informal rules, and requires no changes to Java syntax.
2. **Static Verification** via the JAVACOP pluggable types checker [1]. We have encoded Scoped Types into a set of JAVACOP rules used to validate source code. We also show how to extend the built-in rules with application-specific constraints.
3. **Aspect-based real-time development.** We show how an aspect-oriented approach can decouple real-time concerns from the main application logic.
4. **Implementation in a real-time JVM.** We demonstrate viability of STARS with an implementation in the Ovm framework [2]. Only minor changes (18 lines of code in all) were needed to support STARS.
5. **Empirical evaluation.** We conducted a case study to show the impact STARS has on both code quality and performance in a 20 KLoc hard real-time application.

Refactoring RTSJ code to a STARS program proved easy and the resulting program enjoyed a 28% performance improvement over the RTSJ equivalent.

Compared with our previous work, STARS presents two major advances. First, Scoped Types enforce a per-owner relation [10, 18] via techniques based on Confined Types [9, 22]. The type system described here refines the system described in [21] which includes a proof of correctness, but no implementation. In fact, the refactoring discussed in that paper does not type check under the current type system. Secondly, the idea of using aspects to localize real-time behaviour is also new.

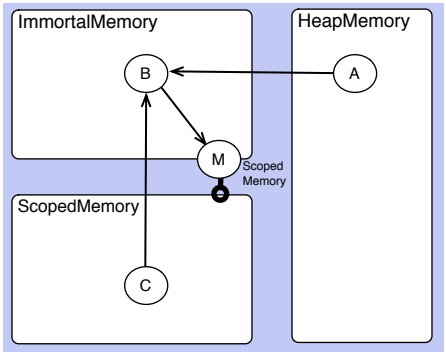
The paper proceeds as follows. After a survey of background and previous work, Section 2 presents an overview of the STARS programming model while Section 3 overviews the current STARS prototype implementations. Section 4 follows with a case study using STARS in the implementation of a real-time collision detection system. Finally we conclude with discussion and future work.

1.1 Background: The Challenges of Real-Time Memory Management

The Real-time Specification for Java (RTSJ) provides real-time extensions to Java that have shown to be effective in the construction of large-scale systems [2, 17, 20]. Two key benefits of the RTSJ are first, that it allows programmers to write real-time programs in a type-safe language, thus reducing opportunities for catastrophic failures; and second, that it allows hard-, soft- and non-real-time tasks to interoperate in the same execution environment. To achieve this second benefit, the RTSJ adopts a mixed-mode memory model in which garbage collection is used for non-real time activities, while manually allocated regions are used for real-time tasks. Though convenient, the interaction of these two memory management disciplines causes significant complexity, and consequently is often the culprit behind many runtime memory errors.

The problem, in the case of real-time tasks, is that storage for an allocation request (i.e. `new`) must be serviced differently from standard Java allocation. In order to handle real-time requests, the RTSJ extends the Java memory management model to include dynamically checked regions known as *scoped memory areas* (or also memory scopes), represented by subclasses of `ScopedMemory`. 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 is torn down as soon as all threads exit that scope. The RTSJ defines two distinguished scopes for *immortal* and *heap* memory, respectively for objects with unbounded lifetimes and objects that must be garbage collected. Two new kinds of threads are also introduced: *real-time* threads which may access scoped memory areas; and *no heap real-time* threads, which in addition are protected from garbage collection pauses, but which suffer dynamic errors if they attempt to access heap allocated objects.

Scoped memory areas provide methods `enter(Runnable)` and `executeInArea(Runnable)` that permit application code to execute within a scope, allocating and accessing objects within that scope. Using nested calls, a thread may enter or execute runnables in multiple scopes, dynamically building up the scope hierarchy. The differences between these two methods are quite subtle [4]: basically, `enter` must be used to associate a scope with a thread, whereas `executeInArea` (temporarily)



1. HeapMemory is garbage collected memory with no timeliness guarantees.
2. ImmortalMemory is not subject to reclamation.
3. ScopedMemory can be reclaimed in a single step if no thread is active in the area.
4. Immortal data can be referenced from any region. Scoped data can only be referenced from same scope or a nested scope. Violations lead to dynamic `IllegalAssignmentError`s.
5. `NoHeapRealtimeThread` cannot load heap references.

Fig. 2. *Memory Management in the Real-time Specification for Java.*

changes a thread's active scope to a scope it has previously `enter`ed. Misuse of these methods is punished by dynamic errors, e.g. a `ScopedCycleException` is thrown when a user tries to `enter` a `ScopedMemory` that is already accessible. Reference counting on `enter`s ensures that all the objects allocated in a scope are finalized and reclaimed when the last thread leaves that scope.

Real-time developers must take these memory scopes and threading models into account during the design of a real-time system. Scoped memory areas can be nested to form a dynamic, tree-shaped hierarchy, where child memory areas have strictly shorter lifetimes than their parents. Because the hierarchy is established dynamically, memory areas can move around within the hierarchy as the program runs. Dynamically enforced safety rules check that a memory scope with a longer lifetime does not hold a reference to an object allocated in a memory scope with a shorter lifetime. This means that heap memory and immortal memory cannot hold references to objects allocated in scoped memory, nor can a scoped memory area hold a reference to an object allocated in an inner (more deeply nested) scope. Once again, errors are only detected at runtime and are rewarded with dynamic errors or exceptions.

Given that safety and reliability are two goals of most real-time systems, the fact that these safety rules are checked *dynamically* seems, in retrospect, to be an odd choice. The only guarantee that RTSJ gives to a programmer is that their programs will fail in a controlled manner: if a dynamic assignment into a dynamically changing scope hierarchy trips a dynamic check, the program will crash with an `IllegalAssignmentError`.

1.2 Related Work: Programming with Scoped Memory

Beebee and Rinard provided one of the early implementations of the RTSJ memory management extensions [3]. They found it "close to impossible" to develop error-free real-time Java programs without some help from debugging tools or static analysis. The difficulty of programming with RTSJ motivated Kwon, Wellings and King to propose Ravenscar-Java [16], which mandates a simplified computational model. Their goal was to decrease the likelihood of catastrophic errors in mission critical systems. Further work along these lines transparently associates scoped memory areas with methods,

avoiding the need for explicit manipulation of memory areas [15]. Limitations of this approach include the fact that memory areas cannot be multi-threaded.

