Unification of Safety-Critical Java

Kelvin Nilsen

Unification of Safety-Critical Java

Kelvin Nilsen, Chief Technology Officer Java
Atego Systems, Inc.

Introduction

In response to increasing interest in the use of object-oriented technology for development of safety-critical systems, the new DO-178C guidelines will include supplements to address object-oriented technology, model-driven development, formal methods, and development tool qualification [1]. These supplements correlate well with the emerging safety-critical Java standard. As a portable object-oriented programming language enabling high levels of abstraction, safety-critical Java is an ideal candidate for automatic code generation for programming models. The use of formal methods to prove the absence of certain memory management errors at run time is a critical distinction between safety-critical Java and the Real-Time Specification for Java (RTSJ) [2]. And the specialized development tools that facilitate the use of these formal methods will, in the ideal, be qualified so that the results of their analysis can be relied upon as trustworthy safety certification evidence.

The use of Java [3] in real-time systems has taken various forms over the years. The earliest commercial approaches to real-time Java (circa 1997) used standard edition Java APIs with real-time garbage collection [4]. This preserves the high-level abstractions of Java that make it inherently safer than, for example, C and C++. Later, in 2000, the RTSJ introduced new mechanisms to support higher degrees of determinism in real-time Java [2]. Unfortunately, the RTSJ represented a step backwards in terms of software safety, abstraction, and maintainability. While the RTSJ mechanisms make it possible to write Java programs that equal the real-time latency guarantees of assembly language and C, RTSJ programmers must exercise an abundance of caution in order to avoid illegal memory access errors, memory fragmentation errors, out-of-memory conditions, and priority inversion problems. It has been argued that programming with the RTSJ has few, if any, benefits over real-time programming with more traditional legacy languages like C and Ada [5]. As a result, even ten years after publication of the official RTSJ "standard", there are no published reports of successfully deployed RTSJ systems.

Work on a standard safety-critical Java specification for Java has been ongoing since the Open Group hosted an initial meeting on this topic in July 2003. Early discussions within the Open Group resulted in an architectural framework and a set of annotations to enable modular composition and certification of independently developed safety-critical Java components [6]. These approaches were first implemented in the commercial PERC Pico product by Atego Systems in 2007 [7].

In 2006, the Open Group's safety-critical Java effort transformed itself into the JSR-302 expert group of the Java Community Process. With this transition, new parties joined the process, and these parties brought a shift away from some of the Open Group's earlier directions. In particular, there was a desire to maintain greater compatibility with the RTSJ. The resulting JSR-302 specification, which is now nearly complete, represents a compromise between many alternative perspectives and objectives [8].

This paper discusses four distinct *semantic models* for safety-critical code written in the Java language. The first three of these correspond to a very restrictive runtime environment that manages temporary memory using scope-based allocation instead of tracing garbage collection.

Baseline JSR-302 represents the lowest level of abstraction. In this semantic model, scopes are much simpler than in vanilla RTSJ in that scopes are maintained in a LIFO structure that eliminates memory fragmentation and restricts sharing between threads. Another simplification of safety-critical Java is that the objects that are reclaimed when a scope is exited do not undergo finalization. As with the RTSJ, safety-critical Java programmers are still responsible for explicitly managing scopes and assignments that violate RTSJ scope restrictions throw an *IllegalAssignmentError* exception.

Annotated JSR-302 provides improved abstraction in that a static checker enforces that the application code will not throw an *IllegalAssignmentError*. The checker relies on the presence of optional annotations in the source code. Even with Annotated JSR-302, programmers are required to explicitly manage the creation, sizing, entry into, and exit from scopes.

As with Annotated JSR-302, the *PERC Pico* semantic model relies on the presence of annotations to clarify the programmer's intentions with regards to scoped memory relationships. Unlike JSR-302, the use of annotations in this semantic model is not optional. Annotation enforcement with PERC Pico is more than simply checking that the programmer is doing the right thing. With PERC Pico, annotation checking interacts with native code generation to make the annotations valid.

For example, if the annotations on a body of code indicate that a particular temporary object may be referenced from external scopes, the compiler automatically arranges that the memory allocated for that object is taken from a scope that lives as long as those external scopes.

The fourth and most abstract semantic model discussed in this paper is *Traditional Java*. While this model may not be suitable for the highest levels of safety critical rigor, it is already being deployed in projects that have criticality in the ranges of DO-178B levels C and D. The programming model is much simpler due to reliance on tracing garbage collection; thus it is actually easier to prove certain safety properties of software written according to this semantic model. On the other hand, the use of real-time garbage collection adds considerable complexity to the run-time environment. Proving that the more complex run-time environment is implemented correctly, that the application releases garbage at a rate that is consistent with its appetite for new memory allocations, and that the garbage collector is getting sufficient CPU time to replenish the memory allocation pool before it is depleted all add significantly to the costs of safety certification.

Besides examining key differences between the various semantic models, this paper proposes a unifying approach to combine their respective strengths. Most of the technical approaches discussed in this paper have not yet been fully implemented. Consistent with common technical writing practice, the paper speaks of these proposed future technologies using the present tense.

