# Scoped Types for Real-time Java

Tian Zhao
University of Wisconsin – Milwaukee

James Noble
Victoria University of Wellington

Jan Vitek
Purdue University

***Abstract*— A memory model based on scoped areas is one of the distinctive features of the Real-Time Specification for Java (RTSJ). Scoped Types ensure timely reclamation of memory and predictable performance. The price to pay for these benefits is an unfamiliar programming model that, at the same time, is complex, requires checking all memory accesses, and rewards design-time errors with run-time crashes. We investigate an alternative approach, referred to as *Scoped Types*, that simplifies the task of managing memory in real-time codes. The key feature of our proposal is that the run-time partition of memory imposed by scoped areas is straightforwardly mirrored in the program text. Thus cursory inspection of a program reveals which objects will inhabit the different scopes, significantly simplifying the task of understanding real-time Java programs. Moreover, we introduce a type system which ensures that no run-time errors due to memory access checks will occur. Thus a RTSJ-compliant virtual machine does not require memory access checks. The contributions of this paper are the concept of Scoped Types, and a proof soundness of the type system. Experimental results will be described in future work.**

## I. Introduction

The Real-Time Specification for Java (RTSJ) [1] is designed to allow the construction of large scale real-time systems in a type-safe programming language. The benefits of using Java for mission critical systems are currently being evaluated by, e.g., Boeing [2] and JPL [3]. As of this writing a high-quality commercial implementation of the specification has been released by TimeSys (JTime [4]) and an open source virtual machine is being developed at Purdue (Ovm [5]). While the RTSJ includes many necessary features, the one that is most likely to affect how real-time Java programs are written is the new memory management model based on *scoped memory areas*.

An obvious concern for meeting hard real-time constraints in Java is the interaction of automatic memory management with real-time tasks. While garbage collection (GC) frees the programmer from the burden of tracking memory usage, it introduces unpredictability because the timing and duration of GC pauses is unknown. To address this problem, the RTSJ provides: (*a*) regions of memory which are not subject to garbage collection, called scoped memory areas, or *scopes*, (*b*) threads that can not access the heap and thus can never interfere with, or be preempted by, the garbage collector. Scoped memory areas provide predictable allocation, and ensure that hard real-time threads will not block while memory is being reclaimed.

In principle, scoped memory is similar to a cactus-stack [6], [7]. Each scope provides a pool of memory that can be used to allocate objects. Individual objects cannot be deallocated, instead the entire scope is reclaimed when its contents become unreachable. The main difference with stack allocation is that scopes are first-class entities which can be entered by multiple threads. The order in which threads enter scopes induces a runtime structure on scopes that determines permissible reference patterns. When a real-time thread executing in scope $\mathtt{M}_1$ first enters scope $\mathtt{M}_2$, scope $\mathtt{M}_1$ becomes the parent of $\mathtt{M}_2$. RTSJ semantics guarantees that $\mathtt{M}_2$ will be reclaimed before $\mathtt{M}_1$, the lifetime of a nested scope is thus always shorter than that of its parent. Threads executing in an scope allocate objects from the same pool of memory and communicate though shared variables. When the last thread exits a scope, the objects allocated within it are reclaimed. The last-in first-out natures of scoped memory allows for objects allocated in nested scopes, *e.g.* $\mathtt{M}_2$, to refer objects in their parent, *e.g.* $\mathtt{M}_1$, but not the converse as holding on to a reference into a shorter-lived scope may lead to dangling references, and jeopardize type safety.

To ensure type safety of real-time Java programs, the following invariants must be maintained at runtime:

1) Because a scope can be reclaimed at any time, an outer scope may not hold a reference to an object within an inner scope.
2) To avoid cycles in the scope parent relation, the nesting structure of scopes is restricted to trees. In other words, a scope may have at most one parent.
3) Because scopes can be shared by multiple threads, objects allocated within a scope cannot be discarded until all threads have finished using that scope.

Maintaining these invariants impose burdens on programmers and increases the potential for bugs as they require reasoning about *localities*. With the RTSJ, programmers must be aware *where* an object has been allocated, a piece of information that cannot be obtained straightforwardly by inspection of the program text. The result is that scopes are, in our experience, the main source of program errors in RTSJ programs.

We propose a new programming mechanism called *Scoped Types*, designed to support safe scoped memory programming in concurrent real-time systems. Our goal is to devise a solution which does not require changes to the language or the tools that are used to write and run programs (e.g. compilers, development environment, and preprocessors) and ensure that programs will not experience run-time access violations. Yet Scoped Types should not impose undue restrictions on RTSJ programs. The underlying motivation for the proposed mechanisms is to keep the cost of adopting low. In fact, we have tried to retain the option of translating programs using

Scoped Types into RTSJ programs and running them on an unmodified real-time Java virtual machine.

To appeal to programmers, our proposal requires minimal syntactic overhead and only modest changes to RTSJ programs. With Scoped Types, the run-time hierarchy of scopes and subscopes is captured in the program text by the static hierarchy of Java packages. Thus, to comply with scoped types, RTSJ programs must be refactored so that objects that are meant to live in the same scope are declared in the same Java package. While this requires changes to grouping of classes and may entail increasing the visibility of some classes, these costs are mitigated by gains in clarity. The definition and behavior of scoped classes is constrained by six simple rules which taken together ensure that scope access errors will never occur, and object deallocation will never introduce dangling pointers. Furthermore, cycles in the scope hierarchy are impossible.

The contributions of this paper are the following:

1) **A new programming model** which provides *static correctness guarantees while remaining simple* and eschewing the need to modify Java in a significant way.
2) **Proof of soundness** by extracting the essence of scoped types in a core object calculus and proving soundness of the type system.

One of the difficulties in extending a language such as Java is that features interact. This can lead to undesirable side effects, even for seemingly simple changes, significantly complicating the lives of implementers and users. This motivates our choice of starting with a formalization of scoped types as a proof of soundness is an essential first step to any language extension of the kind we are considering here. We are currently implementing support for scoped types in the Ovm RTSJ virtual machine [5] and refactoring several large programs to abide with the scoped type discipline.

## II. An Example: A Real-time Collision Detector

To illustrate the usage of scopes, we present the example of an aircraft collision detection algorithm [8]. Collision detection is performed by a single real-time thread which receives a series of *frames* containing aircraft call signs along with positions and direction vectors. The output of the algorithm is a warning each time any pair of aircraft are on a collision course. We have implemented two versions of the algorithm — one in plain Java and one in RTSJ. The Java implementation is 2500 lines of code of which fewer than 200 lines had to be adapted to make the program RTSJ compliant.

Collision detection is performed iteratively. A `Frame` object containing a number of `Aircraft` objects is received from the sensors once per iteration of the main run loop. The contents of the frame are used to update a state table and then compute collision vectors. The RTSJ implementation of the algorithm uses four scopes, the distinguished `ImmortalMemory` and `HeapMemory`, along with two user-defined instances of `ScopedMemory`. Fig. 1 illustrates the memory structure of the program. While the main object, `App`, of the application is created in immortal memory, none of the other objects should be allocated there, as objects in immortal memory are never deleted. Thus, the first action of the `App.run()` method is to create and enter a new scope, `mem`. That scope will be used to contain the program's stable storage. A second scope, `cdmem`, is used for temporary storage, so that all the temporary objects are deleted at the end of each iteration.

Fig. 2 illustrates the main points of the algorithm. The `App.main()` method is responsible for starting a new real-time thread (`App` extends `NoHeapRealtimeThread`). As real-time threads may not execute within the heap, the first action that is performed by that method is to enter immortal memory. Note the use of reflection (`newInstance`) to create objects in different scopes. Then the `App.run()` method creates a new scope to hold the application's stable store (all state that must be preserved between iterations is stored in the instance of class `StateTable`) and starts an instance of `Runner` in the newly created scope.

The `Runner.run()` method is an example of the *scoped run loop* pattern [8]. The method starts by creating a scope, `cdmem`, to hold temporary objects. Then it repeatedly executes the `Detector.run()` method within `cdmem`. Since there is no other thread contending for that scope, after each iteration the scope is cleared. We remark that the `ScopedMemory` object itself remains intact between invocation, as does the `Detector` – both are allocated in the `mem` area which is not reclaimed for the lifetime of the application.