In contrast, systems like Islands [13], Ownership Types [10], and their successors restrict the scope of references to enable modular reasoning. The idea of using ownership types for the safety of region-based memory was first proposed by Boyapati et al. [5], and required changes to the Java syntax and explicit type annotations. Research in type-safe memory memory management, message-based communication, process scheduling and the file system interface management for Cyclone, a dialect of C, has shown that it is possible to prevent dangling pointers even in low-level codes [11]. The RTSJ is more challenging than Cyclone as scopes can be accessed concurrently and are first-class values.

Scoped types are one of the latest developments in the general area of type systems for controlled sharing of references [21]. This paper builds on Scoped Types and proposes a practical programming model targeting the separation of policy and mechanism within real-time applications. The key insight of Scoped Types is the necessity to make the nested scope structure of the program explicit: basically, every time the programmer writes an allocation expression of the form `new Object()`, the object's type shows where the object fits into the scope structure of the program. It is not essential to know which particular scope it will be allocated in, but rather the object's hierarchical relationship to other objects. This ensures that when an assignment expression, e.g. `obj.f=new F()`, is encountered, Scoped Types can statically (albeit conservatively) ensure that the assignment will not breach the program's scope structure.

2 The STARS Programming Model

STARS guides the design and implementation of real-time systems with a simple, explicit programming model. As the STARS name suggests, this is made up of two parts, Scoped Types, and Aspects. First, Scoped Types ensure that the relative memory location of any object is obvious in the program text. We use nested packages to define a *static* scope hierarchy in the program's code; a pluggable type checker ensures programs respect this hierarchy; at runtime, the dynamic scope structure simply instantiates this static hiearchy. Second, we use Aspect-Oriented Programming to decouple the real-time parts of STARS programs from their application logic. Aspects are used as declarative specifications of the real-time policies of the applications (the size of scoped memory areas or scheduling parameters of real time threads), but also to link Scoped Types to their implementations within a real-time VM.

The main points of the STARS programming model are illustrated in Fig. 3. The main abstraction is the *scoped package*. A scoped package is the static manifestation of an RTSJ scoped memory area. Classes defined within a scoped package are either *gates* or *scoped classes*. Every instance of a gate class has its own unique scoped memory area, and every instance of a scoped class will be allocated in the memory area belonging to a gate object in the same package. Because gate classes can have multiple instances, each scoped package can correspond to multiple scoped memory areas at runtime (one for each gate instance), just as a Java class can correspond to multiple instances. Then, the dynamic structure of the nested memory areas is modelled by the

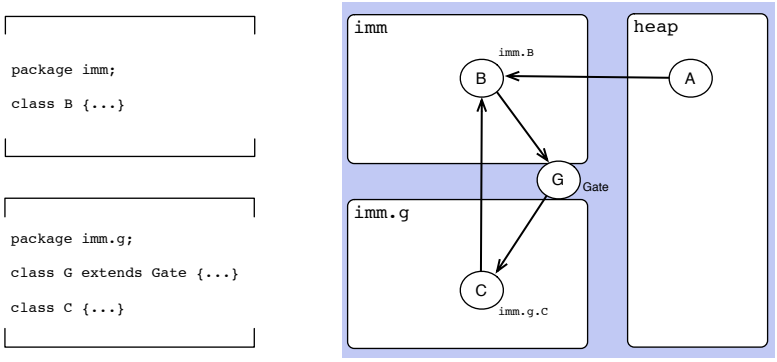


Fig. 3. *The STARS Programming Model. Each runtime scope has a corresponding Java package. Objects defined in a package are always allocated in a corresponding scope. A scope's Gate is allocated in its parent scope.*

static structure of the nested scoped packages, in just the same way that the dynamic structure of a program's objects is modelled by the static structure of the program's class diagram.

Scoped types are allowed to refer to types defined in an ancestor package, just as in RTSJ, objects allocated in a scope are allowed to refer to an ancestor scope: the converse is forbidden. The root of the hierarchy is the package `imm`, corresponding to RTSJ's immortal memory. There will be as many scoped memory areas nested inside the immortal memory area as there are instances of the gate classes defined in `imm`'s immediate subpackages.

STARS 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. In practice, STARS changes the way packages are used: 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 so grouping them in the same package is not unrealistic. We argue that this package structure is a small price to pay for STARS' static guarantees, and for the clarity it brings to programs' real-time, memory dependent code.

2.1 Scoped Types: Static Constraints

The following Scoped Types rules ensure static correctness of STARS programs. In this rules, we assume that a scoped package contains exactly one *gate class* and zero or more scoped classes or interfaces (the *scoped types*). By convention, the gate is named with the package's name with the first letter capitalized. The descendant relation on packages is a partial order on packages defined by package nesting. The distinguished

package `imm` is the root of the scope hierarchy. In the following we use `S` and `G` to denote respectively scoped and gate types, we use `C` to refer to any class. We use `p` to refer to the fully qualified name of a package. We refer to types not defined in a scoped package as *heap types*.

Rule 1 (Scoped Types).

1. *The package* `imm` *is a scoped package. Any package nested within a scoped package is scoped.*
2. *Any type not defined in a scoped package is a heap type.*
3. *The type of a gate class* `p.G` *defined within a scoped package* `p` *is a gate type.*
4. *The type of any non-gate interface or class* `p.S` *defined within a scoped package* `p` *is a scoped type. The type of an array with elements of scoped type is a scoped type.*

Rule 2 (Visibility).

1. *An expression of scoped type* `p.S` *is visible in any type defined in* `p` *or any of its subpackages.*
2. *An expression of gate type* `p.G` *is visible in any type defined in the immediate super-package of* `p`*. An exception to this rule is the local variable* `this` *which can be used within a gate class.*
3. *The type of the top-level gate* `imm.G` *is visible in heap types.*
4. *An expression of heap type is only visible in other heap types.*

The visibility rule encodes the essence of the RTSJ access rules. An object can be referenced from its defining memory area (denoted statically by a package), or from a memory area with shorter lifetime (a nested package). Gate classes are treated differently, as they are handles used from a parent scope to access a memory area. They must only be accessible to the code defined in the parent scope. The reason other types in the same scope package cannot refer to a gate is that we must avoid confusion between gates of the same type; a parent can instantiate many gates of the same type and the contents of these gates must be kept separate. Even though a gate's type is not visible in its own class, a single exception is made so that a gate object can refer to itself through the `this` pointer (because we know which gate "`this`" is).