1. JSR-302 Overview

A JSR-302 application is structured as one or more missions, running either in sequence or concurrently. Each mission has a mission memory to hold all of the temporary objects that are to be shared between the independently executing threads that comprise the mission. JSR-302 threads are embodied as periodic event handlers, asynchronous event handlers, or ongoing threads. Each thread has a stack of private memory areas to hold the temporary objects required for its computations.

One of the design ideals of the JSR-302 specification was to maintain compatibility with RTSJ. This is most noteable in its approach to allocation of memory for short-lived objects. Like the RTSJ, the JSR-302 specification requires that programmers explicitly manage memory scopes. Each memory scope has a size that must be determined by the application programmer. A JSR-302 program enters a scope by executing one of several possible standard APIs. Once entered, subsequent new object allocations are, by default, satisfied by taking the memory from the currently entered scope. After all threads exit the scope, all of the objects allocated within the scope are discarded and their memory is reclaimed.

Unlike the RTSJ, JSR-302 supports only two very restrictive scope types. Both scope types are instantiated only by infrastructure, only in particular contexts at particular times. A *MissionMemory* scope is instantiated and sized only during the initialization of a new mission. A *PrivateMemory* scope is only instantiated when an event handler begins to run, or when user code requests to enter a new private memory area. These restrictions on the generality of RTSJ scopes simplify the run-time execution model and make it possible for implementations of the JSR-302 specification to guarantee that memory fragmentation does not prevent the timely and reliable creation of new memory areas, unlike the RTSJ.

As with the RTSJ, JSR-302 programmers are responsible for honoring the restriction that objects residing within a particular memory scope are never allowed to refer to objects residing within more inner-nested memory scopes. At each point in a program's execution that this rule would be violated, an *IllegalAssignmentError* exception is thrown instead.

Clearly, it is undesirable that a safety-critical program might terminate with a run-time error due to an inappropriate pointer assignment. In its current form, the JSR-302 specification states that it is ultimately the developers' responsibility to assure the absence of illegal assignments in their safety-critical Java programs. Vendors of safety-critical Java programming tools and run-time environments are encouraged to provide tools to help programmers prove the absence of these exceptions. And the draft specification describes an optional annotation system and enforcement tool that can be applied to safety-critical applications at the discretion of developers or project leads to guarantee the absence of these exceptions. This annotation system is simpler and less expressive than the system that had been designed during the earlier Open Group standardization efforts.

Figure 1 provides an example program written according to the rules of JSR-302, using the optional annotations to enforce scope assignment safety. This program fragment represents code that might be executed within a mission's constructor to initialize a cryptography key.

The excerpted code omits certain details. For example, it does not show the annotation that requires instances of *TheMission* class to reside within the *"TM"* scope. The constructor for *TheMission* is shown at the bottom of the figure.

The sample code illustrates the complexity of managing explicit scopes as is required to perform certain intermediate computations in temporary memory that can be reclaimed upon termination of the constructor. This sample code creates and enters a private memory area to hold an *AbsoluteTime* object, a *Random* object, three *BigInteger* objects, and an *AssignCryptoKey* object. Note that the calculation of the private memory's scope size must account for internal objects which are allocated within

```
@Scope("TM") @SCJAllowed(members=true)
static class CalculateCryptoKey implements Runnable {
  @DefineScope(name="TM.0", parent="TM")
  @Scope("TM.0") @SCJAllowed(members=true)
  static class AssignCryptoKey implements Runnable {
    TheMission tm;     // resides in scope "TM"
    BigInteger bi;          // resides in scope "TM.0"
    AssignCryptoKey(TrainMission tm, BigInteger bi) {
      this.tm = tm;
      this.bi = bi;
    }
    @RunsIn("TM")
    public void run() {
      // copy bi into the "TM" scope (from the "TM.0" scope)
      tm.crypto_key = bi.multiply(BigInteger.ONE);
    }
  }

  TrainMission tm;
  public CalculateCryptoKey(TheMission the_mission) {
    tm = the_mission;
  }

  @RunsIn("TM.0")
  public void run() {
    AbsoluteTime now =
        javax.realtime.Clock.getRealtimeClock().getTime();
    Random r = new Random(now.getMilliseconds());
    BigInteger t1, t2, t3;
    t1 = new BigInteger(128, 24, r);
    t2 = new BigInteger(128, 24, r);
    t3 = t1.multiply(t2);
    AssignCryptoKey assigner =
        new AssignCryptoKey(tm, t3);
    MemoryArea.getMemoryArea(tm).
        executeInArea(assigner);
  }
}

@SCJRestricted(INITIALIZATION)
public TheMission() {
  CalculateCryptoKey calculator =
      new CalculateCryptoKey(this);
  SizeEstimator z = new SizeEstimator();
  z.reserve(AbsoluteTime.class, 1);
  z.reserve(Random.class, 1);
  z.reserve(BigInteger.class, 3);
  z.reserveArray(20, byte.class);
  z.reserveArray(20, byte.class);
  z.reserveArray(40, byte.class);
  z.reserve(CalculateCryptoKey.AssignCryptoKey.class, 1);
  ((ManagedMemory) MemoryArea.getMemoryArea(this)).
  enterPrivateMemory(z.getEstimate(), calculator);
}
```

Figure 1. JSR-302 Constructor for TheMission

the constructors of *AbsoluteTime*, *Random*, or *BigInteger* instances. This information may not be known without scrutiny of the respective constructor implementations. Reliance on this information violates best practice guidelines for information hiding and encapsulation.

