Draft Safety Critical Java Standard

Kelvin Nilsen, Ph.D., CTO, Aonix North America

1. Introduction

In order to apply the Java language to the very specialized needs of the safety-critical development community, certain revisions from the traditional Java standard are required. This document outlines a proposed standard set of revisions for adoption both as a JCP and ISO standard. This revision of the document reflects changes motivated by discussions at the Brussels, Belgium Open Group meeting held the week of April 18.

2. Throwing of Exceptions

In traditional Java, the standard practice is to allocate a new object within the heap each time an exception must be thrown. The safety-critical Java domain does not have a garbage collected heap so that practice is not practical. Instead, we recommend the following conventions:

1. Rather than allocate a new exception object each time a given exceptional circumstance is discovered, the "standard library code" will preallocate representative instances of each "standard" exception and will simply throw this preallocated instance when the exception is indicated.
2. Each thread will maintain in thread-local storage a single bounded stack backtrace, representing the context from which the most recently thrown exception originated. This bounded stack backtrace buffer will only represent the most deeply nested N stack frames for the most recently thrown exception of each thread. For a given thread, this buffer will be overwritten each time a new exception is thrown. The application code will query the stack backtrace using the traditional Java API, which associates the stack backtrace information with the thrown exception. When this information is requested of a particular exception object by calling `Throwable.getStackTrace()`, `getStackTrace()` will itself throw `IllegalStateException` if this `Throwable` object is not the one most recently thrown by the current thread.

Discussion. At the Belgium Open Group meeting, we discussed this proposal briefly. There was general agreement that it is not practical to dynamically allocate a new exception object each time an exception must be thrown. However, there was uncertainty as to whether there should exist a single representative exception object to be shared across all threads, or whether each thread should maintain its own collection of preallocated exception objects.

The deciding factor hinges on whether there is useful and important information represented in the "message" portion of the thrown exception. If so, we would want to have separate preallocated exceptions for each thread. If not, we might as well share the same preallocated exception across multiple threads.

We recommend that the decision regarding which exceptions will be shared across threads and which will be preallocated on a per-thread basis be determined on a case-by-case basis, depending on how much personalization is required for each exception object.

3. Threads and Interrupts

At the San Diego Open Group meetings, it was decided that the Safety Critical Java subset would support 28 priorities. For any given platform, the highest N priorities may map to interrupt levels.

A traditional software-scheduled thread is not allowed to set its priority to interrupt level. Only InterruptHandler events, which extend AsyncEventHandler are allowed to run at interrupt level. The byte code verifier assures that the interrupt handling code is execution-time bounded.

Discussion. At the Belgium meeting, we did not reach consensus on whether to allow first-level interrupt handlers to be written in the hard real-time Java subset. Several expressed the opinion that given the need for someone to develop first-level interrupt handlers, and the difficulty of maintaining this code and coordinating the sharing of control and data with higher level "second-level" interrupt handlers, we ought to provide the best possible development environment available to developers of these first-level interrupt handlers. Others expressed concern that allowing Java developers to write first-level interrupt handlers would unduly complicate and compromise the quality of the real-time Java developer's experience. It was decided to leave this topic open until other parts of the safety-critical Java specification stabilize. We do not yet share a common basis for discussion of the cost and benefit analysis regarding this feature.

4. Static Determination of Resource Bounds

We desire that in certain contexts, the byte-code verifier requires that code be time- and resource-bounded. For our purposes, this means that the programmer has followed certain conventions that make it possible to automate analysis of the code in order to determine worst-case execution time. Among restrictions that are imposed by the byte-code verifier are absence of recursion and iteration bounds on every loop.

At the Belgium Open Group meeting, a preference to use programmer annotations rather than simply defining an analyzable subset of Java was expressed. Following is my proposal for such annotations, based on the JDK 1.5 meta-data and assertion facilities. Note that implementation of these J2SE 1.5 enhancements requires minor changes to the JVM, but does not necessarily require that we support all of the 1.5 libraries in our safety-critical subset.

There are several objectives of using annotations to support execution-time analysis:

1. A program component for which annotations indicate that execution time must be analyzable must follow particular style guidelines in order to enable automatic analysis. These style guidelines will be enforced. If an annotated program component cannot be analyzed, the program component does not satisfy the byte-code verifier.
2. System architects should be able to specify that certain program components are required to exhibit certain static properties. It should be possible to reflect these requirements in the source code so as to enable the development environment to enforce that the constraints are satisfied.
3. The results of static property analysis must be available as compile-time constants. The values of these constants may be referenced explicitly in static initialization expressions relating to scheduling parameters and resource budgeting enforcement, for example.