Rule 3 (Widening). *An expression of a scoped type* `p.S` *can be widened only to another scoped type in* `p`*. An expression of a gate type* `p.G` *cannot be widened to any other types.*

Rule 3 is traditional in confined type systems where types are used to enforce structural properties on the object graph. Preventing types from being be cast to arbitrary supertypes (in particular `Object`) makes it possible to verify Rule 2 statically.

Rule 4 (Method Inheritance). *An invocation of some method* `m` *on an expression of scoped type* `p.S` *where* `p` *is a scoped package is valid if* `m` *is defined in a class* `p.S'` *in the same package. An invocation of a method* `m` *on an expression of gate type* `p.G` *is valid only if* `m` *is defined in* `p.G`*.*

Rule 4 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 one were to invoke a method inherited from a heap class.

Rule 5 (Constructor Invocation). *The constructor of a scoped class* `p.S` *can only be invoked by methods defined in* `p`.

Rule 5 prevents a subpackage from invoking `new` on a class that is allocated in a different area than the currently executing object. This rule is not strictly necessary, as an implementation could potentially reflect upon the static type of the object to dynamically obtain the proper scope. In our prototype, we use factory methods to create objects.

Rule 6 (Static Reference Fields). *A type* `p.S` *defined in a scoped package* `p` *is not allow to declare static reference fields.*

A static variable would be accessible by different instances of the same class allocated in different scopes.

2.2 Correctness

The fact that a package can only have one parent package trivially ensures that the RTSJ single parent rule will hold. Moreover, a scope-allocated object o may only reference objects allocated in the scope of o, or scopes with a longer lifetime, preventing any RTSJ `IllegalAssignmentError`. For example, suppose that the assignment `o.f=o'` is in the scope `s`, where `o` and `o'` have types `p.C` and `p'.C'` respectively. If `p.C` is a scoped type, then the rules above ensure that `o` and `o'` can only be allocated in `s` or its outer scopes. By Rules 2 and 3, the type of the field `f` is defined in `p'`, which is visible to `p.C`. Thus, the package `p'` is the same as or a super-package of `p` and consequently `o'` must be allocated in the scope of `o` or its outer scope. The same is true if `p.C` is a gate type, in which case `o` either represents `s` or a direct descendant of `s`. A formal soundness argument can be found in the extended version of this paper.

3 The STARS Prototype Implementation

The STARS prototype has two software components — a checker, which takes plain Java code that is supposed to conform to the Scoped Types discipline, and verifies that it does in fact follow the discipline, and an series of AspectJ aspects that weaves in the necessary low-level API calls to run on a real-time virtual machine.

3.1 Checking the Scoped Types Discipline

We must ensure that only programs that follow the scoped types discipline are accepted by the system: this is why we begin by passing our programs through a checker that enforces the discipline. Rather than implement a checker from scratch, we have employed

the JAVACOP "pluggable types" checker [1]. Pluggable types [6] are a relatively recent idea, developed as extensions of soft type systems [8] or as a generalization of the ideas behind the Strongtalk type system [7]. The key idea is that pluggable types layer a new static type system over an existing (statically or dynamically typed) language, allowing programmers to have greater guarantees about their programs' behaviour, but without the expense of implementing entirely new type systems or programming languages. JAVACOP is a pluggable type checker for Java programs — using JAVACOP, pluggable type systems are designed by a series of syntax-directed rules that are layered on top of the standard Java syntax and type system and then checked when the program is compiled. STARS is a pluggable type system, and so it is relatively straightforward to check with JAVACOP. The design and implementation of JAVACOP is described in [1].

The JAVACOP specification of the Scoped Type discipline is approximately 300 lines of code. Essentially, we provide two kinds of facts to JAVACOP to describe Scoped Types. First we define which classes must be considered scoped or gate types; and then we to restrict the code of those classes according to the Scoped Type rules.

Defining Scoped Types is relatively easy. Any class declared within the `imm` package or any subpackage is either a scoped type or a gate. Declaring a scoped type in the JAVACOP rule language is straightforward: a class or interface is scoped if it is in a scoped package and is not a gate. A gate is a class declared within a scoped package and with a name that case-insensitively matches that of the package. Array types are handled separately: an array is scoped if its element types are scoped.

```
1  declare gateNamed(ClassSymbol s){
2    require(s.packge.name.equalsIgnoreCase(s.name));
3  }
4  declare scoped(Type t){
5    require(!t.isArray);
6    require(!gateNamed(t.getSymbol));
7    require(scopedPackage(t.getSymbol.packge));
8  }
9  declare scoped(Type t){
10   require(t.isArray && scoped(t.elemtype));
11 }
12 declare gate(Type t){
13   require(!t.isArray);
14   require(gateNamed(t.getSymbol));
15   require(scopedPackage(t.getSymbol.packge));
16 }
```

The rule that enforces visibility constraints is only slightly more complex. The following rule matches on a class definition (line 1) and ensure that all types of all syntax tree nodes found within that definition (line 2) meet the constraints of Scoped Types. A number of types and syntactic contexts, such as Strings and inheritance declarations, are deemed "safe" (`safeNodes` on line 3, definition omitted) and can be used in any context. Lines 4-5 ensure that top level gates are only visible in the heap. Lines 7-8 ensure that a gate is only visible in its parent package. Lines 10-11 ensure that the visibility of a scoped type is limited to its defining package and subpackages. Lines 13-16 apply if `c`

is defined within a scoped package and ensure that types used within a scoped package are visible.

```
rule scopedTypesVisibilityDefn1(ClassDef c){
 forall(Tree t : c){
  where(t.type != null && !safeNode(t)){
   where(topLevelGate(t.type)){
    require(!scopedPackage(c.sym.packge)):
     warning(t,"Top level gate visible only in heap"); }
   where(innerGate(t.type)){
    require(t.type.getSymbol.packge.owner == c.sym.packge):
     warning(t,"gate visible only in immediate superpackage"); }
   where(scoped(t.type)){
    require(t.type.getSymbol.packge.isTransOwner(c.sym.packge)):
     warning(t,"type visible only in same or subpackage"); }
   where(scoped(c.sym.type)){
    require(scopedPackage(t.type.getSymbol.packge) ||
        specialPackage(t.type.getSymbol.packge)    ||
        visibleInScopedOverride(t)):
      warning(t,"Type not visible in scoped package."); }
  }
 }
}
```

We restrict widening of scoped types with the following rule. It states that if we are trying to widen a scoped type, then the target must be declared in the same scoped package, and if the type is a gate widening disallowed altogether. The `safeWidening-Location` predicate is an escape hatch that allows annotations that override the default rules.

```
rule scopedTypesCastingDef2(a <: b @ pos){
 where(!safeWideningLocation(pos)){
  where(scoped(a)){
    require(a.getSymbol.packge == b.getSymbol.packge) :
     warning(pos,"Illegal scoped type widening."); }
  where(gate(a)){
     require(b.isSameType(a)) :
       warning(pos,"May not widen gate."); }
 }
}
```

JAVACOP allows users to extend the Scoped Types specification with additional restrictions. It is thus possible to use JAVACOP to restrict the set of allowed programs further. The prototype implementation has one restriction, though, it does not support AspectJ syntax. JAVACOP is thus not able to validate the implementation of aspects. As long as aspects remain simple and declarative, this will not be an issue. But in the longer term we would like to see integration of a pluggable type checker with an Aspect language.

