

Enforcement of Static Properties in Evolving Standards for Safety-Critical and Mission-Critical Java

Kelvin Nilsen, CTO, Aonix

The popular Java programming language has been widely heralded for its programmer productivity gains in a variety of enterprise and desktop applications. Developers often describe Java as a dynamic programming language and many programmer productivity gains result from certain dynamic aspects of the Java platform. Examples of dynamic language features include dynamic class loading and JIT compilation to accelerate the develop-test-debug cycle, and automatic garbage collection to simplify the task of managing dynamic memory allocation. These dynamic features, which are very empowering to Java developers, often stand out in developers' minds as the key reasons why Java allows them to be approximately twice as productive as C++ developers.

It turns out, however, that Java's dynamic features represent only part of the reason that Java developers are more productive than C++ developers. In comparison with C++, the Java compilers and linkers also perform more static property analysis to enforce stronger consistency checking within software systems. This stronger consistency checking reduces programming errors, which further improves developer productivity. Examples of built-in static property enforcement in Java include:

1. Strong type checking prohibits incompatible type coercion between, for example, an integer and a reference type, or between two incompatible reference types.
2. Programmers are prohibited from adding integer values to reference values to create references to new objects.
3. Byte-code verification assures that the types of actual parameters passed to a method invocation exactly match the types of the formal parameters declared for the invoked method.
4. Contexts that invoke methods that might throw "checked exceptions" are required by the compiler and byte-code verifier to deal explicitly with the possibility that an exception might be thrown.

This consistency checking is especially helpful when large software systems are assembled from components that were independently produced by different teams of developers. It turns out that Java productivity benefits are even greater during the integration and maintenance phases of the software life cycle than during the development of new functionality. When compared with C++, Java developers typically see a 5 to 10 fold reduction in the effort, calendar time, and costs incurred during software maintenance.

Because of the strong static property checking already enforced by the Java programming environment, Java has attracted the attention of safety-critical developers as a possible language of choice for implementation of future safety-critical systems. Safety-critical software includes anti-lock braking systems in consumer vehicles, fly-by-wire control of flight surfaces in commercial aircraft, automatic shutdown of nuclear power plants, and computer-controlled switching systems in passenger railroad stations. Various different protocols exist for certifying that software written for these sorts of applications is reliable and correct, depending on the regulating government agency which is responsible for the particular industry. All of the certification techniques share a common goal: to prove through analysis of static properties of the software that it will always run correctly.

A draft specification for safety-critical Java is being developed by the Embedded and Real-Time Systems Forum of the Open Group. This activity will produce a standard that is endorsed both by the Java Community Process and by ISO. This standard is based on the traditional Java platform as extended by the official Real-Time Specification for Java (RTSJ). From this basis, dynamic features such as dynamic class loading, automatic garbage collection, and the RTSJ's concept of ScopedMemory regions are all being removed.

Then, certain enhancements will be added to provide even more static verification than is available in traditional Java. Programmers who develop their code according to the published safety-critical guidelines can rely upon assurances from the safety-critical Java compiler that:

1. The maximum amount of CPU time required to execute particular methods (and all overriding methods) is bounded by a constant that can be derived as a static property of the program and knowledge of the target hardware.
2. The maximum amount of stack memory required to execute particular methods (and all overriding methods) is bounded by a constant that can be derived as a static property of the program and knowledge of the target hardware.
3. Execution of a particular method (or of any overriding method) will not allocate any memory in the shared immortal heap. (Since the safety-critical Java environment has no automatic garbage collector, any objects allocated in the heap are considered to be immortal, because their memory will never be reclaimed.)
4. Execution of a particular method will not result in throwing of RTSJ-defined `CeilingViolationException`, `DuplicateFilterException`, `IllegalAssignmentError`, `InaccessibleAreaException`, `MemoryAccessError`, `MemoryScopeException`, `MemoryTypeConflictException`, `OutOfMemoryError`, `ScopedCycleException`, `StackOverflowError`, or `ThrowBoundaryError` exceptions.

There is not room in this paper to describe all aspects of the safety-critical Java specification. Here, we focus only on the representation and analysis of static properties in two sample safety-critical Java programs.