As can be seen from Fig. 2, although perhaps simple in theory, RTSJ Scoped Memory is complex in practice. For example, in each iteration of the run loop a new frame object is created along with an aircraft and a position. These objects are all allocated in the `cdmem` scope, whereas the state table is in the parent `mem` scope. In order to store a newly detected plane in the state table, the program has to reflectively create instances of `Aircraft` and `Position` in the correct (`mem`) scope. For another example, consider that

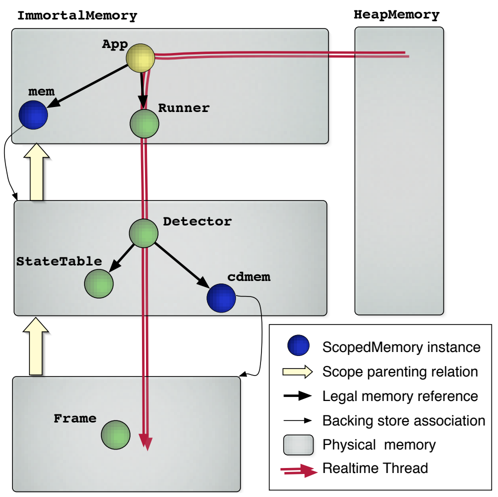


Fig. 1. Scopes in the example application. The main `App` object is allocated in immortal memory `imm`. Application stable state is held in the scoped area `mem`, per-iteration objects are allocated in `cdmem`.

```
class App extends NoHeapRealtimeThread {
   static void main() {
      imm = ImmortalMemory.instance();
      app = (App) imm.newInstance( App.class);
      app.start(); }
   void run() {
      LTMemory mem = new LTMemory( ...);
      mem.enter( new Runner() ); }
}
class Runner implements Runnable {
   void run() {
      LTMemory cdmem = new LTMemory( ...);
      Detector cd =
         new Detector( new StateTable() );
      while ( true )
         cdmem.enter( cd); }
}
```

```
class Detector implements Runnable {
   StateTable state;
   void run() {
      Frame frame = receiveFrame();
      Position pos_in_table =
            state.get( frame.getAircraft());
      if (pos_in_table == null) {
         mem = MemoryArea.getMemoryArea( this);
         Aircraft new_plane =
            mem.newInstance( Aircraft.class);
         frame.getAircraft().update( new_plane);
         pos_in_table =
            mem.newInstance( Position.class);
         state.put( new_plane, pos_in_table);
      }
      pos_in_table.update( frame.getPosition());
   }
}
```

Fig. 2. The main method of the application is used to bootstrap the real-time task. The run method of `App` is used to set up the application's stable store. The `Runner` class holds the application's main loop. All methods are public unless stated otherwise. Class `LTMemory` is a particular kind of scoped memory area which guarantees linear time allocation. Class `NoHeapRealtimeThread` is the parent class of all hard real-time thread classes, it is guaranteed not to interfere with the garbage collector.

the `Aircraft.update()` method (not shown here), takes an aircraft as argument and copies the information out of itself into its argument. We were forced to use this tortured design so that we can copy data allocated in the inner `cdmem` scope into an object in the out stable store `mem` scope.

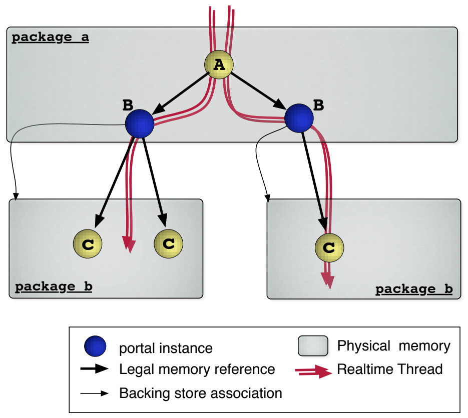
**Discussion.** This complex explanation shows that a large amount of information about this example is implicit in the RTSJ code. The memory scopes within which variables are allocated (and therefore to which they can refer) are not recorded in the text of the program; there is no information about the scope that a particular instance of the `Aircraft` class is stored in, for example. This means that a minor typographical error could go undetected by the compiler, and then cause a runtime failure during rare circumstances as the program is run — such as when the program actually detects a collision. Similarly, the nesting relationship between the `imm`, `mem`, and `cdmem` scopes is implicit in the code: if the `run()` method of the `Detector` class attempted to reenter the `mem` scope, the program would suffer a `ScopedCycleException`. Finally, the programmer intends all objects contained within the `cdmem` to be discarded after each iteration, but this is not in any way obvious from the code of the program.

## III. Real-time Programming with Scoped Types

To make real-time Java programming more reliable and predictable, we have developed a type system that can statically guarantee the absence of reference assignment errors, enforce the single parent rule for scopes, and ensure that there can be no references into the contents of a scope when it is scheduled to be discarded. Our proposal is referred to as *Scoped Types*. Scoped Types require the addition of two modifiers to the language, no compiler changes, and minimal support from the underlying virtual machine.

**Language extensions.** Our model distinguishes between two kinds of classes in a real-time Java program: *scoped classes* which are allocated within a particular memory scope, and *gate classes* which reify memory scopes. Most of the objects in the program are instances of scoped classes, and are allocated in memory scopes. Instances of gate classes turn scopes into first-class entities: threads enter memory scopes by invoking methods of gate objects, and exit scopes when these calls return. The key observation is that an object allocated in a scoped memory area can only be used in that scope and its nested subscopes. Thus, we statically restrict the accessibility of a scoped class to the classes whose instances are allocated in the same or nested scopes.

**Integration with Java.** To minimize the changes to the language, or at least to its syntax and to the tool processing the language, we take advantage of existing concepts, such as visibility rules and access modifiers, to integrate Scoped Types with Java. Scoped and gate classes are declared by appending the respective modifiers to class declarations (`@scoped` and `@gate`), no other annotations are needed. These annotations are consistent with the Metadata JSR, and will be recognized by Java 1.5 compilers. We call packages that contain scoped types *scoped packages*. Scoped packages are the unit of protection and of allocation. Each scoped package is the static representation of a family of memory scopes and defines the types of objects that may be allocated in these scopes. We use nested packages to represent potentially nested memory scopes: a memory scope created by some gate class in a scoped package can only contain nested subscopes defined by gate



```
package a;
    @scoped class A {
        ...
    }

package a.b;
    @gate class B {
        ...
    }
    @scoped class C {
        ...
    }
```

Fig. 3. A program written with Scoped Types. The program's static structure consists of two packages `a` and `a.b`. At runtime two instance of the gate class `B` are created, thus giving rise to two distinct scopes. Notice that gate objects, like `ScopedMemory` instances in the RTSJ, are allocated in the parent scope. Overall, the code is shorter than the RTSJ version and makes explicit the allocation context of objects.

classes in immediate subpackages. Instances of classes defined in top-level package are allocated in immortal memory.

**Dynamics.** While scoped packages describe the static structure of an application, restricting programs to a single instance of each scope (and thus exactly matching the static package hierarchy) prevents some useful programming idioms. Thus, at runtime, every *instance* of a gate class corresponds to a new memory scope. So an application that creates two gates for the same package, gets two distinct scopes which can be used independently. The type system guarantees that references across sibling scopes cannot arise, thus objects allocated within two instances of the same gate class can safely be reclaimed at different times. A scope's gate is the *only* object from the scoped package that is visible in the parent package. In fact, gates are allocated in their parent scope, just as RTSJ `ScopedMemory` objects are allocated in an enclosing area. The current allocation context is *always* defined by the package in which the current class was defined. Changing allocation context is thus as simple as calling a method of an object living in a different scope. Concurrency comes in quite naturally – multiple threads execute in the same scope if they invoke a method on the same gate. The implementation keeps track of the number of threads in a scope by a simple reference counting scheme. Just as in RTSJ, objects within a scope can be reclaimed when the last thread exits. Fig. 3 illustrates these concepts.

**Static guarantees.** Our model imposes some static constraints on the accessibility of classes. We require that scoped classes in a package be accessible only to the classes defined in that package and its subpackages, while gate classes are only accessible to classes defined in their immediate parent package. In other words, classes are *not* allowed to access classes in inner nested subpackages (other than the gates of their immediate subpackage). 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 `IllegalAssignmentErrors` can never happen. Threads can only enter the scopes defined in some package (by calling methods on gate classes in that package) from the code in the immediate super-package. This ensures that the hierarchy of memory scopes always follows the same hierarchy as the corresponding packages, enforcing the single parent rule and preventing `ScopedCycleExceptions`.