3.2 Aspects for Memory Management & Real-time

Though the design of memory management in a real-time system may be clear, typically, its implementation will be unclear, because it is inherently tangled throughout the code. For this reason we chose an aspect-oriented approach for modularizing scope management. This part of STARS is implemented using a (subset of) the Aspect-Oriented Programming features provided by AspectJ [14]. For performance, predictability and safety reasons we stay away from dynamic or esoteric features such as *cflow* and features that require instance-based aspect instantiation such as *perthis* and *pertarget*.

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. 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, and then either 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.

```
1  package scope;
2
3  public class STARS {
4    static public boolean waitForNextPeriod() { ... }
5    public @WidenScoped void runInThread(Runnable r) {}
6  }
7
8  public class Gate extends STARS {
9    private MemoryArea mem;
10 }
11
12 privileged abstract aspect ScopedAspect {
13   abstract pointcut InScope();
14   pointcut NewGate(Gate g) : execution(Gate+.new(..))
15                              && target(g)
16                              && InScope();
17   pointcut GateCall(Gate g) :
18                             execution(public void Gate+.*(..))
19                             && this(g);
20   pointcut RunInThread(Runnable r, STARS g) :
21                          execution(void STARS+.runInThread(..))
22                          && target(g)
23                          && args(r);
24   ...
25 }
```

Fig. 4. *STARS Interface. The* `scope` *package contains two classes,* `STARS` *and* `Gate`*, and an abstract aspect* `ScopedAspect`*. Every gate class inherits from* `Gate` *and has access to two methods* `waitForNextPeriod()` *and* `runInThread()`*. Every STARS aspect extends* `ScopedAspect`*, must define pointcut* `InScope` *and has access to a number of predefined pointcuts.*

STARS programs are written against a simple API, shown in Fig. 4. The use of the API is intentionally simple. Gate classes must extend `scope.Gate`, which gives access to only two methods: `waitForNextPeriod()`, which is used to block a thread until its next release event, and `runInThread()`, which is used to start a new real-time thread. The single argument of `runInThread` is an instance of class that implements the `Runnable` interface. The semantics of the method is that the argument's `run` method will be executed in a new thread. The characteristics of the thread are left unbound in the Java code.

STARS aspects must deal with two concerns: the specifics of the memory area associated with each gate and the binding between invocations of `runInThread()` and real-time threads. Specifying memory area parameters is done by declaring a `before` advice to the initialization of a newly allocated gate. The privileged nature of the aspect allows the assignment to the `Gate.mem` private field. The `ScopedMemory` class is abstract, the advice must specify one of its subclasses `LTMemory` and `VTMemory` which provide linear time and variable time allocation of objects in scoped memory areas respectively. It must also declare a size for the area.

```
1  before(Gate g): NewGate(g) && execution(MyGate.new(..)){
2    g.mem = new VTMemory( sz );
3  }
```

The above example shows an advice for class `MyGate`. The memory area associated has size `sz` and is of type `VTMemory`. The code can get more involved when `SizeEstimator`s are used to determine the proper size of the area.

It is noteworthy that the `mem` field is not accessible from the application logic as it is declared private. This means that memory areas are only visible from aspects. (As an aside, strict application of the scoped type discipline would preclude use of those classes in any case.)

3.3 Instrumentation and Virtual Machine Support

The implementation of STARS relies on a small number of changes to a real-time Java virtual machine. In our case, we needed only add 18 lines to the Ovm framework [2] and 105 of lines of AspectJ to provide the needed functionality.

The added functionality consists of the addition of three new methods to the abstract class `MemoryArea`. These methods expose different parts of the implementation of the `MemoryArea.enter()`. The `STARSenter()` method increments the reference count associated to the area, changes allocation context and returns an opaque reference to the VM's representation of the allocation context before the change. `STARSexit()` leaves a memory area, possible reclaiming its contents and restores the previous allocation context passed in as argument. `STARSrethrow()` is used to leave a memory area with an exception. Three methods of the class `LibraryImports` which mediates between the user domain and the VM's executive were made public. They are: `setCurrentArea()` to change the allocation context, `getCurrentArea()` to obtain the allocation context for the current thread, and `areaOf()` to obtain the area in which an object was allocated. All of these methods operate on opaque references.

```
1 Opaque MemoryArea.STARSenter();
2 void   MemoryArea.STARSrethrow(Opaque,Throwable);
3 void   MemoryAreaSTARSexit(Opaque area);
4
5 static Opaque LibraryImports.setCurrentArea(Opaque area);
6 static Opaque LibraryImports.getCurrentArea();
7 static Opaque LibraryImports.areaOf(Object ref);
```

We show two key advices from the `ScopedAspect` introduced in Figure 4. The first advice executes before the instance initializer of any scoped class or array (lines 1-4). This advice obtains the area of `o` – which is the object performing the allocation – and sets the allocation context to that area. The reasoning is that if we are executing a `new` then the target class must be visible. We thus ensure that it is co-located.

```
1 before(Object o): AllocInScope(o) {
2  return LibraryImports
3          .setCurrentArea(LibraryImports.areaOf(o));
4 }
```

We use the second advice to modify the behaviour of any call to a gate (recall that these can only originate from the immediate parent package). This `around` advice uses the memory region field of the gate to change allocation context. When the method returns we restore the previous area.

```
1 void around(Gate g) : GateCall(g) {
2  Opaque x = g.mem.STARSenter();
3   try {
4    try {
5     proceed(g);
6    } catch(Throwable e) { g.mem.STARSrethrow(x, e); }
7   } finally { g.mem.STARSexit(x); }
8  }
```

3.4 Extensions and Restrictions

We have found that, for practical reasons, a small numbers of adjustments needed to be made to the core of the scoped type system.