Programmer Declaration of Static Properties

Figure 1 represents source code for an analyzable bubble-sort program. In this program, Java 5.0 style meta-data annotations and assertions allow developers to specify static properties associated with their programs. The `@StaticAnalyzable` annotation in line 10 denotes that the worst-case execution time and the worst-case memory allocation needs for the `sort()` method introduced on line 11 must be analyzable as a static property of the program. Note that the `@StaticAnalyzable` annotation is accompanied by two attributes, named `enforce_analysis` and `modes`. The `modes` attribute represents an enumeration class that identifies the various modes of operation for this method's execution. Analysis of the method's static resource requirements always depends on the mode specified in the context from which the method is invoked.

Figure 2 shows the declaration of the `AnalysisModes` enumeration. In this example, there are five different modes of operation. The first mode is named `UNBOUNDED`. This represents a situation in which the size of the array passed as an argument to the `sort()` method is not known as a static property of the caller's context. In this mode of operation, the resource needs cannot be analyzed. This is indicated by a `false` value in the first entry of the `enforce_analysis` array. The other modes of operation are named `SMALL`, `BIG`, `SMALL_PRESORTED`, and `BIG_PRESORTED`. In all of these cases, the corresponding initialization value in the `enforce_analysis` array's initializer evaluates to `true`. In the `SMALL` mode of operation, the array argument is assumed to have fewer than 16 elements. In the `BIG` mode of operation, the array argument is assumed to have fewer than 64 elements. `SMALL_PRESORTED` and `BIG_PRESORTED` correspond to similarly sized arrays in which all but one of the elements in the incoming array are known to be in sorted order. In analyzing the CPU time required to complete execution of the `sort()` method, we expect `SMALL_PRESORTED` to execute in less time than `BIG_PRESORTED`, which will execute in less time than `SMALL`, which will execute in less time than the `BIG` mode of operation. Whenever the `sort()` method is called from a context that is itself analyzable, the calling context must define the mode of operation under which it expects the `sort()`

```

[1] package samples;
[2]
[3] import javax.jsc.StaticAnalyzable;
[4] import javax.jsc.Stackable;
[5] import javax.jsc.StaticLimit;
[6] import samples.AnalysisModes;
[7]
[8] public class BubbleSort {
[9]
[10]     @StaticAnalyzable(enforce_analysis = { false, true, true, true, true }, modes = AnalysisModes.class)
[11]     public void sort(@Scoped int a[]) {
[12]         int i, j, k, t;
[13]         int len = a.length;
[14]         boolean sorted = false;
[15]
[16]         for (i = 0, k = len; !sorted && (i < len); i++) {
[17]             assert StaticLimit.IterationBound(AnalysisModes.SMALL, 16);
[18]             assert StaticLimit.IterationBound(AnalysisModes.BIG, 64);
[19]             assert StaticLimit.IterationBound(AnalysisModes.SMALL_PRESORTED, 2);
[20]             assert StaticLimit.IterationBound(AnalysisModes.BIG_PRESORTED, 2);
[21]             k--;
[22]             sorted = true;                                     // assume array is sorted
[23]             for (j = 0; j < k; j++) {
[24]                 assert StaticLimit.NestedIterationBound(AnalysisModes.SMALL, 1, 120);
[25]                 assert StaticLimit.NestedIterationBound(AnalysisModes.BIG, 1, 2016);
[26]                 assert StaticLimit.NestedIterationBound(AnalysisModes.SMALL_PRESORTED, 1, 29);
[27]                 assert StaticLimit.NestedIterationBound(AnalysisModes.BIG_PRESORTED, 1, 125);
[28]                 if (a[i] < a[j]) {
[29]                     assert StaticLimit.NestedIterationBound(AnalysisModes.SMALL_PRESORTED, 1, 15);
[30]                     assert StaticLimit.NestedIterationBound(AnalysisModes.BIG_PRESORTED, 1, 63);
[31]                     t = a[i];
[32]                     a[i] = a[j];
[33]                     a[j] = t;
[34]                     sorted = false;                           // correct assumption if shown invalid
[35]                 }
[36]             }
[37]         }
[38]     }
[39] }

```

Fig. 1. Source Code for Analyzable BubbleSort program

method's resource requirements to be analyzed. This is accomplished by preceding the invocation with an assertion that specifies the intended mode of operation for the called method, as shown in Figure 3.