In this sample code, the temporary *CalculateCryptoKey* and *SizeEstimator* objects that are allocated by the *TheMission* constructor are placed in *MissionMemory* and live there until the mission itself terminates. It would have been preferable to place these objects also in temporary memory, but it is very difficult to do so using the JSR-302 programming conventions which were inherited from the RTSJ.

For comparison, a C implementation of a comparable memory model is represented by the code shown in Figure 2. This sample code assumes that libraries exist to perform similar actions in C that are performed by the existing Java libraries. The point of presenting this C version of the code is to clarify how a typical C programmer would arrange code so that the relevant objects are allocated in the appropriate stack memory locations. Note that the C version of *BigInteger* makes a reference to a separately allocated array of 8-bit characters to represent its integer encoding. The *initializeMission()* function links each *BigInteger* structure to the corresponding array.

```
typedef struct { char *digits;
  unsigned char avail_digits;
  unsigned char used_digits;
  unsigned char sign; } BigInteger;

typedef struct { BigInteger crypto_key; } TheMission;

TheMission tm;
char digits[40];

void initializeMission() {
  BigInteger t1, t2;
  char digits1[20], digits2[20];
  struct timespec now;
  longlong seed;
  clock_gettime(CLOCK_REALTIME, &now);
  seed = now.tv_nsec +
     (longlong) now.tv_sec * 1000000000;
  tm.crypto_key.digits = digits;   tm.avail_digits = 40;
  t1.digits = digits1; t1.avail_digits = 20;
  t2.digits = digits2; t2.avail_digits = 20;
  fillRandomBigInteger(&t1, 128, 24, &seed);
  fillRandomBigInteger(&t2, 128, 24, &seed);
  multiplyBigInteger(&t1, &t2, &(TheMission.crypto_key));
}
```

Figure 2. C initializer for TheMission

2. PERC Pico Overview

The PERC Pico technology is based on the original system of annotations that had been developed in the Open Group's original safety-critical Java meetings. This

annotation system provides increased expressive power and enables automatic determination of scope sizes. A sample constructor implementation written in the style of PERC Pico is shown in Figure 3.

```
@StaticAnalyzable
public TheMission() {
  @CaptiveScoped AbsoluteTime now;
  @CaptiveScoped r = new Random();
  @CaptiveScoped BigInteger t1, t2;
  now =  javax.realtime.Clock.getRealtimeClock().getTime();
  Random r = new Random(now.getMilliseconds());
  assert StaticLimit.InvocationMode("Digits=20");
  t1 = new BigInteger(128, 24, r);
  assert StaticLimit.InvocationMode("Digits=20");
  t2 = new BigInteger(128, 24, r);
  assert StaticLimit.InvocationMode("Digits=40");
  this.crypto_key = t1.multiply(t2);
}
```

Figure 3. PERC Pico Constructor for TheMission

Note that the code is much more concise and more readable than code written in the style of JSR-302. The PERC Pico compiler uses a sophisticated byte-code analyzer to trace the flow of each allocated object. Based on how the variable is used, it automatically determines the memory scope from which each allocation should be satisfied. For example, allocations assigned to a *captive-scoped* variable are always taken from an implicit local scope associated with execution of the current method. Within a constructor, variables assigned to *scoped* variables that are not *captive* are always taken from a special region of memory known as the *constructed scope*. The constructed scope is simply an expansion of the scope that holds the object being constructed. The reason the non-captive scoped objects that are allocated within a constructor are not simply placed in the same scope as the constructed object *this* is because the client code that invokes the constructor generally does not know about the internal behavior of a constructor, so it normally cannot calculate how large the corresponding scope would need to be.

In this sample code, the *BigInteger.multiply()* method is known to be declared with a *@CallerAllocatedResult* annotation. This indicates that the method places its result into a memory buffer that is supplied by the caller. The caller decides where the result is to be placed, which may not be in the caller's private scope. In this case, the result of the *t1.multiply()* invocation is placed in the constructed scope, so that it can be directly referenced by this object's *crypto_key* field.

Note that the constructor for *TheMission* is declared *@StaticAnalyzable*. This means that the compiler is expected to automatically analyze the size of the respective scopes. There are two scopes that are relevant to this particular example: the private temporary memory scope of the constructor must hold one *AbsoluteTime* object, one *Random* object, and two *BigInteger* objects; the constructed scope of the constructor must hold the *BigInteger* object returned from *t1.multiply()*.

The *BigInteger* data type represents arbitrary precision so its memory requirements depend on the value to be represented. The documentation for the *BigInteger* constructor makes clear that the memory requirements can only be computed (for any given context) if the programmer supplies an assertion to bound the value to be stored within the constructed *BigInteger* object. This example shows that each of the invocations responsible for constructing or allocating a *BigInteger* object is preceded by an assertion that limits the number of digits required to represent the object in a decimal representation of the integer.