Intrinsics. Some important features of the standard Java libraries are presented as static methods on JDK classes. Invoking static methods from a scoped package, and especially ones that are not defined in the current package, is illegal. This is too restrictive and we relaxed the JAVACOP specification to allow calls to static methods in the following classes `System`, `Double`, `Float`, `Integer`, `Long`, `Math`, and `Number`. Moreover, we have chosen to permit the use of `java.lang.String` in scoped packages. Whether this is wise is debatable – for debugging purposes it is certainly useful to be able to construct messages, but it opens up an opportunity for runtime memory errors. It is conceivable that the JAVACOP rules will be tightened in the future to better track the use of scope allocated strings.

Exceptions. All subclasses of `Throwable` are allowed in a scoped package. This is safe within the confines of standard use of exceptions. If an exception is allocated and thrown within a scoped package, it is either caught by a handler within that package or escape out of the memory area. In which case it will be caught by the around advice at the gate boundary and `STARSrethrow` will allocate a RTSJ `ThrowBoundaryError` object in the parent scope and rethrow the newly allocated error. One drawback of this rule is that a memory error could occur if a programmer managed to return/assign a scope-allocated error object to a parent area. Luckily there is a simple solution that catches most reasonable use-cases. We define a JAVACOP rule that allows exceptions to be created only if they are within a `throw` statement.

```
1 declare treeVisInScoped(Tree t){
2  require(NewClass n, Throw th;
3        n <- env.tree && th<-env.next.tree){
4   require(th.expr == n);
5   require(t == n.clazz);
6 }
7 }
```

Annotations. We found that in rare cases it may be necessary to let users override the scoped type system — typically where (library) code is clearly correct, but where it fails the conservative Scoped Types checker. For this we provide two Java 5 annotations that are recognised by the JAVACOP rules. `@WidenScoped` permits to declare that an expression which performs an otherwise illegal widening is deemed safe. `@MakeVisible` takes a type and makes it visible within a class or method.

Reflection. In the current implementation we assume that reflection is not used to manipulate scoped types. But a better solution would be to have reflection enforce the STARS semantics. This can be achieved by making the implementation of reflection scope-aware. Of course, whether reflection should be used in a hard real-time system, considering its impact on compiler analysis and optimization is open for discussion.

Native methods. Native methods are an issue for safety. This is nothing new, even normal Java virtual machines depend on the correctness of the implementation of native methods for type safety. We take the approach that native methods are disallowed unless explicitly permitted in a JAVACOP specification.

Finalizers. While the STARS prototype allows finalizers, we advocate that they should not be used in scoped packages. This because there is a well-known pathological case where a `NoHeapRealtimeThread` can end up blocking for the garbage collector due to the interplay of finalization and access to scope by `RealtimeThread`s. This constraint is not part of the basic set of JAVACOP rules. Instead we add it as a user-defined extension to the rule set. This is done by the following rule:

```
1 rule nofinalizers(MethodDef m){
2  where(m.name.equals("finalize") && m.params.length == 0){
3    require(ClassSymbol c; c <- m.sym.owner) {
4     require(!scopedPackage(c.packge)):
5       warning(m,"Scoped class may not define a finalizer");
6    }
7  }
8 }
```

4 Case Study: A Real-time Collision Detector

We conducted a case study to demonstrate the relative benefits of STARS. The software system used in this experiment is modeling a real-time *collision detector* (or CD). The collision detector algorithm is about 25K Loc and was originally written with the Real-time Specification for Java. As a proof-of-concept for our proposal, we refactored the CD to abide by the scoped type discipline and to use aspects.

The architecture of the STARS version of the CD is given in Fig. 5. The application has three threads, a plain Java thread running in the heap to generate simulated workloads, a 5Hz thread whose job is to communicate results of the algorithm to an output device and finally a 10Hz `NoHeapRealtimeThread` which periodically acquires a data frame with positions of aircrafts from simulated sensors. The system must detect collision before they happen. The numbers of planes, airports, and nature of flight restrictions are variables to the system.

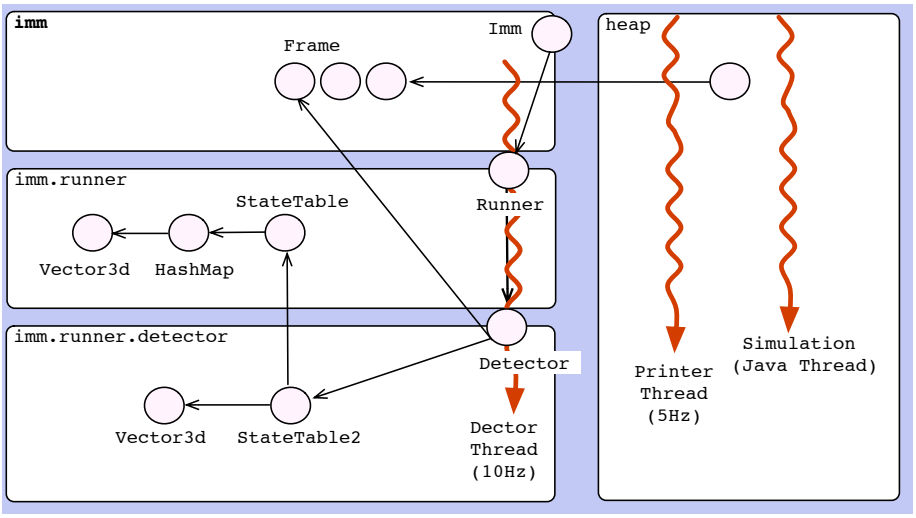


Fig. 5. *Collision Detector. The CD uses two scoped memory areas. Two threads run in the heap: the first simulates a workload, the second communicate with an output device. The memory hierarchy consists of* `imm` *(immortal memory) for the simulated workload,* `imm.runner` *for persistent data, and* `imm.runner.detector` *for frame specific data.*

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 moved classes amongst packages so that the STARS-CD package structure matched the scope structure. Third, we removed or replaced explicit RTSJ memory management idioms with equivalent constructs of our model.

Fig. 6 compares the package structure of the two versions. In the original CD the packages `atc` and `command` were responsible of computing trajectories based on a user-defined specification. They were not affected by the refactoring. Package `detector` contained all of the RTSJ code as well the program's `main()`. Finally `util` contained a number of general purpose utility classes. We split the code in the `detector` package in four groups. The package `heap` contains code that runs in the heap–this is the main and the data reporting thread. The package `imm` contains classes that will be allocated in immortal memory and thus never reclaimed. Below immortal memory there is one scope that contains the persistent state of the application, we defined a package `imm.runner` for this. The main computation is done in the last package, `imm.runner.detector`. This is the largest real-time package which contains classes that are allocated and reclaimed for each period.