**Scoped Type Confinement Rules.** Scoped Types' static guarantees are enforced by the following syntactic rules that must hold for all scoped and gate classes. Rules $\mathcal{C}1$, $\mathcal{C}2$, and $\mathcal{C}3$ bind scoped classes, while Rules $\mathcal{S}1$ to $\mathcal{S}3$ bind gate classes. Besides the visibility constraints of Rules $\mathcal{C}1$ and $\mathcal{S}1$, we also require that ($\mathcal{C}2$) references of scoped type cannot be widened to types in other packages while ($\mathcal{S}2$) the references of gate types cannot be widened to other types. Note that a reference of type `C` can be widened to type `D` only if `C` is a subtype of `D`. Reference widening can happen through operations such as assignments, casts, and method invocations. The restrictions on reference widening help us to track references by their

| | |
|---|---|
| $\mathcal{C}1$ | A scoped type is visible only to classes in the same package or subpackages. |
| $\mathcal{C}2$ | A scoped type can only be widened to other scoped types in the same package. |
| $\mathcal{C}3$ | The methods invoked on a scoped type must be defined in the same package. |
| $\mathcal{S}1$ | A gate type is only visible to the classes in the immediate super-package. |
| $\mathcal{S}2$ | A gate type cannot be widened to other types. |
| $\mathcal{S}3$ | The methods invoked on a gate type must be defined in the same class. |

static types.

These rules are similar in spirit to the confinement rules presented in [9]. The type system presented in the next section formalizes these intuitive rules.

**Restrictions.** Scoped Types do restrict the set of valid Java programs. Even though they do not require changes to the syntax they do change the programming model. To start with, while an instance of a scoped class may extend an arbitrary class, none of the methods defined outside of the scoped package can be invoked. This really means that, for scoped classes, inheritance of code from classes defined outside of their package is disallowed. Functionality provided in a (non-scoped) parent class must be overridden in the scoped class. Moreover the restrictions on widening mean that the reuse of library classes, such as vectors, hash tables and the like, will be limited. We defend these choices by remarking that most existing Java libraries have not been designed for a real-time setting. In our experience most library classes rely on garbage collection to reclaim internally allocated objects and are thus not suited for use in with the RTSJ. Another potentially contentious issue is that, from a software engineering point of view, grouping classes with respect to their allocation context may lead to somewhat unnatural program structures. While this may be true, we believe that the benefits in clarity and correctness are sufficient to justify our approach. We discuss possible extensions to overcome some of these restrictions in Section VI.

### A. The Example Revisited

Scoped Types simplify programming within nested memory scopes. We can rewrite the collision detector example to use Scoped Types with very few changes. We first need to define three packages to model the three scopes of the original application. This is because in Scoped Types, the programmers have to choose the package within which each class should be statically *defined*, rather than deciding where instance should be dynamically *allocated*, as in RTSJ.

The scoped version of the program, shown in Fig. 4 and the code in Fig. 5, consists of three packages, `imm`, `imm.mem` and `imm.mem.cdmem` mirroring the dynamic scope hierarchy of the algorithm. The class `Main` is the only class that executes in immortal memory, and its only purpose is to create an instance of the `App` class, which is the gate of the `imm.mem` package. The `App` class, once started, will then allocate an instance of the `Detector` class, which is the gate for the `imm.mem.cdmem` scope. The run loop again boils down to calling the detector's `run` method. The program's stable state is held in the `imm.mem` package, and is composed of instances of the `StateTable`, `Aircraft`, and `Position` classes. Per-iteration temporary objects are stored in the `cdmem` package

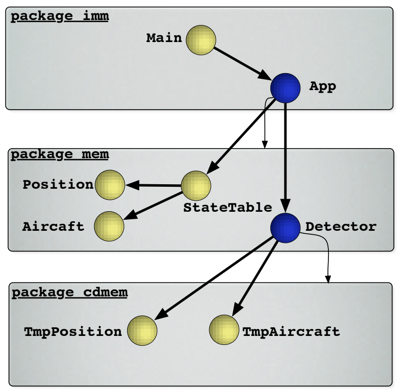


Fig. 4. The reference patterns of scoped and gate objects in the Scoped Type version of the example. The only references allowed to go from a parent package to a child are references originating from the gate. The gate object is a dominator for all scoped types in its package and subpackages. Note that although the figure does not show it, references from child packages to their parents are allowed.

```
package imm;

@scoped class Main {
   static void main() { new App().start();
   }
}

package imm.mem;

@gate final class App
      extends NoHeapRealtimeThread {
   void run() {
      cd = new Detector();
      state = new StateTable();
      key = new Aircraft();
      while ( true ) cd.run( state, key); }
}

@scoped class StateTable ...
@scoped class Aircraft ...
@scoped class Position ...

package imm.mem.cdmem;

@gate final class Detector {
   void run(StateTable state, Aircraft key){
      Frame frame = receiveFrame();
      TmpAircraft plane = frame.getAircraft();
      plane.update( key);
      Position pos_in_table = state.get( key);
      if ( pos_in_table == null )
         state.put(plane.copy(),
                  frame.getPosition().copy());
      else
         frame.getPosition()
            .update(pos_in_table);
   }
}

@scoped class TmpAircraft ...
@scoped class TmpPosition ...
@scoped class Frame ...
```

Fig. 5. The collision detector example with Scoped Types. The program is split into three packages representing the different scopes used in that program. All support classes (*e.g.* Aircraft) are defined in the appropriate scope.

and consist of `TmpAircraft`, `TmpPosition` and `Frame`.

Notice that with Scoped Types it is impossible to confuse planes in the inner `imm.mem.cdmem` scope with planes in the stable `imm.mem` scope, as they are represented by different types. A `copy` method is implemented in `TmpAircraft` to create a `Aircraft` object that *must*, by definition, be allocated in the parent scope. Similarly, since the state table is allocated in `imm.mem`, the types in `imm.mem.cdmem` are not accessible to it. Thus, we cannot use a `TmpAircraft` object as the `key` to find out whether a plane is already stored in the table, and we use an `Aircraft` object instead. The `update` method of a `TmpAircraft` object refreshes the `key` with the information about the current plane.

In this way, Scoped Types statically maintains the invariants that RTSJ checks dynamically. By statically associating scoped objects to their defining packages, Scoped Types can ensure that illegal incoming references are never created. Similarly, by modeling nested scopes with nested packages, Scoped Types ensure that scopes will never form cycles. Finally, by statically tracking the objects contained within each scope, Scoped Types ensure that it is safe to discard all the objects in a scope once the last thread has left it.

### B. *Refactoring an existing system*

We are in the process of refactoring a RTSJ program called Zen to use Scoped Types. Zen is a CORBA object request broker designed to support distributed, real-time and embedded (DRE) applications, written in the Real-time Specification for Java [10]. Zen has been designed for memory-constrained DRE applications. For our experiment we have selected a minimal configuration (about 20K LOC of Java code) that provides sufficient functionality for a number of benchmark applications that will be tested on a 300Mhz PPC board with 64MB of memory. We used the Eclipse development environment to assist in the refactoring exercise.

The process of refactoring Zen proved surprisingly simple. The first step is to instrument the program to print for each object instantiated its class and the scope in which it is created and also to print the parent relation between scopes. The second step involve creating Java packages that mirrored the scope hierarchy. Then, classes that are used unambiguously in only one scope can be moved to the respective package (with Eclipse this is painless). The remaining classes are used in several scopes, for these it may be necessary to either find a way to modify the application logic, or, in last resort, to duplicate code.

Figure 6 summarizes the difference in package structure between Zen and ScopedZen. The majority of classes in subpackages of `zen.orb` are moved up to the new package `orb`. This corresponds to the main memory area in which an ORB executes. The class `ORB` is the gate class for the `orb` package, each instance of this class represent one ORB running in the virtual machine. The `orb.transport` and `orb.transport.message` are used in the processing of requests. Once a request has been processed all objects allocated in those scopes are reclaimed. Finally, the other scopes

| ZEN | SCOPED ZEN |
|---|---|
| `zen.orb` | `orb` |
| `zen.orb.any` | `orb.acceptor` |
| `zen.orb.dynany` | `orb.poa` |
| `zen.orb.giop` | `orb.transport` |
| `zen.orb.policies` | `orb.transport.message` |
| `zen.orb.resolvers` | `orb.threadpool` |
| `zen.orb.transport` | `orb.waiter` |
| `zen.orb.transport.iiop` | |
| `zen.poa` | |
| `zen.poa.mechanism` | |
| `zen.poa.policy` | |

Fig. 6. Refactoring the Zen CORBA ORB.

(`orb.acceptor`, `orb.threadpool` and `orb.waiter`) are present to mirror the original design of Zen. They could be folded in `orb` as their lifetime is exactly that of `orb`.

## IV. THE SJ CALCULUS

To gain confidence in the programming model underlying our proposal, we introduce the SJ calculus, a sparse imperative and concurrent object calculus, modeled after Featherweight Java [11], in which scopes are first-class values. SJ formalizes the type confinement rules of Scoped Type in terms of a type system. Our proof of type soundness gives us the guarantee that confinement cannot be breached during execution of a well-typed program. We can then proceed to prove that the shape of the scope hierarchy is restricted to tree. And, finally, that deallocation of a scope will not result in dangling references. SJ is a simple object calculus, to keep the semantics concise we have omitted some features that are not essential to the main results. These features include static methods, synchronization, access modifiers, and down-cast expressions. Some specific features related to scoped memory such as the size and the type of the memory area (linear or variable allocation time) are also omitted.