In this example, the `sort()` method is to be analyzed according to the `SMALL_PRESORTED` mode of operation. Note that the initialization expression for the array (line 11) creates an array with six elements, only

```

[1] package samples;
[2]
[3] public enum AnalysisModes {
[4]     UNBOUNDED,           // can't analyze the most general case
[5]     SMALL,              // array smaller than 16 elements
[6]     BIG,                // array up to 64 elements
[7]     SMALL_PRESORTED,   // small array only one element out of order
[8]     BIG_PRESORTED     // big array only one element out of order
[9] }

```

Fig. 2. Source Code for AnalysisModes enumeration

```

[1] package samples;
[2]
[3] import javax.jsc.StaticAnalyzable;
[4] import javax.jsc.StaticLimit;
[5] import samples.AnalysisModes;
[6] import samples.BubbleSort;
[7]
[8] public class BubbleClient {
[9]
[10]     @StaticAnalyzable public static void main(String args[]) {
[11]         int a[] = {3, 8, 15, 12, 18, 22};
[12]
[13]         assert StaticLimit.InvocationMode(AnalysisModes.SMALL_PRESORTED);
[14]         BubbleSort.sort(a);
[15]     }
[16] }

```

Fig. 3. Source Code for BubbleClient class

one of which is out of sequential order. If the assertion in line 13 of Figure 3 had been omitted, the `sort()` method would have been analyzed according to the rules of the first entry in the `AnalysisModes` enumeration. Since, in this case, the corresponding entry in the `sort()` method's `enforce_analysis` attribute is `false`, the safety-critical Java compiler would reject this program as invalid because the caller's context requires analyzability of all called methods.

Note that the assertions provided in the body of the `sort()` method (Figure 1) are sufficient to limit the number of iterations consumed in each of the analysis modes except for `UNBOUNDED`. The assertion in line 19, for example, denotes that the inner-most nested loop (the one that begins on line 16) will iterate no more than twice if this method is being executed in analysis mode `SMALL_PRESORTED`. The `NestedIterationBound` assertion in line 26 signifies that the body of code that includes this assertion will execute no more than 29 times for each entry into the outer loop at outer-nesting level one if the `sort` method is being analyzed according to the `SMALL_PRESORTED` analysis mode. Similarly, the `NestedIterationBound` assertion at line 29 requires that the enclosing body of the `if-then` statement executes no more than 15 times for each

entry into the outer-nested loop if this method is being analyzed according to the SMALL_PRESORTED mode of execution.

Enforcement of Static Properties

The astute reader may have already observed that it is not possible in the most general case for a compiler to enforce that the programmer supplied assertions are all valid. A programmer might, for example, claim that he is passing a small array to the `sort()` method, when he is really passing an array with more than 1,000 elements. Or a programmer who asserts that a particular loop will execute no more than 100 times for a given configuration of input values might have misrepresented or misunderstood the algorithmic complexity. The primary objectives of the static property analysis in the draft safety-critical Java specification are to enable programmers to make very clear their intentions regarding the static properties associated with particular components, to enable automatic enforcement of consistency between the static properties of independently created components, and to eliminate much of the tedium associated with analyzing worst-case execution times, stack growth, and heap memory allocation.

Enforcement of static software properties in the safety-critical Java specification takes two specific forms:

1. The compiler, byte-code verifier, and linker work together to assure that all programmer-declared assertions are internally consistent. This means if a particular component is declared to be analyzable, any components that it depends on must also be declared to be analyzable and the results of analysis must satisfy all programmer-declared requirements.
2. If the program is compiled with assertions enabled, an assertion exception will be thrown any time the program behaves in such a way as to violate any of the assertions that describe the programmer-intended static properties.