The entire code of the real-time aspect for the CD is given in Fig. 7. This aspect simply declares the memory area types for the `imm.runner` and `imm.runner.detector` gates. Then it gives an around advice that specifies that the thread used by the CD algorithm is a `NoHeapRealtimeThread` and gives appropriate scheduling and priority parameters.

The overall size of the Scoped CD has increased because we had to duplicate some of the utility collection classes. This duplication is due to our static constraints. A number of collection classes were used in the `imm.runner` package to represent persistent state, and in the `imm.runner.detector` package to compute collisions. While we could have avoided the duplication by fairly simple changes to the algorithm and the use of problem specific collections, our goal was to look at the 'worst-case' scenario, so we tried to make as few changes to the original CD as possible. The methodology used to duplicate collection classes is straightforward: we define a scoped replacement for the `Object` class and replace all occurrences of `Object` in the libraries with the scoped

CD packages	classes per package	**Scoped CD packages**	classes per package
`atc`	989	`atc`	989
`command`	21198	`command`	21198
`util`	927	`util`	927
`detector`	1041		
		`heap`	105
		`imm`	120
		`imm.runner`	162
		`imm.runner.detector`	1587
		`collections`	8322

Fig. 6. *Package structure of the CD (left) and the STARS CD (right).*

```
1  privileged aspect CDAspect extends ScopedAspect{
2
3   before(Gate g): NewGate(g) && execution(Runner.new(..)){
4     g.mem = new LTMemory(Constants.SIZE*2,Constants.SIZE*2);
5   }
6
7   before(Gate g): NewGate(g) && execution(Detector.new(..)){
8     g.mem = new LTMemory(Constants.SIZE);
9   }
10
11  void around(STARS g, Runnable r): RunInThread(r, g){
12     Thread t = new NoHeapRealtimeThread(
13         new PriorityParameters(Constants.PRIORITY),
14         new PeriodicParameters(null,
15                 new RelativeTime(Constants.PERIOD, 0),
16                 null, null, null),
17         null, ((Gate) g).mem, null, r);
18     t.start();
19  }
20 }
```

Fig. 7. *Real-time Aspect for the CD. The aspect specifies the characteristics of memory areas as well as that of the real-time thread used by the application. The CD logic does not refer to any of the RTSJ APIs.*

variant. There were some other minor changes, but these were also fairly straightforward.

4.1 Patterns and Idioms

RTSJ programmers have adopted or developed a number of programming idioms to manipulate scopes. After changing the structure of the original CD, 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.

Scoped Run Loop. At the core of the CD is an instance of the ScopedRunLoop pattern identified in [19]. The `Runner` class creates a `Detector` and periodically executes the detector's `run()` method within a scope. Fig. 8 shows both the RTSJ version and the STARS version. In the RTSJ version, the runner is a `NoHeapRealtimeThread` which has in its `run()` method code to create a new scoped memory (lines 11-12) and a run loop which repeatedly enters the scope passing a detector as argument (lines 17-18).

In the STARS version, `Runner` and `Detector` are gates to nested packages. Thus the call to `run()` on line 16 will enter the memory area associated with the detector. Objects allocated while executing the method are allocated in this area. When the

method returns these objects will be reclaimed. Fig. 9 illustrates how a `Runner` is started. In the RTSJ version a scoped memory area is explicitly created (lines 2-3) and the real-time arguments are provided (lines 6-11). In the STARS version most of this is implicit due to the fact that a runner is a gate and the use of the `runInThread()` method which is advised to create a new thread. What should be noted here is that STARS clearly separates the real-time support from the non-real-time code. In fact we can define an alternative aspect which allows the program to run in a standard JVM.

```
1  public class Runner extends
2         NoHeapRealtimeThread {
3
4   public Runner(
5           PriorityParameters r,
6           PeriodicParameters p,
7           MemoryArea m) {
8    super(r, p, m);
9   }
10  public void run() {
11   final LTMemory cdmem =
12    new LTMemory(CDSIZE,CDIZE);
13   StateTable st =
14    new StateTable();
15   Detector cd =
16    new Detector(st, SIZE);
17    while (waitForNextPeriod())
18     cdmem.enter(cd);
19  }
20 }
```

```
1  public class Runner
2         extends Gate {
3
4
5
6
7
8
9
10  public void run() {
11   StateTable st =
12     new StateTable();
13   Detector cd =
14     new Detector(st, SIZE);
15   while (waitForNextPeriod())
16     cd.run();
17  }
18 }
19
20
```

Fig. 8. *Scoped Run Loop Example. The Runner class: RTSJ version (on the left) and Scoped version (on the right).*

```
1  public void run() {
2   LTMemory memory =
3    new LTMemory(MSZ, MSZ);
4   NoHeapRealtimeThread rt =
5   new Runner(new PriorityParameters(P),
6    new  PeriodicParameters(null,
7     new RelativeTime(PER,0),
8     new RelativeTime(5,0),
9     new RelativeTime(50,0),
10    null,null),
11    memory);
12  rt.start();
13 }
```

```
1   public void run() {
2     Runner rt =
3      new Runner();
4     runInThread(rt);
5   }
6
7
8
9
10
11
12
13
```

Fig. 9. *Starting up. The* `imm.Imm.run()` *method: RTSJ version (left-hand side) and Scoped version (right-hand side).*

Multiscoped Object. A multiscoped object is an object which is used in several allocation contexts as defined in [19]. In the RTSJ CD the `StateTable` class keeps persistent state and is allocated in the area that is not reclaimed on each period. This table has one entry per plane holding the plane's call sign and its last known position. There is also a method `createMotions()` invoked from the transient scope. The class appears in Fig. 10.