### A. *Syntax and Types*

The syntax of the SJ calculus, Figure 7, draws on our previous work [9]. The formalism and syntax is based on the Featherweight Java (FJ) system which has been widely adopted as a vehicle of language research. SJ has two kinds of class declarations, *scoped classes* and *gate classes*, the former annotated with a `scoped` and the latter with a `gate`. Classes belong to packages, which can be nested in an arbitrary

$$
\begin{array}{lcl}
\mathtt{L} & ::= & \circ\ \mathtt{class}\ \mathtt{P.C}\ \triangleleft\ \mathtt{D}\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\mathtt{K}\ \overline{\mathtt{M}}\,\} \\
\mathtt{K} & ::= & \mathtt{C}()\ \{\mathtt{super}();\ \mathtt{this}.\overline{\mathtt{f}} := \overline{\mathtt{new}\ \mathtt{D}();}\,\} \\
\mathtt{M} & ::= & \mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\,\{\,\mathtt{return}\,\mathtt{e};\ \} \\
\mathtt{e} & ::= & \mathtt{x}\ \mid\ \mathtt{e.f}\ \mid\ \mathtt{e.m}(\overline{\mathtt{e}})\ \mid\ \mathtt{new}\,\mathtt{C}()\ \mid\ \mathtt{e.f} := \mathtt{e} \\
 & & \ \mid\ \mathtt{spawn}\,\mathtt{e}\ \mid\ \mathtt{reset}\ \mathtt{e}\ \mid\ \mathtt{v} \\
\circ & ::= & \mathtt{gate} \mid \mathtt{scoped} \qquad \mathtt{v} ::= \ell \qquad \mathtt{P} ::= \mathtt{p}\ \mid\ \mathtt{p.P}
\end{array}
$$

Fig. 7. Syntax of the SJ calculus.

**Allocation:**

$$\begin{array}{l}
\text{Suppose } \sigma(\ell) = \mathtt{C}_0^{\ell'}(\overline{\mathtt{v}}) \\
\mathit{allocScope}_\sigma(\mathtt{C}, \ell) = \ell \text{ if } \mathtt{C}_0 \text{ is a gate type} \\
\quad \text{and} \left\{ \begin{array}{l} \text{either } \mathtt{C} \text{ is a scoped type and} \\ \quad \mathtt{C}, \mathtt{C}_0 \text{ in the same package} \\ \text{or } \mathtt{C} \text{ is a gate type and} \\ \quad \mathtt{C} \text{ in the immediate subpackage of } \mathtt{C}_0 \end{array} \right. \\
\mathit{allocScope}_\sigma(\mathtt{C}, \ell) = \mathit{allocScope}_\sigma(\mathtt{C}, \ell') \quad \text{otherwise}
\end{array}$$

**Evaluation context:**

$$\begin{array}{lcl}
E[\circ] & ::= & \circ \ \mid \ E[\circ].\mathtt{m}(\overline{\mathtt{e}}) \ \mid \ \mathtt{v}.\mathtt{m}(\ldots, \mathtt{v}_{\mathtt{i}-1}, E[\circ], \mathtt{e}_{\mathtt{i}+1} \ldots) \\
& \mid & E[\circ].\mathtt{f}_\mathtt{i} \ \mid \ E[\circ].\mathtt{f}_\mathtt{i} := \mathtt{e} \ \mid \ \mathtt{v}.\mathtt{f}_\mathtt{i} := E[\circ] \ \mid \ \mathtt{reset}\, E[\circ]
\end{array}$$

**Scope reference counts:**

$$\mathit{refcount}(\ell, t[\,\overline{\ell\, \mathtt{e}}\,] \mid P') = \mathit{count}_\ell(\overline{\ell}) + \mathit{refcount}(\ell, P')$$

$$\mathit{refcount}(\ell, \emptyset) = 0 \qquad \mathit{count}_\ell(\emptyset) = 0$$

$$\mathit{count}_\ell(\overline{\ell}\,.\,\ell) = 1 + \mathit{count}_\ell(\overline{\ell}) \qquad \mathit{count}_\ell(\overline{\ell}\,.\,\ell') = \mathit{count}_\ell(\overline{\ell})$$

Fig. 8. Auxiliary definitions.

package hierarchy. Each package may contain a mixture of scoped and gate classes. We add an assignment expression and an expression for creating a new thread of control. Finally, we add a *reset* expression, which clears the fields of a gate object if the objects is not used by any threads. Reseting a gate corresponds to deallocating a scope in RTSJ, the operation was added to model GC, but is interesting in its own right as we observed in [8].

We take metavariables $\mathtt{C}, \mathtt{D}$ to range over classes, $\mathtt{M}$ to range over methods, $\mathtt{K}$ over constructors, and $\mathtt{f}$ and $\mathtt{x}$ to range over fields and variables (including parameters and the pseudo variable `this`), respectively. We also use $\mathtt{P}$ for package names, $\mathtt{e}$ for expressions and $\ell$ for memory references. We use over-bar to represent a finite ordered sequence, for instance, $\overline{\mathtt{f}}$ represents $\mathtt{f}_1\,\mathtt{f}_2 \ldots \mathtt{f}_\mathtt{n}$. The term $\overline{\mathtt{l}}\,.\,\overline{\mathtt{l}'}$ denotes sequence concatenation. The calculus has a call-by-value semantics. The expression $[^{\mathtt{v}}/_{\mathtt{v}_\mathtt{i}}]\overline{\mathtt{v}}$ yields a sequence identical to $\overline{\mathtt{v}}$ except in the $i$th field which is set to $\mathtt{v}$. We use the usual dot notation to represent nested packages. That is, the package `p.q` is a subpackage of `p`. The presentation of the calculus inherits some of the syntactic oddities of FJ, so $\overline{\mathtt{e}}\,\mathtt{e}$ is a short hand for $\mathtt{e}_1 \ldots \mathtt{e}_\mathtt{n}\,\mathtt{e}$, and $\mathtt{m}(\overline{\mathtt{C}}\,\overline{\mathtt{x}})$ stands for $\mathtt{m}(\mathtt{C}_1\,\mathtt{x}_1, \ldots, \mathtt{C}_\mathtt{n}\,\mathtt{x}_\mathtt{n})$.

## *B. Semantics*

In SJ, each gate object represents a distinct scoped memory area and whenever a gate is reset all of the objects that were allocated within the associated scope are reclaimed. While the package hierarchy imposes a static structure on scopes, gate objects allow multiple scope instances to be created at runtime. The main restriction imposed by SJ is that a gate can only allocate objects of scoped classes belonging to the same package and gates defined in immediate subpackages. When this restriction is combined with confinement invariants that prevents gate objects leaking from their parent package, we obtain the key property for scoped memory management, namely the restriction that threads enter scopes in the same order as the nesting relation of the packages containing the gate classes.

As in Featherweight Java, the semantics assumes the existence of a class table containing the definitions of all classes. We had to add a store $\sigma$ and a collection of threads $\overline{P}$ labeled by distinct identifiers $\overline{t}$. Objects are of the form $\mathtt{C}^\ell(\overline{\mathtt{v}})$, where $\mathtt{C}$ is a class, $\overline{\mathtt{v}}$ the value of the fields, and $\ell$ the gate of the scope in which it was allocated. The store $\sigma$ is a sequence, $\overline{\mathtt{C}^\ell(\overline{\mathtt{v}})}$, of objects, each denoted by a distinct label $\ell_\mathtt{i}$. Fig. 8 defines a number of auxiliaries relations. The partial function $\mathit{allocScope}_\sigma(\mathtt{C}, \ell)$ retrieves the allocation scope for an object of the type $\mathtt{C}$ when the current receiver object is $\ell$. Our type system ensures that all the scopes form a tree. Intuitively, $\mathit{allocScope}_\sigma(\mathtt{C}, \ell)$ searches the scope tree upward starting from $\ell$ or the scope of $\ell$ until it finds a scope $\ell'$ of the type $\mathtt{C}'$, which is in the same package as $\mathtt{C}$ if $\mathtt{C}$ is scoped and is in the immediate super-package of $\mathtt{C}$ if $\mathtt{C}$ is a gate. For some $\mathtt{C}$ and $\ell$, $\mathit{allocScope}_\sigma(\mathtt{C}, \ell)$ is not defined. An evaluation context, Fig. 8, is an expression $E[\circ]$ with a hole that can be filled in with another expression of proper type. An expression $\mathtt{e}_0$ is written as $E[\mathtt{e}]$ only if $\mathtt{e}$ is in the form of $\mathtt{v}$, $(\mathtt{C})\ \mathtt{v}$, $\mathtt{v}.\mathtt{f}$, $\mathtt{v}.\mathtt{f} := \mathtt{v}'$, $\mathtt{v}.\mathtt{m}(\overline{\mathtt{v}})$, $\mathtt{spawn}\,\mathtt{e}$, or $\mathtt{reset}\ \mathtt{e}$. For a non-value expression $\mathtt{e}_0$, there exists an unique evaluation context $E[\circ]$ such that $\mathtt{e}_0 = E[\mathtt{e}]$ and $\mathtt{e}$ is not a value. Evaluation contexts do not include the body of `spawn`.