3. Comparisons Between PERC Pico and JSR-302 annotations

There are fundamental differences between the approaches of PERC Pico and JSR-302. PERC Pico enables a more concise and more abstract programming style. Early experimentation with PERC Pico in several different domains has confirmed that the abstractions it supports make it easier to develop and maintain complex and evolving software systems [9]. In the paragraphs that follow, we draw several comparisons between the Annotated JSR-302 and PERC Pico semantic models.

Guiding Design Principles. Annotated JSR-302 is implemented as a specialization of the RTSJ. The simplicity of the annotation system was motivated in part by a desire to formally prove relevant attributes of the RTSJ subset's semantic model. PERC Pico is structured instead as a specialization of standard edition Java. The higher levels of abstraction provided by this semantic model were motivated by a desire to ease the software development and maintenance burden with less emphasis on developing a formal proof of the semantic properties.

In theory, it is easier to qualify the tool chain of a JSR-302 implementation but easier to develop and maintain a PERC Pico application.

Scope Relationships. One of the strengths of PERC Pico is that the PERC Pico annotations assert relative rather than absolute scope nesting relationships. When describing the semantic constraints on a particular API, the PERC Pico programmer uses annotations to state that certain arguments must reside in scopes that enclose, equal, or are enclosed by the scopes holding certain other arguments. This means that the same API can be used in many different contexts, as long as the actual arguments satisfy the relative scope nesting constraints. In contrast, JSR-302 annotations specify that particular classes always reside in particular named scopes. If the same class is required to appear in multi-

ple distinct scopes, programmers may be required to replicate the class (possibly using inheritance), and annotate each replication of the class with the distinct scope name in which it is intended to appear.

Scope Management. With Annotated JSR-302, classes that are not annotated are not allowed to escape the scope in which they are allocated. Classes that are annotated may be seen in other scopes, because the class annotation specifies the name of the scope in which all instances of that class reside. The JSR-302 annotation checker enforces that annotated classes are only instantiated when the current scope matches the named scope. The run-time environment's notion of current scope changes under explicit program control, with invocations of the *ManagedMemory.enterPrivateMemory()* or *MemoryArea.executeInArea()* methods. With Annotated JSR-302, it is the responsibility of application code to calculate the size of each scope.

With PERC Pico, instances of any class may reside in any scope. PERC Pico annotations are associated with reference variables rather than the class declarations. Certain variables are known to refer to the current method's private scope. Other variables are known to refer to scopes that enclose (are either identical to, or externally nested around) certain other scopes. Scopes are entered and exited implicitly, as control flows into and out of methods. Scopes are sized automatically as guided by static analysis of the memory allocation needs associated with each scope.

Abstraction and Information Hiding. The static analysis techniques implemented by both Annotated JSR-302 and Perc Pico are sound in that both systems assure the absence of dangling pointers at compile time. This represents a significant improvement over RTSJ, which relies on run-time checks and exceptions to prevent the introduction of dangling pointers.

With Annotated JSR-302, it is generally not possible to accurately calculate the required sizes of each scope without having a full awareness of the memory allocation behavior that occurs inside of abstract data types. These implementation details are ideally hidden from users of the abstract data type because any dependency on this information makes code brittle. If subsequent software maintenance activities make changes to the internal memory allocation behaviors, it becomes necessary to find and update all of the code throughout the application that makes use of this information.

With PERC Pico, software developers have the option of structuring code so that a static analysis tool automatically computes the required sizes of each scope. If evolution of an application's source code results in changes to its internal memory allocation behavior, the static analysis tool automatically reflects these changes in all relevant scope sizes when the code is recompiled. Further, PERC Pico introduces the notion of a constructed scope. A *constructed scope* represents a conceptual expansion of the scope that holds a newly allocated *this*. This scope serves to hold the objects that are allocated within a constructor or within *reentrant scope* methods (such as *Vector.add()*) which must reside at the same (or enclosing) scope level as the constructed object because they must be referenced from the constructed object.

Reuse of Software Components and Certification Artifacts. It is difficult to reuse software and certification evidence with Annotated JSR-302 because classes must be annotated differently for each context (scope) in which they appear. This results in code replication (possibly using inheritance) rather than code sharing. Each replica of the code must be independently certified in its given context. Programming errors in the sizing of scopes for particular contexts may cause the software to fail in those contexts.

In contrast, the PERC Pico annotations allow the same code to be used in many different contexts with its memory usage automatically tailored for the context in which it appears.

Modular Composition of Software and Certification Evidence. Both PERC Pico and annotated JSR-302 represent significant improvements over vanilla RTSJ in that annotations on component interfaces clarify the scope requirements associated with incoming arguments and instance fields. This allows the static checker to assure that composition of software components does not introduce scope assignment errors.

Among the certification artifacts that would be reused in an ideal scenario are audited logs of peer review activities, traceability of requirements to source to test plan, requirements-based test plan, and analysis of test results and code coverage.