This code is particularly tricky because the state table object is allocated in the persistent area and the method `createMotions()` is executed in the transient area (when called by the `Detector`). The object referred to by `pos` (line 8) is transient and one must be careful not to store it in the parent scope. When a new plane is detected, `old` is null (line 11) and a new position vector must be added to the state table. The complication is that at that point the allocation context is that of the transient area, but the `HashMap` was allocated in the persistent scope (line 2). So we must temporarily

```
1  class StateTable {
2   HashMap prev = new HashMap();
3   Putter putter = new Putter();
4
5   List createMotions(Frame f) {
6    List ret = new LinkedList();
7    for (...) {
8     Vector3d pos = new Vector3d();
9     Aircraft craft = iter.next(newpos);
10    ...
11    Vector3d old = (Vector3d) prev.get(craft);
12    if (old == null) {
13     putter.c = craft;
14     putter.v = pos;
15     MemoryArea current =
16       MemoryArea.getMemoryArea(this);
17     mem.executeInArea(putter);
18    }
19   }
20   return ret;
21  }
22
23  class Putter implements Runnable {
24   Aircraft c;
25   Vector3d v;
26   public void run() {
27     prev.put(c, new Vector3d(v));
28   }
29  }
30 }
```

Fig. 10. *RTSJ StateTable. This is an example of a RTSJ multiscoped object – an instance of class allocated in one scope but with some of its methods executing in a child scope. Inspection of the code does not reveal in which scope* `createMotions()` *will be run. It is thus incumbent on the programmer to make sure that the method will behave correctly in any context.*

change allocation context. This is done by defining an inner class whose sole purpose is to create a new vector and add it to the hash map (lines 23-39). The context switch is performed in lines 15-17 by first obtaining the area in which the StateTable was allocated, and finally executing the `Putter` in that area (line 17). This code is a good example of the intricacy (insanity?) of RTSJ programming.

The scoped solution given in Fig. 11 makes things more explicit. The `StateTable` class is split in two. One class, `imm.runner.StateTable`, for persistent state and a second class, `imm.runner.detector.StateTable2` that has the update method. This split makes the allocation context explicit. A `StateTable2` has a reference to the persistent state table. The `createMotions()` method is split in two parts, one that runs in the transient area (lines 23-30) and the other that performs the update to the persistent data (lines 8-14).

Since our type system does not permit references to subpackages the arguments to `StateTable.put()` are primitive. The most displeasing aspect of the refactoring is

```
1  package imm.runner;
2  public class Vector3d { ... }
3
4  public class StateTable {
5   HashMap prev = new HashMap();
6
7   void put(Aircraft craft, float x, float y, float z) {
8    Vector3d old = prev.get(craft);
9    if (old==null)
10     prev.put(craft, new Vector3d(x, y, z));
11   else
12     old.set(x, y, z);
13  }
14 }
15
16 package imm.runner.detector;
17 class Vector3d { ... }
18
19 class StateTable2 {
20  StateTable table;
21
22  List createMotions(Frame f) {
23   List ret = new LinkedList();
24   for (...) {
25    Vector3d pos = new Vector3d();
26    ...
27    table.put(craft, pos.x, pos.y, pos.z);
28   }
29   return ret;
30  }
31 }
```

Fig. 11. *STARS StateTable. With scoped types the table is split in two. This makes the allocation context for data and methods explicit.*

that we had to duplicate the `Vector3d` class - there are now two identical versions - in each `imm.runner` and `imm.runner.detector`. We are considering extensions to the type system to remedy this situation.

4.2 Performance Evaluation

We now compare the performance of three versions of the CD: with the RTSJ, with STARS, and with a real-time garbage collector. The latter was obtained by ignoring the STARS annotations, with all objects allocated in the heap. The benchmarks were run on an AMD Athlon(TM) XP1900+ running at 1.6GHz, with 1GB of memory. The operating system is Real-time Linux with a kernel release number of 2.4.7- timesys-3.1.214. We rely on AspectJ 1.5 as our weaver. We use the Ovm virtual machine framework [2] with ahead-of-time compilation ("engine=j2c, build=run"). The GCC 4.0.1 compiler is used for native code generation. The STARS VM was built with dynamic read and write barriers turned off. The application consists of three threads, 10Hz, 5Hz, and plain Java. Priority preemptive scheduling is performed by the RTSJVM.

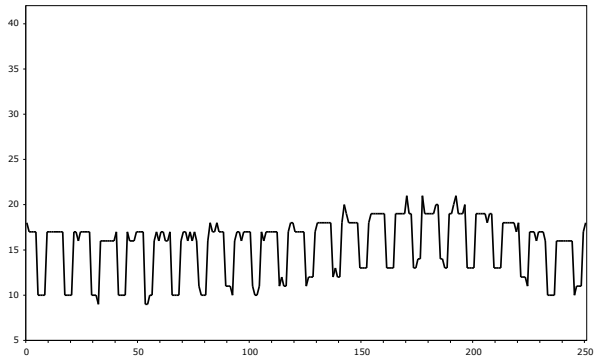
Fig. 12 shows the difference in running time between the three versions of the CD. Some of the variation is due to the workloads – collisions require more computational resources.

The results suggest that STARS outperforms both RTSJ and Real-time GC. On average, STARS is about 28% faster per frame than RTSJ and RTGC. This means that the overhead of before advice attached to every allocation is negligible. This is only a single data point, we feel that more aggressive barrier elimination could reduce the overhead of RTSJ programs and that the performance of our RTGC is likely not yet optimal. Nevertheless, the data presented here suggested that there is a potentially significant performance benefit in adopting STARS.

5 Discussion and Future Work

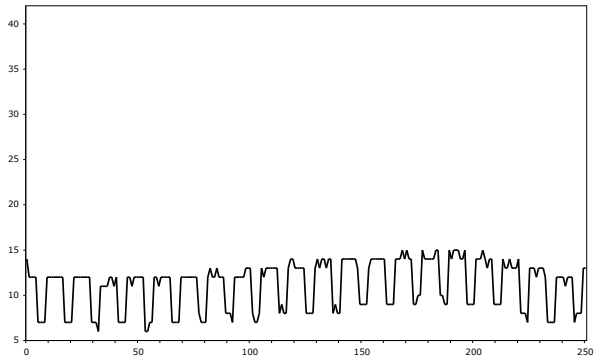
The combination of Scoped Types with Aspects is a promising means of structuring policy with its corresponding mechanism. When a real-time program is in this form, we can get the benefit of high level abstractions along with increased flexibility of their key mechanisms as aspects. The approach further allows for flexible combinations of lightweight static verification. The prototype implementation of STARS shows that the benefits of our approach can be obtained using mostly off-the-shelf technologies, in particular, existing aspect-oriented languages and static verifiers, with minimal changes to a real-time Java virtual machine. There is also potential for significant performance improvements. In our benchmark we have seen that a STARS program may run 28% faster than the corresponding RTSJ program.