The dynamics semantics of SJ is split in two: expression evaluation rules given in Fig. 9 and the computation rules in Fig. 10.

**Expression rules.** These evaluation rules consider only operations performed within a single thread. The evaluation relation has the form $\sigma, \ell\ \mathtt{e}\ \rightarrow\ \sigma', \ell'\ \mathtt{e}'$ where $\sigma$ is the initial store, $\ell$ is the reference to object currently executing, and $\mathtt{e}$ the expression to evaluate. The reduction rules field select (R-FIELD), field update (R-UPDATE), and method invocation (R-INVK) are not surprising, where $\mathit{mbody}(\mathtt{m}, \mathtt{C})$ returns parameters $\overline{\mathtt{x}}$ and method body $\mathtt{e}$ of $\mathtt{m}$ when it is invoked on an object of the type $\mathtt{C}$. The instantiation rule (R-NEW) must ensure that the class of the object about to be created, $\mathtt{C}$, can be instantiated in the current scope (as defined by $\mathit{allocScope}_\sigma(\mathtt{C}, \ell_0) = \ell'$). The fields of $\mathtt{C}$ must also be initialized, though not in the current scope but rather in the scope of the newly allocated object (this only matters if $\mathtt{C}$ is a gate). Finally the store is updated with a fresh reference $\ell$ bound to the newly allocated object. The helper function $\mathit{init}(\mathtt{C})$ returns the list of initial values of the fields in $\mathtt{C}$ (including the fields inherited from its super-classes). Rule (R-RESET) clears all fields of the target object. This ensures that all objects previously in the scope are now unreachable. In practice, programmers do not explicitly write such reset expressions, they are performed implicitly by the VM.

**Computation rules.** The computation rules are of the form $\sigma, P \Rightarrow \sigma', P'$ where $\sigma$ is a store and $P$ is a set of threads. Each thread $t[\,\overline{\ell\, \mathtt{e}}\,]$ in $P$ has a distinct label $t$ and a runtime call

$$\frac{\sigma(\ell) = \texttt{C}^{\ell'}(\overline{\texttt{v}}) \quad \mathit{fields}(\texttt{C}) = (\overline{\texttt{C}}\ \overline{\texttt{f}})}{\sigma, \ell_0\ \ell.\texttt{f}_\texttt{i} \ \longrightarrow\ \sigma, \ell_0\ \texttt{v}_\texttt{i}} \qquad \text{(R-FIELD)}$$

$$\frac{\begin{array}{c}\sigma(\ell) = \texttt{C}^{\ell'}(\overline{\texttt{v}}) \quad \mathit{fields}(\texttt{C}) = (\overline{\texttt{C}}\ \overline{\texttt{f}}) \\ \sigma' = \sigma[\ell \rightarrow \texttt{C}^{\ell'}([{}^{\texttt{v}}/_{\texttt{v}_\texttt{i}}\overline{\texttt{v}}])]\end{array}}{\sigma, \ell_0\ \ell.\texttt{f}_\texttt{i} := \texttt{v} \ \longrightarrow\ \sigma', \ell_0\ \texttt{v}} \qquad \text{(R-UPDATE)}$$

$$\frac{\begin{array}{c} \mathit{allocScope}_\sigma(\texttt{C}, \ell_0) = \ell' \quad\ \ell \text{ fresh} \\ \mathit{init}(\texttt{C}) = \overline{\texttt{new D()}} \quad \sigma'' = \sigma[\ell \rightarrow \texttt{C}^{\ell'}(\overline{\texttt{null}})] \\ \sigma'', \ell\ \texttt{new}\ \texttt{D}_1() \ \longrightarrow\ \sigma_1, \ell\ \texttt{v}_1 \\ \dots \\ \sigma_{\texttt{n}-1}, \ell\ \texttt{new}\ \texttt{D}_\texttt{n}() \ \longrightarrow\ \sigma_\texttt{n}, \ell\ \texttt{v}_\texttt{n} \\ \sigma' = \sigma_\texttt{n}[\ell \rightarrow \texttt{C}^{\ell'}(\overline{\texttt{v}})] \end{array}}{\sigma, \ell_0\ \texttt{new C()} \ \longrightarrow\ \sigma', \ell_0\ \ell} \qquad \text{(R-NEW)}$$

$$\frac{\begin{array}{c} \sigma(\ell) = \texttt{C}^{\ell'}(\overline{\texttt{v}}) \quad\ \mathit{init}(\texttt{C}) = \overline{\texttt{new D()}} \\ \sigma, \ell\ \texttt{new}\ \texttt{D}_1() \ \longrightarrow\ \sigma_1, \ell\ \texttt{v}'_1 \\ \dots \\ \sigma_{\texttt{n}-1}, \ell\ \texttt{new}\ \texttt{D}_\texttt{n}() \ \longrightarrow\ \sigma_\texttt{n}, \ell\ \texttt{v}'_\texttt{n} \\ \sigma' = \sigma_\texttt{n}[\ell \rightarrow \texttt{C}^{\ell'}(\overline{\texttt{v}}')] \end{array}}{\sigma, \ell_0\ \texttt{reset}\ \ell \ \longrightarrow\ \sigma', \ell_0\ \ell} \qquad \text{(R-RESET)}$$

$$\frac{\sigma(\ell) = \texttt{C}^{\ell'}(\overline{\texttt{v}}') \quad \mathit{mbody}(\texttt{m, C}) = (\overline{\texttt{x}},\ \texttt{e})}{\sigma, \ell_0\ \ell.\texttt{m}(\overline{\texttt{v}}) \ \longrightarrow\ \sigma, \ell\ [{}^{\overline{\texttt{v}}}/_{\overline{\texttt{x}}},\ {}^{\ell}/_{\texttt{this}}]\texttt{e}} \qquad \text{(R-INVK)}$$

Fig. 9. Expression evaluation.

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{e}]\,] \\ P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{e}']\,] \\ \texttt{e} \neq \texttt{reset}\ \ell',\ \ell'.\texttt{m}(\overline{\texttt{v}}) \qquad \sigma, \ell\ \texttt{e} \ \longrightarrow\ \sigma', \ell\ \texttt{e}' \end{array}}{\sigma, P \Rightarrow \sigma', P'} \qquad \text{(G-STEP)}$$

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{e}_0]\,] \\ P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{e}_0]\,.\,\ell'\,\texttt{e}'\,] \\ \texttt{e}_0 = \ell'.\texttt{m}(\overline{\texttt{v}}) \quad \sigma, \ell\ \texttt{e}_0 \ \longrightarrow\ \sigma, \ell'\ \texttt{e}' \end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad \text{(G-ENTER)}$$

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{e}]\,.\,\ell'\,\texttt{v}\,] \\ P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{v}]\,] \\ E[\texttt{e}] = \ell^{\texttt{th}}\ \vee\ \texttt{e} \text{ is not a value} \end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad \text{(G-RETURN)}$$

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,] \\ P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\ell^{\texttt{th}}]\,] \ \mid\ t'[\,\overline{\ell\,\ell^{\texttt{th}}}\,.\,\ell\,\texttt{e}'\,] \\ \texttt{e} = E[\texttt{spawn}\,\texttt{e}'] \qquad t' \text{ fresh} \end{array}}{\sigma, P \Rightarrow \sigma, P'} \qquad \text{(G-SPAWN)}$$

$$\frac{\begin{array}{c} P = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{reset}\ \ell']\,] \\ P' = P'' \mid t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\ell']\,] \\ \text{if } \mathit{refcount}(\ell', P) \neq 0 \text{ then } \sigma' = \sigma \\ \text{else } \sigma, \ell\ \texttt{reset}\ \ell' \ \longrightarrow\ \sigma', \ell\ \ell' \end{array}}{\sigma, P \Rightarrow \sigma', P'} \qquad \text{(G-RESET)}$$

Fig. 10. Computation rules.