In the case that certification artifacts have been gathered for independent software components, it is easier to leverage these preexisting certification artifacts with PERC Pico than with Annotated JSR-302. This is because a PERC Pico software component can be integrated in many different contexts without any changes to the PERC Pico source or executable code. Further, the PERC Pico annotations assure more preconditions than the JSR-302 annotations, including availability of memory within relevant scopes and availability of stack memory to create additional scopes as might be required to reliably execute the code of a particular software component. In contrast, Annotated JSR-302 software components can rarely be used as is in new contexts. Usually, the code must be copied and then annotated differently so that allocated objects are allowed to reside in particular scopes. The safety certification analysis must then consider the behavior of the code within the new environment.

Another important consideration is certification of the standard libraries, which are intended to be reused in many distinct contexts. With Annotated JSR-302, standard libaries are not annotated, and the formal methods that are enforced by the JSR-302 static analysis tools are not in force. It is the vendor's responsibility to correctly implement the standard libraries, and to prove correctness using *ad hoc* techniques. With Perc Pico, the library implementations are subjected to the same static analysis techniques as application code. This is made possible because the Perc Pico annotation system is sufficiently expressive to describe the scoping constraints on the library APIs and their implementations.

Design Patterns. The JSR-302 annotation system is designed primarily to support the scoped memory design pattern in which all objects of a linked data structure reside in the same scope. It handles this particular design pattern fairly well, without requiring annotations on class declarations. Whenever a given execution context refers to multiple distinct scopes (such as when computations use temporary objects to produce a result object result and the result object must be stored into a more permanent outer-nested structure), the JSR-302 annotation system becomes cumbersome. This is when it becomes necessary to name scopes and bind particular class instances to specific named scopes. Sometimes, it is possible to create multiple versions of a library class using inheritance, with each version bound to a distinct named scope. However, early experimentation with the JSR-302 annotation system suggests it is not always possible to achieve the desired flexibility by subclassing predefined standard libraries, and even when it is possible, the resulting code can be difficult to understand because of the requirement to divide logical control flows between multiple *Runnable* objects.

In contrast, the PERC Pico annotation system was designed with specific design patterns in mind, including differentation between scoped and captive-scoped variables, constructed scopes, caller-allocated-result methods, reentrant-scope classes, and same-scope linked data structures. The annotations have evolved to improve the ease of expression for common design patterns during nearly five years of experimentation with a variety of applications, including implementation of scoped-memory collection libraries, a scoped-memory dynamic class loader, and multiple real-time applications.

4. Mixed-Mission Deployment

As the JSR-302 draft nears final approval, Atego is undertaking to make PERC Pico compliant with the JSR-302 specification. This compatibility upgrade consists of the following critical accommodations:

1. The set of standard libraries supported by PERC Pico will be unified with the set of libraries offered by JSR-302. In general, JSR-302 defines a larger set of libraries than was anticipated in the earlier drafts of the safety-critical Java standard so this activity consists primarily of adding libraries.
2. Modify PERC Pico so that all threads are organized into missions.
3. Implement the JSR-302 standard so that PERC Pico offers developers the option of running applications comprised of either JSR-302 code or traditional PERC Pico code.
4. Enhance the PERC Pico verifier and run-time environment to enable the mixing of JSR-302 missions and PERC Pico missions. A mission sequencer would have the option of running a JSR-302 mission followed by a PERC Pico mission followed by another JSR-302 mission. A PERC Pico mission would be allowed to nest within a JSR-302 mission. And a JSR-302 mission would be allowed to nest within a PERC Pico mission.

For objects residing in immortal memory, both PERC Pico and JSR-302 enforce the same rules: Immortal objects can only refer to other objects residing in immortal memory. For both systems, no annotations are associated with the objects residing in immortal memory. Since the initial *Safelet* and its mission sequencer are both allocated in immortal memory, it is straightforward to interleave PERC Pico and JSR-302 missions under the direction of a *Safelet*'s outermost mission sequencer.

The rules enforced by the respective annotation systems differ, however, for objects that might reside in an outer-nested scope. This is relevant whenever a mission of one type is executed within the context of a mission of the other. To understand the significance of these differences, an overview of the salient features of the two annotation systems is provided below.

Rules for Annotated JSR-302 scopes. Based on annotations placed in source code, the JSR-302 checker enforces the following invariant properties:

1. For classes that are annotated to reside only in particular scopes, assure that the class is instantiated only within the specific named scope.
2. In the case that a superclass has different annotations than its subclass, certain upcasts from the subclass to the superclass are forbidden.
3. Instance field annotations may indicate that the field always refers to the same scope as the object itself.
4. Instance field annotations may indicate that the field always refers to immortal memory, or to a specific named scope.
5. Certain instance field annotations may indicate that the field refers to an object in an unknown scope.

6. Certain method invocations require that certain reference arguments refer to the same scope as the object that is the target of the invocation.
7. Certain method invocations require that particular reference arguments refer to specific named scopes, or to immortal memory.
8. Certain method invocations allow particular reference arguments to refer to objects residing in unknown scopes.