In a safety-critical system, a separate proof system is required to assure that each component satisfies the static properties described in the programmer-supplied annotations. In particular, it must be proven that for each method's available analysis modes, the bounds on loop iterations are valid for every possible set of input values that is consistent with that particular analysis mode. Further, it must be shown that whenever an analyzable method is invoked from within a context that is itself considered analyzable, the invocation is preceded by an appropriate `StaticLimit.InvocationMode()` assertion which "sets" the invoked method's analysis mode to a value that appropriately represents the complexity of executing the invoked method with the input values to be passed as parameters from this context. These proofs are required in all safety-critical development. Some of these proofs can be automated using automatic theorem proving technologies facilitated, for example, by formal JML (Java Modeling Language) annotations. In the most general cases, generation of these proofs cannot be entirely automated because automatic theorem proving of arbitrary static program properties is "not computable". This is why the proposed safety-critical Java language standard restricts its automatic static property analysis to the realm of static property checking and consistency enforcement that can be easily and efficiently solved by compilers and other static analysis tools.

Memory Allocation for Temporary Objects

Since it is as an object-oriented programming language, all structured data is represented by objects. With a traditional Java run-time environment, all objects are allocated within a region of memory known as the heap, and the memory for these objects is reclaimed by an automatic garbage collector. In the proposed safety-critical Java specification, there is no automatic garbage collector because automatic garbage collection is considered to add more complexity and more asynchrony to the execution environment than can be reasonably certified to DO-178B level A standards.

In traditional block-structured languages like Ada, Modula, and Pascal, records (the equivalent of Java's objects) that are declared local to a given procedure are visible within inner-nested procedures and it is not possible to explicitly copy the address of an object into programmer-declared variables. This provides a safe mechanism for stack allocation of temporary objects. But Java is not a block-structured language. Rather, it derives more directly from C and C++ in that it does not support declaration of nested procedures.

In C and C++, programmers can also declare structures that are local to a function. These structures are allocated on the run-time stack and their memory is automatically reclaimed when the enclosing function returns. Since C and C++ do not allow for inner-nested functions, the only way for a function to share access to its stack-allocated local structures is by passing the addresses of these objects to called subroutines in global variables or as input parameters. One of the dangers with this common C and C++ practice is that there is no compiler-enforced protection to assure that references to stack-allocated objects do not outlast the objects to which they refer. With C and C++, it is far too easy to create dangling pointers to objects that no longer exist. Whenever this occurs, these programming errors are among the most difficult to debug.

In the draft safety-critical Java specification, special provisions are made to allow safety-critical Java components to allocate objects on the run-time stack, using programming constructs similar to those of the C and C++ programming languages. Unlike C and C++, statically enforced programmer annotations allow the safety-critical Java compiler to ensure that the use of stack-allocated memory does not create dangling pointers. This is illustrated in Figure 4 and Figure 5.

```
[1] @Scoped Complex x, y, z;  
[2]  
[3]     x = new Complex ((float) 3.5, (float) 1.2);  
[4]     y = new Complex ((float) 2.8, (float) 8.2);  
[5]     z = x.multiply(y);  
[6]     z = z.add(x);
```

Fig. 4. Sample Program with Stack-Allocated Objects

Each of the statements in lines 3 through 6 of Figure 4 allocates a new `Complex` object. Because the variables `x`, `y`, and `z` are each declared to have the `@Scoped` attribute, and because the corresponding constructors and methods are declared to expect scoped parameters, all four of the allocated objects are allocated within the stack activation frame of the method that contains this code.

In Figure 5, the various `@ScopedPure` annotations denote that all of the method's incoming reference arguments, including its implicit `this` argument may refer to objects that reside within the stack-allocated activation frame of some outer-nested method. Upon recognizing this annotation, the safety-critical Java compiler enforces that these incoming arguments are never copied to a variable that might outlive the object to which the reference arguments refer. More specifically, the compiler enforces that:

1. If the value of the incoming reference argument is copied to an outgoing reference argument, the corresponding declaration of the formal outgoing parameter is also declared to have the `@ScopedPure` (or `@Scoped`) attribute.

```

[1] package samples;
[2]
[3] import javax.jsc.*;
[4]
[5] public class Complex {
[6]     public float real, imaginary;
[7]
[8]     public @ScopedPure Complex(float r, float i) {
[9]         real = r;
[10]        imaginary = i;
[11]    }
[12]
[13]    public @CallerAllocatedResult @ScopedPure Complex add(Complex arg) {
[14]        float r, i;
[15]
[16]        r = this.real + arg.real;
[17]        i = this.imaginary + arg.imaginary;
[18]        return new Complex(r, i);
[19]    }
[20]
[21]    public @CallerAllocatedResult @ScopedPure Complex multiply(Complex arg) {
[22]        float r, i;
[23]
[24]        r = this.real * arg.real - this.imaginary * arg.imaginary;
[25]        i = this.real * arg.imaginary + this.imaginary * arg.real;
[26]        return new Complex(r, i);
[27]    }
[28] }