This work has illustrated how aspects can extract and localize real-time concerns. In our case study the entire real-time specific portion of the application could be extracted as a simple declarative aspect. But the STARS interface is intentionally spartan and covers only part of the Real-time Specification for Java API. We hope that our approach can be extended to address a much larger set of real-time applications.



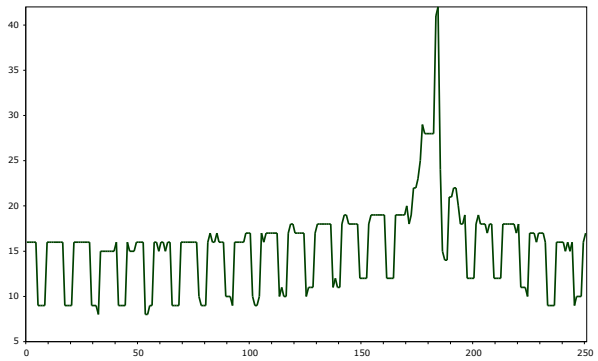
(a) RTSJ
with scope checks

median	17
avg	15.4
max	21



(b) STARS
no scope checks

median	12
avg	11.2
max	15



(a) Real-time GC
with scope checks

median	16
avg	15.4
max	42

Fig. 12. *Performance Evaluation. Comparing the performance of the collision detection implemented with (a) RTSJ, (b) STARS and (c) Java with a real-time garbage collector. We measure the time needed to process one frame of input by a 10Hz high-priority thread. The x-axis shows input frames and the y-axis processing time in milliseconds. The RTGC spikes at 43ms when the GC kicks in. No deadlines are missed. The average per frame processing time of STARS is 28% less than that of RTSJ and RTGC. Variations in processing time are due to the nature of the algorithm.*

One of the advantages of STARS is its truly lightweight type system. So lightweight, in fact, that one only needs make a judicious choice of package names to denote nesting of memory regions. The attraction is that no changes are needed in the language and tool chain, and that the rules are simple to explain. We do not attempt to sweep the costs of adopting STARS under the rug. As we have seen in the case study, there are cases where we had to change interfaces from objects to primitive types, thus forfeiting some of the software engineering benefits of Java. We were forced to duplicate the code of some common libraries in order to abide by the rules of scoped types. While there are clear software engineering drawbacks to code duplication, the actual refactoring effort in importing those classes was small. With adequate tool support the entire refactoring effort took less than a day. The hard part involved discovering and disentangling the scope structure of the programs that we were trying to refactor.

The benefits in terms of correctness can not be overemphasized. Every single practitioner we have met has remarked on the difficulty of programming with RTSJ-style scoped memory. In our own work we have encountered numerous faults due to incorrect scope usage. As a reaction against this complexity many RTSJ users are asking for real-time garbage collection. But RTGC is not suited for all applications. In the context of safety critical systems a number of institutions are investigating restricted real-time 'profiles' in which the flexibility of scoped memory is drastically curtailed [12]. But even in those proposals, there are no static correctness guarantees. Considering the cost of failure, the effort of adopting a static discipline such as the one proposed here is well justified.

We see several areas for future work. One direction is to increase the expressiveness of the STARS API to support different kinds of real-time systems and experiment with more applications to further validate the approach. Another issue to be addressed is to extend JAVACOP to support AspectJ syntax. In the current system, we are not checking aspects for memory errors. This is acceptable as long as aspects remain simple and declarative, but real-time aspects may become more complex as we extend STARS, and their static verification will become a more pressing concern. Finally we want to investigate extensions to the type system to reduce, or eliminate, the need for code duplication.

Acknowledgments. This work is supported in part by the National Science Foundation under Grant No. 0509156, and in part by the Royal Society of New Zealand Marsden Fund. Filip Pizlo and Jason Fox implemented the Collision Detector application, Ben Titzer wrote supporting libraries. We are grateful to David Holmes and Filip Pizlo for their comments, and to the entire Ovm team at Purdue.

References

1. Chris Andreae, James Noble, Shane Markstrum, and Todd Millstein. A framework for implementing pluggable type systems. Submitted, March 2006.
2. Jason Baker, Antonio Cunei, Chapman Flack, Filip Pizlo, Marek Prochazka, Jan Vitek, Austin Armbuster, Edward Pla, and David Holmes. A real-time Java virtual machine for avionics. In *Proceedings of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS 2006)*. IEEE Computer Society, 2006.

3. 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.
4. Greg Bollella, James Gosling, Benjamin Brosgol, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley, June 2000.
5. Chandrasekhar Boyapati, Alexandru Salcianu, William Beebee, and Martin Rinard. Ownership types for safe region-based memory management in real-time Java. In *ACM Conference on Programming Language Design and Implementation*, June 2003.
6. Gilad Bracha. Pluggable type systems. In *OOPSLA 2004 Workshop on Revival of Dynamic Languages*, 2004.
7. Gilad Bracha and David Griswold. Strongtalk: Typechecking smalltalk in a production environment. In *In Proc. of the ACM Conf. on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA)*, September 1993.
8. Robert Cartwright and Mike Fagan. Soft typing. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 278–292. ACM Press, 1991.
9. Dave Clarke, Michael Richmond, and James Noble. Saving the world from bad Beans: Deployment-time confinement checking. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages, and Appplications (OOPSLA)*, Anaheim, CA, November 2003.
10. David G. Clarke, John M. Potter, and James Noble. Ownership types for flexible alias protection. In *OOPSLA '98 Conference Proceedings*, volume 33(10) of *ACM SIGPLAN Notices*, pages 48–64. ACM, October 1998.
11. 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.
12. HIJA. European High Integrity Java Project. `www.hija.info.`, 2006.
13. John Hogg. Islands: Aliasing Protection in Object-Oriented Languages. In *Proceedings of the OOPSLA '91 Conference on Object-Oriented Programming Systems, Languages and Applications*, 1991.
14. 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.
15. Jagun Kwon and Andy Wellings. Memory management based on method invocation in RTSJ. In *OTM Workshops 2004, LNCS 3292, pp. 33–345*, 2004.
16. 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.
17. NASA/JPL and Sun. Golden gate. `research.sun.com/projects/goldengate`, 2003.
18. James Noble, John Potter, and Jan Vitek. Flexible alias protection. In *Proceedings of the 12th Eurpean Conference on Object-Oriented Programming (ECOOP)*, Brussels, Belgium, July 1998.
19. 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.
20. 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.
21. 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.
22. Tian Zhao, Jens Palsberg, and Jan Vitek. Type-based confinement. *Journal of Functional Programming*, 16(1), January 2006.