Rules for PERC Pico scopes. In contrast, the invariants enforced by the PERC Pico verifier are the following:

1. Certain instance fields are known to refer to immortal memory.
2. Certain instance fields are known to refer to objects residing the same scope as *this*.
3. The remaining instance fields are known to refer to objects residing in an enclosing scope. Immortal memory encloses all other scopes. Every scope is considered to enclose itself.
4. Certain method invocations require that certain reference arguments refer to immortal memory.
5. Certain method invocations require that certain reference arguments refer to objects residing in the same scope as the object that is the target of the invocation.
6. Certain method invocations require that particular reference arguments refer to scopes that enclose the scope of the object that is the target of the invocation.
7. Certain method invocations allow particular reference arguments to refer to objects residing in unknown scopes.

While the JSR-302 checker has the ability to statically enforce the invariant properties that are relevant to the JSR-302 scope safety model, it does not gather the information required to enforce, for example, that particular unknown scopes are known to enclose (be more outer-nested than) or be the same as certain other unknown scopes. Likewise, the PERC Pico verifier does not keep track of the information required to enforce the JSR-302 model. For example, there is no notion of named scopes within PERC Pico, and there is no way for a PERC Pico programmer to describe a requirement that instances of a particular class may only be instantiated within particular scopes. Thus, there is no way to translate PERC Pico annotations into equivalent JSR-302 annotations, or to translate JSR-302 annotations into equivalent PERC Pico annotations.

Differences in allowed behaviors. The execution models of the PERC Pico and JSR-302 environments also differ significantly. The most noteworthy distinction is the respective treatments of scopes.

In PERC Pico, a scope is a hidden implementation artifact, similar to an activation frame in C or Ada. There is no PERC Pico API to allow application software to refer to a particular scope, or to ask how large it is, or to ask how much memory remains available within it. There is also no API to ask which scope holds a particular object, or to request that a particular new object allocation be taken from a particular scope. This is intentional. If application software were allowed to directly manipulate scopes, then static analysis of scope safety would be much more difficult.

In contrast, JSR-302, because it is structured as a literal subset of RTSJ, treats scopes as first-class objects. While this generality provides a certain expressive power to software developers, it also introduces difficult challenges for static analysis. For example, with JSR-302, any software component that has access to an object residing in a particular scope is allowed to create new objects in that same scope. The protocol is straightforward:

1. Invoke *area = MemoryArea.getMemoryArea(object)*, and
2. Invoke *area.executeInArea(logic)* with a *Runnable logic* argument that performs memory allocations.

Note that in the presence of such allocations, it is extremely difficult for a static analysis tool to automatically calculate the required size of each scope.

Another distinction between PERC Pico and JSR-302 execution models is that PERC Pico allows classes to be dynamically loaded into mission memory, whereas JSR-302 requires all classes to be loaded into immortal memory prior to initialization of the application. The static variables of a PERC Pico dynamically loaded class may refer to objects residing in outer-nested mission scopes, whereas JSR-302 assumes that static variables refer only to immortal memory. In a mixed-model application, all of the JSR-302 code is statically loaded and reflection libraries are not supported, so the JSR-302 code cannot make any references to the static variables of dynamically loaded PERC Pico classes.

Sharing mission memory with inner-nested missions. Potential problems arise when JSR-302 missions nest inside of PERC Pico mission or when PERC Pico missions nest within JSR-302 missions. In both cases, the inner-nested mission may be able to see objects residing in an outer-nested mission's memory. When accessing those outer-nested objects (and/or manipulating the outer-nested mission scopes), the inner-nested mission needs to honor the constraints of the model that governs legal behavior within the outer-nested mission.

When a mission implemented according to one semantic model spawns an inner-nested mission implemented

according to the rules of a different semantic model, a proxy object isolates concerns between the two.

The complete application is divided into three distinct partitions. The PERC Pico missions are linked against standard libraries implemented with PERC Pico. Likewise, the annotated and unannotated JSR-302 missions are each compiled and linked with compatible libraries. Objects are never shared directly between the distinct execution models. Cross-model invocations are implemented as remote procedure calls.

Mission proxies implement the boundaries between missions compiled according to the conventions of the distinct semantic models. To make a mission eligible for execution as a sub-mission of a mission that is compiled according to a different semantic model, the application developer submits the mission to a proxy generation tool, which produces a wrapper to interface between the inner mission and its surrounding context.

The proxy generation tool requires that all arguments to the mission's constructors are either primitive types or are references to a declared interface. Furthermore, all method declarations associated with the interfaces that characterize the constructor's argument types must themselves be described as either primitive types or interface (of interfaces and primitives) types.

The output from the proxy generator is an implementation of a wrapper class that represents the original mission within the surrounding context. This wrapper class extends the *Mission* type so that an instance of this class can be returned from the enclosing mission sequencer's *getNextMission()* method. The wrapper class implements the conventions of the enclosing mission's semantic model. The mission that it wraps normally implements different conventions. In terms of JSR-302 annotation checking, the automatically generated wrapper mission is treated as infrastructure so that it is allowed to invoke the *cleanUp()* and *initialize()* methods of the wrapped mission. The mission proxy is assumed to reside within the inner-nested mission's mission memory. The constructor for the mission proxy instantiates the enclosed mission within this same mission memory.