The design of this API is based on the following assumptions:

1. We consider an individual "method" to be the fundamental unit of analyzability.
2. Treat analyzability of code as a property that may be enforced independent of our ability to actually analyze the code.
3. Analysis never depends on run-time information. There can be no parameterization of the analyzer that might depend, for example, on the value of a particular method argument.
4. We expect to support several different approaches towards the use of static annotation information.

a. At bare minimum, we will assure that any method annotated with the @StaticAnalyzable annotation within which the enforce_analysis attribute has true value contains only constructs that can be fully analyzed. Otherwise, the program does not pass byte-code verification.

b. In some environments, it may not be feasible to perform full analysis of static resource requirements, even though the code is considered to be analyzable. Special codes are available to reflect that the analysis has not been performed.

c. When the analysis is performed, it is contingent upon the validity of the programmer's annotations. If the programmer's annotations were incorrect, the static resource analysis will likewise be incorrect. These errors may have global impact, stealing away resources that may have originally been set aside for other components which themselves are well behaved. To deal with the eventuality that the programmer's annotations are erroneous, we expect to support the following:

   i. Full run-time enforcement of all static analysis assertions. We keep a distinct stack of analysis details, including, for example, the mode of each method invocation and the number of times each active loop has iterated. This requires cooperation with the JIT and/or AOT compiler to identify and instrument each loop.

   ii. Partial run-time enforcement of static analysis assertions. For any method that is loaded with assertions enabled, validation will be performed. Any method loaded without assertions enabled will not perform validation. There's a weakness with this approach. If a caller has assertions disabled, but the called method has assertion enabled, the caller can only validate assertions if they do not depend on having a mode passed in on the analysis stack. In this case, a special invalid mode (e.g. value -1) is automatically supplied as the mode value. This default mode value causes the called method to ignore validation even though its own assertions were enabled.

   iii. No run-time enforcement of static analysis assertions. We simply trust that all assertions were correct and valid. In the case of a safety-critical system, the ideal execution mode is no run-time validation of assertions, accompanied by static verification that the assertions are all valid.

Annotations. The annotations are as follows[1]:

@StaticAnalyzable

A method identified with this attribute must conform to certain style guidelines that make it possible to analyze worst-case execution time, stack memory allocation, and heap memory allocation.

@StaticAnalyzable(enforce_analysis = {false})

This method annotation denotes that the method should be analyzed, but does not require that all static properties be analyzable. The analysis infrastructure simply gathers as much information as it can.

```
@StaticAnalyzable(
    enforce_analysis = {false, true, true, true},
    modes = ModeEnumeration.class
    )
```

Assume the following declaration of ModeEnumeration:

1. These annotations are patterned after an annotation style described in "Portable Worst-Case Execution time Analysis Using Java Byte Code", by Guillem Bernat, Alan Burns, and Andy Wellings.

```
public enum ModeEnumeration { UNBOUNDED, SMALL, MEDIUM, LARGE }
```

The method annotation denotes that the method has four different modes of operation. The first mode, identified as UNBOUNDED, cannot be analyzed. The other three modes must be analyzable.

StaticLimit.InvocationMode(Enum callee_mode)

This assertion always returns true. Place this as an assertion within the body of a @StaticAnalyzable method, immediately preceding an invocation of another method to specify that the other method, when called from this particular context, should be analyzed according to its mode callee_mode. If no InvocationMode() assertions precede invocation of the method, it is assumed that the method runs in its default mode, which is always the first mode in its corresponding mode enumeration.

StaticLimit.InvocationMode(Enum caller_mode, Enum callee_mode)

This assertion always returns true. Place this as an assertion within the body of a @StaticAnalyzable method, immediately preceding an invocation of another method to specify that the other method, when called from this particular context, should be analyzed according to its mode callee_mode if this method is being analyzed in mode caller_mode. If no InvocationMode() assertions precede invocation of the method, it is assumed that the method runs in its default mode, which is always the first mode in its corresponding mode enumeration. If multiple InvocationMode() assertions precede invocation of a method, each one having a different value of the caller_mode attribute, the first one to match the caller_mode is applied.

StaticLimit.IterationBound(int max_iterations)

This assertion enforces that the inner-most enclosing loop iterates through this particular statement no more than max_iterations times for each time the loop itself is entered. If this assertion appears nested within conditionally executed code, it serves to limit the number of times the conditionally executed context will execute for each time the enclosing loop is entered.

StaticLimit.IterationBound(Enum analysis_mode, int max_iterations)

This assertion enforces that the inner-most enclosing loop iterates through this particular statement no more than max_iterations times for each time the loop itself is entered when the enclosing method is analyzed according to analysis_mode. If this assertion appears nested within conditionally executed code, it serves to limit the number of times the conditionally executed context will execute for each time the enclosing loop is entered.