stack which is a list of receiver-expression pairs $\ell, \texttt{e}$. Rule (G-STEP) is simple, it picks one thread for execution and evaluates the expression $E[\texttt{e}]$ on the top of the thread's stack. Note that this rule applies when $\texttt{e}$ is not a method invocation or reset expression. Rule (G-ENTER) evaluates a thread $t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,]$ containing a method call $\texttt{e} = E[\texttt{e}_0]$ and $\texttt{e}_0 = \ell'.\texttt{m}(\overline{\texttt{v}})$. It creates a new stack frame for the body of the method, $\texttt{e}'$, and the result is a frame $\overline{\ell\,\texttt{e}}\,.\,\ell\, E[\texttt{e}_0]\,.\,\ell'\,\texttt{e}'$. If the expression on the top of a thread's stack is reduced to a value $\texttt{v}$, then by Rule (G-RETURN), the thread can pop the stack frame and continue execution with $\texttt{v}$ as the resulted value of a method call. Note that if the expression replaced by $\texttt{v}$ is not a value, then its evaluation context is unique. Thus, the replacement is unambiguous. Rule (G-SPAWN) evaluates a thread $t[\,\overline{\ell\,\texttt{e}}\,.\,\ell\,\texttt{e}\,]$ containing a spawn expression $\texttt{e} = E[\texttt{spawn}\,\texttt{e}_0]$. The value of the spawn expression in $\texttt{e}$ is the distinguished $\ell^{\texttt{th}}$ which is a unique, global reference to an object of class $\texttt{Thread}$ and we assume that $\ell^{\texttt{th}}$ is allocated in immortal memory. A new thread $t'$ is created to evaluate $\ell\,\texttt{e}_0$. The new thread is started with a call stack $\overline{\ell\,\ell^{\texttt{th}}}$ that matches the call stack of the original thread $t$ to ensure that scope reference counts are accurate. Rule (G-RESET) clears the fields of a gate $\ell'$ when no thread is using that gate (*i.e.* when $\mathit{refcount}(\ell', P) = 0$). For simplicity, the fields of a gate are reset to default value explicitly by a *reset* expression of the form $\texttt{reset}\ \ell'$ and if the reference count of $\ell'$ is not zero, then the fields of $\ell'$ are not cleared (this makes *reset* nonblocking to avoid deadlock).

## C. Type Rules

The typing rules are shown in Figure 11 and 12. Some auxiliary functions used in typing rules are defined as follows: $\mathit{fields}(\texttt{C})$ returns the list of field declarations in the class $\texttt{C}$ (including the inherited fields) in the form of $\overline{\texttt{C f}}$; $\mathit{mdef}(\texttt{m}, \texttt{C})$ returns the defining class of the method $\texttt{m}$ by searching the class hierarchy upward from $\texttt{C}$; $\mathit{mtype}(\texttt{m}, \texttt{C})$ returns the type signature $\overline{\texttt{C}} \rightarrow \texttt{C}'$ of the method $\texttt{m}$ called on the type $\texttt{C}$, where $\overline{\texttt{C}}$, $\texttt{C}'$ are the parameter and return types. The type judgments are of the form $\Gamma, \Sigma \vdash \texttt{e} : \texttt{C}$, where $\Gamma$ is the type environment of variables and $\Sigma$ is the type environment of object labels.

The subtyping relation $<:$ is a reflexive and transitive closure of the relation that $\texttt{C} <: \texttt{C}'$ if the class $\texttt{C}$ extends the class $\texttt{C}'$. We define the partial order $\preceq$ on types to limit the variables that can refer to scoped objects and gates; $\texttt{C} \preceq \texttt{C}'$ is defined if $\texttt{C} <: \texttt{C}'$, and $\texttt{C}, \texttt{C}'$ are both scoped types in the same package or $\texttt{C}$ is a gate type and $\texttt{C} = \texttt{C}'$. If $\texttt{C} \preceq \texttt{C}'$, then we say that $\texttt{C}$ is a *scope-safe* subtype of $\texttt{C}'$ and the widening of a reference from the type $\texttt{C}$ to $\texttt{C}'$ is *scope-safe*.

By Rules (T-UPDATE) and (T-INVK), the reference widening in the field assignments and parameters passing is *scope-safe*. Rule (T-STORE) of the form $\Sigma \vdash \sigma$ says that object store $\sigma$ is well typed, if the type environment $\Sigma$ has the same domain as $\sigma$ and for each object label $\ell$ in the domain of $\sigma$, $\Sigma(\ell)$ is equal to the type of $\sigma(\ell)$ and $\sigma(\ell)$ must also be well-typed. If

$$\Gamma, \Sigma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \qquad \text{(T-VAR)}$$

$$\Gamma, \Sigma \vdash \ell : \Sigma(\ell) \qquad \text{(T-LOC)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C} \quad \mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}})}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} : \mathtt{C}_\mathtt{i}} \qquad \text{(T-FIELD)}$$

$$\frac{\begin{array}{c}\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{mdef}(\mathtt{m},\ \mathtt{C}_0) = \mathtt{C}_0' \\ \mathit{mtype}(\mathtt{m},\ \mathtt{C}_0') = \overline{\mathtt{C}} \to \mathtt{C} \\ \Gamma, \Sigma \vdash \overline{\mathtt{e}} : \overline{\mathtt{D}} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}} \quad \mathtt{C}_0 \preceq \mathtt{C}_0'\end{array}}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{m}(\overline{\mathtt{e}}) : \mathtt{C}} \qquad \text{(T-INVK)}$$

$$\Gamma, \Sigma \vdash \mathtt{new}\ \mathtt{C}() : \mathtt{C} \qquad \text{(T-NEW)}$$

$$\frac{\begin{array}{c}\Gamma, \Sigma \vdash \mathtt{e}_0 : \mathtt{C}_0 \quad \mathit{fields}(\mathtt{C}_0) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \\ \Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C} \quad \mathtt{C} \preceq \mathtt{C}_\mathtt{i}\end{array}}{\Gamma, \Sigma \vdash \mathtt{e}_0.\mathtt{f}_\mathtt{i} = \mathtt{e} : \mathtt{C}_\mathtt{i}} \qquad \text{(T-UPDATE)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{Thread}}{\Gamma, \Sigma \vdash \mathtt{spawn}\,\mathtt{e} : \mathtt{Thread}} \qquad \text{(T-SPAWN)}$$

$$\frac{\Gamma, \Sigma \vdash \mathtt{e} : \mathtt{C} \quad \mathtt{C}\text{ is a gate}}{\Gamma, \Sigma \vdash \mathtt{reset}\ \mathtt{e} : \mathtt{C}} \qquad \text{(T-RESET)}$$

Fig. 11. Expression typing.

**Store Typing:**

$$\frac{\begin{array}{c}\mathrm{dom}(\Sigma) = \mathrm{dom}(\sigma) \quad \forall \ell \in \mathrm{dom}(\sigma)\ . \\ \Sigma \vdash \sigma(\ell)\ \wedge\ \Sigma(\ell) = \mathtt{C}\text{ if }\sigma(\ell) = \mathtt{C}^{\ell_0}(\overline{\mathtt{v}})\end{array}}{\Sigma \vdash \sigma} \qquad \text{(T-STORE)}$$

$$\frac{\mathit{fields}(\mathtt{C}) = (\overline{\mathtt{C}}\ \overline{\mathtt{f}}) \quad \emptyset, \Sigma \vdash \overline{\mathtt{v}} : \overline{\mathtt{D}} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}}}{\Sigma \vdash \mathtt{C}^\ell\ (\overline{\mathtt{v}})} \qquad \text{(T-STORELOC)}$$

**Method typing:**

$$\frac{\begin{array}{c}\Gamma = \overline{\mathtt{x}} : \overline{\mathtt{C}}, \mathtt{this} : \mathtt{C}_0 \quad \Gamma, \emptyset \vdash \mathtt{e} : \mathtt{C}' \quad \mathtt{C}' \preceq \mathtt{C} \\ \mathit{override}(\mathtt{m}, \mathtt{D}, \overline{\mathtt{C}} \to \mathtt{C}) \quad \Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C})\end{array}}{\mathtt{C}\ \mathtt{m}(\overline{\mathtt{C}}\ \overline{\mathtt{x}})\,\{\,\mathtt{return}\,\mathtt{e};\ \}\ \mathtt{OK\ IN}\ \mathtt{C}_0 \lhd \mathtt{D}} \qquad \text{(T-METHOD)}$$

**Class typing:**

$$\frac{\begin{array}{c}\mathtt{K} = \mathtt{C}()\ \{\mathtt{super}();\ \mathtt{this}.\overline{\mathtt{f}} := \overline{\mathtt{new}\ \mathtt{D}()};\} \\ \overline{\mathtt{M}}\ \mathtt{OK\ IN}\ \mathtt{C} \lhd \mathtt{D} \quad \overline{\mathtt{D}} \preceq \overline{\mathtt{C}} \quad \mathit{visible}(\overline{\mathtt{CD}}\mathtt{D}, \mathtt{C})\end{array}}{\circ\ \mathtt{class}\ \mathtt{P.C}\ \lhd\ \mathtt{D}\ \{\,\overline{\mathtt{C}}\ \overline{\mathtt{f}};\ \mathtt{K}\ \overline{\mathtt{M}}\,\}\ \mathtt{OK}} \qquad \text{(T-CLASS)}$$

Fig. 12. Type rules of store, method, and class.