The automatically generated mission wrapper isolates scope assignment issues between the enclosing and enclosed missions by implementing the following semantics:

1. For each interface type referenced either directly or indirectly in the declarations of the mission's constructors, including all super-interfaces, the proxy generator creates a concrete class to implement the interface. The inheritance relationship of automatically generated concrete classes mimics the inheritance hiearchy of the interfaces they implement.
   a. All automatically generated proxy classes have hidden constructors. This prevents user code from instantiating proxy objects.
   b. The root of the inheritance hierarchy is one of six specific base classes representing each of the possible combinations between enclosing and enclosed mission semantic models: *PP2SCJstatic_Proxy*, *PP2SCJdynamic_Proxy*, *SCJstatic2PP_Proxy*, *SCJstatic2SCJdynamic_Proxy*, *SCJdynamic2PP_Proxy*, or *SCJdynamic2SCJstatic_Proxy*.
   c. The interface declarations that characterize interactions between an inner-nested mission and an outer-nested mission may carry both JSR-302 and PERC Pico annotations. The proxy generation tool confirms that both sets of annotations are compatible. Compatibility means the annotations as interpreted within the inner-nested mission do not contradict the annotations as interpreted by the outer-nested mission.

   The proxy generator cannot enforce equivalence of annotations because each annotation system can describe scope relationships that cannot be described in the other. When necessary, a run-time check is performed within the proxy's method wrappers. If the required scope relationships are not satisfied by the arguments of an attempted cross-mission method invocation, an *IllegalArgumentException* is thrown. The same exception is thrown if a returned result of a cross-mission method invocation does not satisfy the caller's expected scope-relevant post-conditions.
2. For each reference argument passed to any constructor of the mission wrapper class, the mission wrapper's constructor creates (or finds a preexisting association with) a proxy object to represent the argument within the enclosed mission. The proxy object has the following characteristics:
   a. It implements the interface and provides no other public methods or fields.
   b. It implements additional services for use by infrastructure. These services are hidden from the application code. As an example, each proxy object maintains a reference to the original outer-mission object that it represents within the inner-nested mission.
3. In the case that a method invoked on an object residing in an outer-nested mission returns a reference to an object (which would necessarily reside in an outer-nested mission), the proxy's method wrapper replaces the returned value with a proxy object (either newly created or previously associated) to represent the returned result within the inner-nested mission. Proxy objects reside in mission memory. System integrators are responsible for sizing the

mission memory large enough to hold all of the proxy objects that it may need to represent.

4. Whenever the inner-nested mission invokes a method associated with a proxy representing an object residing in an outer-nested mission, the proxy's implementation of that method forwards the request on to the appropriate object residing within the outer-nested mission's scope after confirming that the scope relationships required by the original API's annotations are satisfied by the passed arguments and replacing all argument proxy references with references to the corresponding actual objects residing in the outer-nested mission's memory.

These protocols assure that an inner-nested mission never holds a direct reference to any object residing in the outer-nested missions. Instead, the inner-nested mission holds references to proxy objects with which it can invoke the services provided by the objects shared by the outer-nested mission.

Note that since the types of mission constructor arguments are restricted to be either primitive scalars or references to interfaces (of interfaces), the inner mission cannot make direct reference to any of the instance fields of the object.

1. Java interfaces do not have instance fields. A Java interface may have *static final* fields, but these can only refer to objects residing in immortal memory so accessing these objects does not introduce scope assignment problems that are any different than the scope assignment problems associated with access to any static field, which are already addressed in all three semantic models.
2. Since a proxy represents the outer-nested object within the inner-nested mission, coercions performed on the proxy object are unable to access the original object's type. This prevents access to methods or fields that are not part of the declared interface.

Note also that these protocols do not allow the inner-nested mission to pass as arguments references to objects residing within the inner-nested mission. The inner-nested mission could invoke a service of the outer-nested mission, passing as an argument a locally allocated object that implements the declared interface. However, the proxy that forwards the invocation to the outer-nested mission performs checks to confirm that every reference argument is actually a proxy and it replaces every proxy argument with a reference to the corresponding proxied object. Since proxy constructors are hidden from application code, there is no way for application code to spoof the run-time checking performed by the proxy's method forwarding mechanism. Any attempt to pass a local object as an argument to an invocation of a service performed in an outer-nested mission results in an *IllegalArgumentException*.

5. Traditional Java Integration

While most safety practitioners are quick to reject the use of tracing garbage collection in safety-critical systems, many software engineers are quick to respond that many of the alternatives to tracing garbage collection have comparable risks and very high software development and maintenance costs. The key relevant considerations are that (1) it is much easier to correctly implement complex software systems with languages that support tracing garbage collection; (2) integration of independently developed software components which share access to certain objects is much easier with languages that support tracing garbage collection; and (3) tracing garbage collection adds complexity to the binary implementation of application code, the schedulability analysis of a complete system, and the certification of system safety. Common practice today is that tracing garbage collection is never used in DO-178B level A or level B software. However, a small number of projects with level C and D certification requirements have been deployed with traditional Java.