```

Fig. 5. Source Code for Stack-Allocatable Complex class

2. If the value is assigned to an instance field of another Java object, the reference variable that identifies the other Java object must have the `@Scoped` attribute, the assigned field must have the `@Scoped` attribute, and the method that performs the assignment must have the `@AllowCheckedScopedLinks` attribute. By default, the safety-critical compiler prohibits all assignment of scoped references to instance fields. However, in the special case that a programmer adds the `@AllowCheckedScopedLinks` annotation to a method's body, the compiler will allow such assignments. To ensure that the assignments are safe, the compiler will generate code to test:
 - a Whether the assigned value refers to a stack-allocated object, and if so
 - b Whether the object being assigned to also resides on the same run-time stack, and if so
 - c Whether the object being assigned is no deeper on this run-time stack than the object whose reference is being assigned.

A method that is declared with the `@AllowCheckedScopedLinks` attribute will throw a `javax.rtsj.IllegalAssignmentError` exception in the case that an attempt is made to create a reference from an object deep on the run-time stack to an object residing in a more shallow location of the run-time stack. This must be prohibited because the shallow object will eventually be

removed from the stack before the deep object, at which point the deep object would hold a dangling pointer to the destroyed shallow object.

We recognize that the use of checked scoped links is somewhat dangerous. Programmers must exercise discretion and great care whenever they use this programming feature. We expect that many projects will adopt guidelines that prohibit the use of checked scoped links, and the safety-critical Java development tools will have the ability to enforce this prohibition.

The key benefits of safe stack allocation of temporary objects are that:

- Temporary memory allocation and deallocation is very fast.
- Temporary memory allocation is very reliable, because the stack never becomes fragmented and the maximum stack size can be determined through static analysis.
- Compared with the RTSJ's `ScopedMemory` abstractions, the performance and footprint overhead of run-time checks, and the risk that critical program components will be aborted (`IllegalAssignmentError`, `InaccessibleAreaException`, `MemoryAccessError`, `MemoryScopeException`, `ScopedCycleException`) because they violate scoped memory protocols are easily avoided.

This gives the safety-critical Java programmer capabilities comparable to what they are already using in more traditional languages like Ada, C, and C++.

Generalization to Mission-Critical Development

Mission-critical development in Java takes various forms. Today's mission-critical Java application span the spectrum from management-plane control of fiber-optic network switches to electronic trading of stock market transactions. All of the mission-critical Java applications commercially deployed to date are most accurately described as soft real time. These systems generally have timing constraints measured in the ranges from 20 ms to several seconds. While the expectation is that the system will always meet every timing constraint, the realization is that compliance with timing constraints has not been analytically guaranteed, and the systems are designed to respond robustly to the possibility that certain deadlines will occasionally be missed.

The early successes of soft real-time mission-critical system developers using the Java language have piqued the appetite for Java in lower-level hard real-time software as well. One large telecommunication equipment provider recently built a fiber-optic switch comprised of approximately 1 million lines of soft real-time Java code in the management plane combined with roughly 4 million lines of C code in the control plane. One of the difficulties encountered with this architecture is that the C programmers who access Java data structures from their C programs occasionally misunderstand or overlook subtle details of the JNI (Java Native Interface) protocol and end up crashing the whole run-time environment. Given the size and complexity of this application, these sorts of bugs are extremely difficult to isolate and correct. In spite of this one shortcoming, this customer has been very pleased with their choice to use Java for the management-plane functionality, and they intend to propagate more Java into lower levels of their architecture in future versions of their product. In an ideal world, they say they would replace all of the low-level code with Java, but they know that Java does not meet the performance and real-time constraints that are required by the lower layers of their system architecture. For this low-level software, the customer needs a version of Java that is capable of matching the performance and footprint of the C programming language. It turns out that the safety-critical profile that is being standardized within the Open Group, combined with a small number of enhancements to add capabilities that would not be appropriate for safety-critical systems, is exactly what they need.