$\sigma(\ell) = \mathtt{C}^{\ell'}(\overline{\mathtt{v}})$, then by Rule (T-STORELOC), an object $\mathtt{C}^{\ell'}(\overline{\mathtt{v}})$ is well-typed, if the types of $\overline{\mathtt{v}}$ are *scope-safe* subtypes of the field types.

In the typing rule for class (T-CLASS), we require that in a class C, and the types of the fields and the base class must be visible in C. Also, all methods in a class must be well-typed by Rule (T-METHOD). If a method in the class $\mathtt{C}_0$ is well-typed, then the method body e is well-typed by the expression typing rules, the type of the method body is a *scope-safe* subtype of the return type, and in addition, the method body must be visible in $\mathtt{C}_0$ as defined by the judgment $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0)$. The predicate $\mathit{override}(\mathtt{m}, \mathtt{C}_0, \overline{\mathtt{C}} \to \mathtt{C})$ in Rule (T-METHOD) is true if either the method m is not accessible in $\mathtt{C}_0$ or the type signature returned by $\mathit{mdef}(\mathtt{m}, \mathtt{C}_0)$ is the same as $\overline{\mathtt{C}} \to \mathtt{C}$. Note that in (T-CLASS) we abuse notation by writing $\mathit{visible}(\overline{\mathtt{C}}, \mathtt{C})$ to assert that all types in the $\overline{\mathtt{C}}$ are visible in C.

**Visibility of types and expressions.** The static constraints in our model are mostly to restrict widening of references, and also to limit the accessibility of expressions by their types. For example, an expression of scoped type C is only visible in the defining package of C and its subpackages. We define a relation on types – $\mathit{visible}(\mathtt{C}, \mathtt{C}_0)$ (type C is *visible from* type $\mathtt{C}_0$), which encodes the SJ access control rules: a scoped type defined in package P is visible to the class $\mathtt{C}_0$ defined in P and its subpackages; a gate class C is only visible from the class $\mathtt{C}_0$ defined in the immediate parent package. One slightly surprising implication of this definition is that a gate type is not visible in its own class definition. Thus a gate class C does not contain code that refers to itself with the exception, as we shall see later, of the pseudo variable `this` which may indeed be used to access fields and methods from within the gate class.

We check the method body to determine whether type visibility constraints are violated in a class. In Rule (T-METHOD), the judgment $\Gamma \vdash \mathit{visible}(\mathtt{e}, \mathtt{C}_0)$ holds if e of type C is visible in a class $\mathtt{C}_0$, which means that either $\mathtt{e} = \mathtt{this}$ or the type C is visible in the class $\mathtt{C}_0$ (i.e. $\mathit{visible}(\mathtt{C}, \mathtt{C}_0)$) and all the subexpressions of e are visible in $\mathtt{C}_0$. We make an exception for `this` because even though a gate type is visible only to the classes of its immediate super-package, a gate object must be able to use the variable `this` for accessing its fields and calling its methods. For any scoped class, the type of the variable `this` are always visible in its class.

## D. Properties.

The purpose of our model is to simplify the allocation of objects in scoped memory areas. Thus, we would like to statically guarantee the properties that during the evaluation of a real-time program,

1) the nesting structure of scopes remain a tree,
2) deallocated objects in scopes are no longer accessible.

In RTSJ, the nesting structure of scopes is determined by how threads enter scopes. In our model, the scope structure is fixed by how the gate objects representing the scopes are created. That is, if a scope $a$ is represented by a gate object created in the scope $b$, then $a$ must be directly contained in $b$; moreover, the gate object representing $a$ is defined in the immediate subpackage of the gate object representing $b$. Thus, our type system guarantees that the scopes represented by the gate objects always form a tree. It also ensures that the threads in a program will preserve such a scope tree such that each thread either enters the scopes already entered by the thread or enters a new scope directly contained in the current scope of the thread. Thus, even though a scope stack of a thread may grow indefinitely (e.g. the thread reenters the scopes already on stack), the nesting structure of scopes resembles the nesting structure of the scoped packages and always remains a tree.

To ensure that deallocated objects are no longer accessible, we require that a scoped object can only access objects with the same or longer lifetime, while a gate object can in addition access the objects allocated in the scope that it represents. Thus, when the last thread in a scope exits, the objects in that scope can be deallocated. The deallocated objects are no longer accessible in the program because they are only accessible to the scoped objects with the same or shorter lifetime and to the gate of the scope, but those scoped objects are already deallocated and no method of the gate object is being invoked. The above accessibility constraints are enforced by SJ's type rules, where a scoped type is accessible in the classes of its defining package and the subpackages. It is possible for two instances of the same class to be allocated in two sibling scopes (they share some parent scope). To prevent such objects from accessing each other, we limit the access to a gate object to itself and the classes in its immediate super-package. Consequently, an object may only gain access to the gate of its own scope and the gates of its immediate nested scopes and thus, it cannot reference objects in its sibling scope.

Our proof strategy for the above properties is to show that the safety invariant that we define below is preserved in each reduction step. We say that an object $o$ can *safely access* $o'$ if either $o'$ has longer lifetime than $o$ or $o$ is the gate of the scope where $o'$ is allocated. A program $\sigma, P$ is *safe* if for each label $\ell$ defined in $\sigma$, the object $\sigma(\ell)$ can *safely access* the objects referenced in its fields, and for each frame $\ell\ \mathtt{e}$ in the call stack of each thread in $P$, the object $\sigma(\ell)$ can *safely access* every object referenced in $\mathtt{e}$.

A program $\sigma, P$ is well-typed if it is *safe* and $\exists\Sigma$ such that $\Sigma \vdash \sigma$ and the call stack of each thread in $P$ is well-typed.

Given $\Sigma$, the call stack $\overline{\ell\,\mathtt{e}}\,.\,\ell\ E[\mathtt{e}]\,.\,\ell'\ \mathtt{e}'$ is well-typed, if $\exists\mathtt{C}'$ such that $\emptyset;\Sigma \vdash \mathtt{e}' : \mathtt{C}'$, and $\emptyset;\Sigma \vdash \mathtt{e} : \mathtt{C}$ implies $\mathtt{C}' \preceq \mathtt{C}$, $\mathit{visible}(\mathtt{e}, \ell)$ is true (defined below), and $\overline{\ell\,\mathtt{e}}\,.\,\ell\ E[\mathtt{e}]$ is well-typed. Given $\Sigma$, the call stack $\ell\ \mathtt{e}$ is well-typed if $\exists\mathtt{C}$ such that $\emptyset;\Sigma \vdash \mathtt{e} : \mathtt{C}$ and $\mathit{visible}(\mathtt{e}, \ell)$ is true.

Given $\Sigma$, the constraint $\mathit{visible}(\mathtt{e}, \ell)$ is true if either $\mathtt{e} = \ell$ or, $\Sigma(\ell) = \mathtt{C}_0$ and $\emptyset;\Sigma \vdash \mathtt{e} : \mathtt{C}$ imply $\mathit{visible}(\mathtt{C}, \mathtt{C}_0)$ and for each subexpression $\mathtt{e}'$ of $\mathtt{e}$, $\mathit{visible}(\mathtt{e}, \ell)$ is true.

In Theorem 3, we prove that if a program is well-typed, then it will not get stuck. We model deallocation using the explicit *reset* expression, which clears the fields of a gate if the gate is not used by any threads. We prove in Theorem 4 that the objects allocated in the gate before the reset are no longer accessible afterward.

**Lemma 1** *If $\sigma, P$ is well-typed and $\sigma, P \Rightarrow \sigma', P'$, then $\sigma', P'$ is well-typed.*

We say that a thread of the form $t[\,\ell_0\ \mathtt{v}\,]$ in $P$ is terminated.

**Lemma 2** *If $\sigma, P$ is well-typed, then either all threads in $P$ are terminated or there exists $\sigma', P'$ such that $\sigma, P \Rightarrow \sigma', P'$.*

We say that an irreducible program $\sigma, P$ is stuck if $P$ contains a non-terminated thread. As usual, $\Rightarrow^*$ is the transitive and reflexive closure of $\Rightarrow$.