Atego has previously demonstrated that PERC Pico components can be very efficiently and reliably integrated within traditional Java applications. Because PERC Pico is object oriented and undergoes full Java byte-code verification, it is possible to integrate PERC Pico services much more efficiently than is possible with C and JNI. A recent demonstration showed that an all-Java application ran twice as fast as the equivalent program comprised of Java and C based on JNI. See reference 10 for a more complete description of this technology.

The same basic approach that was previously used to integrate PERC Pico with traditional Java applications allows applications written in the style of traditional Java to invoke services provided by either Baseline JSR-302 or Annotated JSR-302. The basic approach is outlined below:

1. During mission initialization, particular mission-resident objects may be placed into a registry that is shared with a traditional Java run-time environment.
2. When a safety-critical object is published in the registry, the safety-critical Java programmer supplies an interface that is implemented by the published object. This interface represents the collection of services that traditional Java is allowed to invoke on the shared object.
3. Traditional Java application code may consult this registry to obtain access to shared objects. A shared object is represented by a proxy within the traditional Java domain. The proxy provides implementations of the interface methods only, hiding all fields and methods of the original object that are not mentioned in the interface definition. The proxy implementation is automatically generated by

infrastructure tools. The automatically generated code maps the implementation of invoked proxy services to the relevant safety-critical Java code.

4. When a mission's initializer publishes objects to be shared with traditional Java, it also allocates one or more (under programmer control) *ManagedThread* objects to act on behalf of the invocations received from the traditional Java application.

    a. For complete generality, it is necessary for a safety-critical Java thread to execute the methods associated with safety-critical objects. This is necessary to assure that the thread implements priorities, synchronization, and priority ceilings in accordance with the protocols of the safety-critical Java environment.

    b. In cases where the body of a safety-critical Java method is known to not require any synchronization with other safety-critical Java threads (as determined through static analysis), the implementation may optimize out coordination with a safety-critical Java *ManagedThread* object. The original traditional Java thread may invoke the method directly and may even in-line the method's implementation if the traditional Java compiler is aware of the safety-critical Java environment's object and method representations.

5. At mission termination time, it is necessary to wait for deactivation of all the traditional Java proxies that may refer to object's residing in this mission's memory. Under programmer control, proxy deactivation can be initiated by the safety-critical Java environment (though a deactivated proxy may continue to exist in the traditional Java domain, any attempts to invoke its services would result in an *IllegalStateException*), or by the traditional Java environment (when the garbage collector reclaims the memory of a proxy, the proxy's *finalize()* implementation communicates with the safety-critical Java infrastructure to advise that this proxy has been deactivated.

6. Acknowledgements

The author owes special thanks to Ales Plsek, Daniel Tang, and Jan Vitek of Purdue University for their efforts in designing and implementing the JSR-302 annotation system, for countless discussions clarifying my understanding thereof, for providing me with early access to their implementation of the annotation checker for experimentation, and for helping me to implement representative code fragments. It is important to emphasize that the shortcomings of the JSR-302 annotation system discussed in this paper are primarily the result of constraints imposed by the JSR-302 expert group rather than the result of their design choices.

7. Bibliography

[1] C. Adams, "DO-178C nears finish line, with credit for modern tools and technologies", in Military and Aerospace Electronics, Nov. 1, 2010.

[2] G. Bollella, B. Brosgol, J. Gosling, P. Dibble, S. Furr, M. Turnbull, *The Real-Time Specification for Java*, Addison Wesley Longman, 195 pages, Jan. 15, 2000.

[3] K. Arnold, J. Gosling, D. Holmes. *The Java™ Programming Language, 4th edition.* 928 pages. Prentice Hall PTR. Aug., 2005.

[4] K. Nilsen, "Differentiating Features of the PERC Virtual Machine", http://www.aonix.com/pdf/PERCWhitePaper_e.pdf

[5] K. Nilsen, *Making Effective Use of the Real-Time Specification for Java*, Atego White Paper, September 2004, available at http://research.atego.com/jsc/rtsj.issues.9-04.pdf.

[6] *Meeting minutes, notes, and preliminary materials related to an early draft specification for safety-critical Java*, available at http://research.atego.com/jsc/index.html.

[7] *PERC Pico User Manual*, Apr. 19, 2008, available at http://research.atego.com/jsc/pico-manual.4-19-08.pdf.

[8] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. Hunt, J. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, A. Wellings. *Safety-Critical Java Technology Specification, Public Draft*, 2011, available at http://www.jcp.org/en/jsr/detail?id=302.

[9] M. Richard-Foy, T. Schoofs, E. Jenn, L. Gauthier, K. Nilsen. "Use of PERC Pico for Safety Critical Java", Conference Proceedings: Embedded Real-Time Software and Systems, Toulouse, France, May 2010.

[10] K. Nilsen. "Improving Abstraction, Encapsulation, and Performance within Mixed-Mode Real-Time Java Applications." Conference Proceedings: Java Technology for Embedded Real-Time Systems, Vienna, Austria, September, 2007.