At Aonix, our two decades of experience supplying Ada technologies for mission-critical and safety-critical development have exposed us to a large breadth of requirements and a wide assortment of different development methodologies. We have found, for example, that over half of the customers who license our safety-critical Ada product Raven use it to develop low-level software that does not need to satisfy safety certification requirements. Given this, we are very comfortable generalizing our safety-critical Java technologies for the development of mission-critical systems as well. In fact, our market research suggests that the safety-critical community strongly prefers this approach because it means the costs of supporting their very specialized needs can be shared with the much larger community of mission-critical developers. Ultimately, this means they get more robust development tools, more supported features and platforms, and more reusable software component libraries for lower prices than would be possible if the hard real-time Java technologies were targeted exclusively to the niche needs of the safety-critical developer.

Four million lines of control-plane software in a telecommunication switch represents one example of low-level high-performance real-time mission-critical code that does not have safety certification requirements. Other examples include device drivers for hard disks, global positioning systems, hardware-accelerated vector graphics engines, radar systems, sonar systems, bomb and missile guidance, encryption for secure communications, digital audio and video capture and playback, hardware sensors, and robot control. In almost all of these applications, it is critical for the low-level software to support very efficient interaction and communication with higher-level software, most of which is best classified as soft real-time. To support robust and scalable composition of software components, the interface between the low-level and higher-level software must support clean abstractions and strong separation of concerns.

One required extension to make the safety-critical Java technologies appropriate for mission-critical systems is the ability to very efficiently and safely share information and control between low-level software written in the style of safety-critical Java and higher-level software written to use traditional J2SE-style Java technologies. The mechanism recommended for this purpose is a generalization of the Ada programming language's protected objects. The idea is that a low-level developer can expose particular immortal objects from within its address space to the traditional Java environment. Strong object-oriented encapsulation, enforced by byte-code verification, is used to restrict the traditional Java environment's access to the internals of the shared object. The traditional Java programmer is not allowed to see the instance variables of the hard real-time object, nor can it execute that object's standard methods. The traditional Java programmer is only allowed to invoke the methods that the hard real-time programmer specifically identifies as visible to the traditional Java environment.

To provide full generality and ease of integration with any Java virtual machine, it is straightforward to glue these hard real-time objects into a traditional Java virtual machine using only the JNI protocol. However, to support high performance information flow between low-level and high-level software, the methods exposed by the hard real-time environment can be translated, optimized, and even in-lined by an enhanced ahead-of-time or just-in-time compiler.

Note that the security provided by this protocol is the same that has been proven in many years of operating system design. The low-level software exposes certain entry points (kernel services) to the high-level application software. The application software may make unreasonable requests of the low-level software, but the low-level software is in full control over its choice to honor or reject the application software's requests. If synchronization with other low-level software is required, only code that was written by the hard real-time developer is allowed to run while the low-level synchronization locks are held. This prevents undisciplined priority inversion that might result if misbehaving Java code interferes with access to hard real-time synchronization locks.

Since this architecture imposes strict limits on how soft real-time code may access hard real-time services, and prohibits hard real-time code from directly accessing objects or invoking services of the traditional soft real-time Java environment, we assure against many of the vulnerabilities that commonly plague JNI software. Since the only synchronization that involves coordination between hard and soft real-time components is carried out under the full control of code written by the hard real-time developer, we avoid many of the vulnerabilities that can compromise the RTSJ's use of wait-free queues.

Conclusion

By restricting the use of Java programming language features and libraries, and by exploiting special static analysis tools, it is possible to apply many of the benefits of the Java programming language to the very specialized domain of safety-critical development and hard real-time mission-critical systems. Standards to support these development approaches are being developed under the auspices of the Open Group, which is working to establish standards that are to be blessed both by the Open Group and by ISO.

Synergy between safety-critical, hard real-time mission-critical, and J2SE-based soft real-time mission-critical approaches places a very powerful set of tools in the hands of real-time mission-critical developers. Each tool is best suited for particular layers in a mission-critical hierarchy. Given the preponderance of Java expertise among new college graduates, of Java development tools, and of reusable Java software components, we believe the capabilities of this integrated technology family will far exceed the sum of capabilities offered by each individual technology in isolation.