**Theorem 3** *If $\sigma, P$ is well-typed and $\sigma, P \Rightarrow^* \sigma', P'$, then $\sigma', P'$ is not stuck and it is well-typed.*

Theorem 4 shows that in a well-typed program $\sigma, P$, if a gate $\sigma(\ell_0)$ is reset successfully, then the objects allocated in the scope represented by $\sigma(\ell_0)$ are not reachable in $\sigma, P$. Recall that if a scope represented by $\sigma(\ell_0)$ is not used by any thread, then $\mathit{refcount}(\sigma, \ell_0) = 0$. We say that $\ell$ is reachable in $\sigma, P$ if either it is referenced in a thread of $P$ or it is in the field of $\sigma(\ell')$, where $\ell'$ is reachable in $\sigma, P$.

**Theorem 4** *If $P = P'' \mid t[\,\overline{\ell\,\mathtt{e}}\,.\,\ell\,E[\mathtt{reset}\ \ell_0]\,]$, $\sigma, P$ is well-typed, $\mathit{refcount}(\sigma, \ell_0) = 0$, and $\sigma, P \Rightarrow \sigma', P'$, then the objects of $\sigma$ that are allocated in the scope represented by the gate $\sigma(\ell_0)$ are not reachable in $\sigma', P'$.*

## V. Related Work

The dangers involved in the RTSJ programming model have motivated Kwon *et.al.* to propose a restricted programming model called Ravenscar-Java [12]. In Ravenscar memory areas cannot be nested and are single threaded. Scoped Types are intended to relax some of the restrictions of Ravenscar while remaining easy to understand and to verify. Boyapati [13] combine region-based memory management with ownership types to statically guarantee that real-time threads do not interfere with GC. While more flexible than Scoped Types, this approach is more invasive, requiring more program annotations, and more complex overall. Cyclone [14] is a type-safe language derived from C and it supports region-based memory management. Cyclone includes dynamic regions with lexically scoped lifetimes, stack and a heap region. To prevent dereferencing dangling pointers, Cyclone uses types parameterized by region names to track pointers to regions and uses effect annotations to prevent unsafe access to regions. The regions in Cyclone are limited to single threaded execution model. Also, the use of effects may not work with real-time Java, since the Java's type safety requirement does not allow objects to hold invalid references even if never used. Grossman extended Cyclone with a type system for preventing data races [15]. The MLKit is an implementation of ML which uses regions and region-inference [16], [17]. One of the main difference with the model presented here is that ML is a functional language without built-in support for concurrency.

There are two other open source virtual machines that implement parts of the RTSJ: Flex [18] and JRate [19], [5], as well as a number of commercial products and alternative proposals [20], [21], [22], [23], [24].

## VI. Conclusion

In this paper we have introduced Scoped Types, a static programming discipline to support the kind of scoped memory management found in the Real-Time Specification for Java. The key contribution of our proposal is that it statically maintains the invariants that the RTSJ checks dynamically, yet imposes minimal syntactic overheads upon programmers. In particular, by statically associating scoped objects to their defining packages, Scoped Types ensure that illegal incoming references are never created, eliminating the potential for run-time errors caused by illegal assignment operations. By modeling nested scopes with nested packages, Scoped Types ensure that scopes will never form cycles, again eliminating the potential for runtime exceptions. By statically tracking the objects contained within each scope, Scoped Types ensure that it is safe to discard all the objects in a scope once the last thread has left it. We have formalized Scoped Types within the SJ-calculus and demonstrated that it avoids dangling references and cycles of scopes. We hope the techniques embodied within Scoped Types may be useful in many RTSJ applications, making real-time Java programming more practical, convenient and reliable.

For future work, we plan to relax the type constraints in SJ so that more programs can be written. For example, the current type rules prevent a scoped class from invoking methods inherited from parent packages. Such invocation can result in the implicit reference widening of the variable `this` in the method call and consequently, objects in the outer scope may hold references to the receiver object. To prevent such problems, we may require such methods to be *anonymous* [9] so that the variable `this` can only be used for field access and calls to other anonymous methods. Another limitation of Scoped Types is the lack of reuse of library classes. This problem may be addressed by introducing generics to Scoped Types similar to the Confined Generic Types in [9]. For example, a generic class `Vector<X>` may be used in a scoped package as containers for instances of the scoped class `C` if we instantiate the class as `Vector<C>`. Instances of the class `Vector<C>` can be allocated in the scope of the `C` objects and be confined in that scope and its nested scopes. Of course, we have to decide where to allocate the objects created in the vector class. For example, if the vector class is implemented using linked list, then we may allocate the linked list objects in the current scope as well but we must take care to protect the references to those objects from illegal access. An alternative is to allocate those objects in heap memory but this would make such vector class less usable in `NoHeapRealtimeThread`.

## Acknowledgments

This work is partially supported by DARPA/ITO under the PCES program and by NSF award HDCCSR-0341304. The authors are grateful to Alex Potanin for refactoring Zen, and to David Holmes, Filip Pizlo and Dave Clarke for their contributions to earlier versions of this work.

## References

[1] G. Bollella, J. Gosling, B. Brosgol, P. Dibble, S. Furr, and M. Turnbull, *The Real-Time Specification for Java*. Addison-Wesley, June 2000.

[2] D. 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.

[3] NASA/JPL and Sun, "Golden gate," 2003, `http://research.sun.com/projects/goldengate`.

[4] Timesys Inc., "JTime," 2003, `http://www.timesys.com`.

[5] $S^3$ Lab, "The ovm customizable virtual machine project," Purdue University, Tech. Rep., 2004.

[6] D. G. Bobrow and B. Wegbreit, "A model and stack implementation of multiple environments," *Communications of the ACM*, vol. 16, pp. 591–602, 1973.

[7] K. H. Randall, "Cilk: efficient multithreaded computing," Ph.D. dissertation, Massachusetts Institute of Technology, Dept. of Electrical Engineering and Computer Science, 1998.

[8] F. Pizlo, J. Fox, D. Holmes, and J. 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.

[9] T. Zhao, J. Palsberg, and J. Vitek, "Lightweight confinement for Featherweight Java," in *Proceedings of the ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'03)*. ACM Press, Oct. 2003, pp. 135–148.

[10] A. Krishna, D. Schmidt, and R. Klefstad, "Enhancing Real-Time CORBA via Real-Time Java Features," in *24th International Conference on Distributed Computing Systems (ICDCS 2004)*, Hachioji, Tokyo, Japan, 2004, pp. 66–73.

[11] A. Igarashi, B. C. Pierce, and P. Wadler, "Featherweight Java: a minimal core calculus for Java and GJ," *ACM Transactions on Programming Languages and Systems*, vol. 23, no. 3, pp. 396–450, May 2001.

[12] J. Kwon, A. Wellings, and S. King, "Ravenscar-Java: A high integrity profile for real-time Java," in *Joint ACM Java Grande/ISCOPE Conference*, November 2002.

[13] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. 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.

[14] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney, "Region-based memory management in Cyclone," in *Proceedings of the ACM Conference on Programming Language Design and Implementation*, June 2002.

[15] D. Grossman, "Type-safe multithreading in Cyclone," in *ACM Workshop on Types in Language Design and Implementation*, January 2003.

[16] M. Tofte and L. Birkedal, "A region inference algorithm," *ACM Transactions on Programming Languages and Systems*, vol. 20, no. 4, pp. 724–767, July 1998.

[17] M. Tofte and J.-P. Talpin, "Region-based memory management," *Information and Computation*, Feb. 1997.

[18] W. S. Beebee, Jr. and M. Rinard, "An implementation of scoped memory for real-time Java," in *Proceedings of the First International Workshop on Embedded Software (EMSOFT)*, 2001.

[19] A. Corsaro and D. Schmidt, "The design and performace of the jRate Real-Time Java implementation," in *The 4th International Symposium on Distributed Objects and Applications (DOA'02)*, 2002.

[20] K. Nilsen, "Adding real-time capabilities to Java," *Communications of the ACM*, vol. 41, no. 6, pp. 49–56, June 1998. [Online]. Available: http://www.acm.org:80/pubs/citations/journals/cacm/1998-41-6/p49-nilsen/

[21] D. Buytaert, F. Arickx, and J. Vos, "A profiler and compiler for the Wonka Virtual Machine," in *USENIX JVM'02 Work in Progress*, San Francisco, CA, August 2002.

[22] J. Tryggvesson, T. Mattsson, and H. Heeb, "Jbed: Java for real-time systems," *Dr. Dobb's Journal of Software Tools*, vol. 24, no. 11, Nov. 1999.

[23] U. Gleim, "JaRTS: A portable implementation of real-time core extensions for Java," in *Proceedings of the Java Virtual Machine Research and Technology Symposium (JVM '02)*. Berkeley, CA, USA: USENIX, 2002.

[24] F. Siebert, "Hard real-time garbage collection in the Jamaica Virtual Machine," in *Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, 1999.