StaticLimit.NestedIterationBound(int nesting_level, int max_iterations)

This assertion enforces that the outer nested enclosing loop at nesting level nesting_level iterates through this particular statement no more than max_iterations times for each time the loop itself is entered. If this assertion appears nested within conditionally executed code, it serves to limit the number of times the conditionally executed context will execute for each time the enclosing loop is entered. StaticLimit.NestedIterationBound(0, max_iterations) is identical to StaticLimit.IterationBound(max_iterations).

StaticLimit.NestedIterationBound(Enum analysis_mode, int nesting_level, int max_iterations)

This assertion enforces that the outer nested enclosing loop at nesting level nesting_level iterates through this particular statement no more than max_iterations times for each time the loop itself is entered when the enclosing method is analyzed according to analysis_mode. If this assertion

appears nested within conditionally executed code, it serves to limit the number of times the conditionally executed context will execute for each time the enclosing loop is entered. StaticLimit.NestedIterationBound(analysis_mode, 0, max_iterations) is identical to StaticLimit.IterationBound(analysis_mode, max_iterations).

StaticLimit.NotReached()

This assertion always returns false. It denotes to the execution-time analyzer that this particular statement should never be executed. Place this assertion immediately after a method invocation that is expected to always throw an exception, for example.

StaticLimit.NotReached(Enum analysis_mode)

This assertion always returns false. It denotes to the execution-time analyzer that this particular statement should never be executed when the enclosing method is analyzed according to analysis_mode. Place this assertion immediately after a method invocation that is expected to always throw an exception, for example.

StaticLimit.ArrayLength(byte ba[], int bound)

This assertion appears immediately following an allocation of a byte array. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, byte ba[], int bound)

This assertion appears immediately following an allocation of a byte array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

StaticLimit.ArrayLength(char ca[], int bound)

This assertion appears immediately following an allocation of a character array. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, char ca[], int bound)

This assertion appears immediately following an allocation of a character array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

StaticLimit.ArrayLength(short sa[], int bound)

This assertion appears immediately following an allocation of a short (16-bit) integer array. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, short sa[], int bound)

This assertion appears immediately following an allocation of a short (16-bit) integer array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

StaticLimit.ArrayLength(int ia[], int bound)

This assertion appears immediately following an allocation of a 32-bit integer array. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, int ia[], int bound)
: This assertion appears immediately following an allocation of a 32-bit integer array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

StaticLimit.ArrayLength(float fa[], int bound)
: This assertion appears immediately following an allocation of a 32-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, float fa[], int bound)
: This assertion appears immediately following an allocation of a 32-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

StaticLimit.ArrayLength(long la[], int bound)
: This assertion appears immediately following an allocation of a 64-bit long integer array. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, long la[], int bound)
: This assertion appears immediately following an allocation of a 64-bit long integer array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

StaticLimit.ArrayLength(double da[], int bound)
: This assertion appears immediately following an allocation of a 64-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, double da[], int bound)
: This assertion appears immediately following an allocation of a 64-bit floating point array. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

StaticLimit.ArrayLength(object oa[], int bound)
: This assertion appears immediately following an allocation of an array of references. It establishes an upper bound on the number of entries in the newly allocated array.

StaticLimit.ArrayLength(Enum analysis_mode, object oa[], int bound)
: This assertion appears immediately following an allocation of an array of references. It establishes an upper bound on the number of entries in the newly allocated array when the enclosing method is analyzed according to analysis_mode. If multiple ArrayLength() assertions follow allocation of a new array, all of the assertions that match the current analysis mode are applied.

The @StaticAnalyzable annotation has several additional attributes, not described above. These attributes are to be calculated automatically by the development environment during class loading. The byte-code verifier prevents the programmer from overriding these values. The fields may be consulted by static anal-

ysis tools for purposes of constructing static schedules, and during static initialization for purposes of enforcing resource limits. The fields are:

long [] execution_time()
Represents the nanoseconds required to execute this method's code in each of the identified modes of operation. There is one entry in this array for each of the enumeration constants in the modes() Enum type.

long [] stack_bytes()
Represents the maximum number of bytes of stack growth required during execution of this method for each of the identified modes of operation. There is one entry in this array for each of the enumeration constants in the modes() Enum type.

long [] heap_bytes()
Represents the maximum number of bytes of heap growth required during execution of this method for each of the identified modes of operation. We use the term "heap" loosely. This does not refer to Java's garbage-collected heap. Rather, it refers to the explicitly managed allocation contexts that are used for hard real-time memory allocation. There is one entry in this array for each of the enumeration constants in the modes() Enum type.

boolean [] non_blocking()
Represents whether the method is considered to be non-blocking for each of the identified modes of operation. There is one entry in this array for each of the enumeration constants in the modes() Enum type. If the corresponding entry in this array is true, the method is proven not to block in this particular mode of execution. This means this method can be invoked in this mode from within a context that holds a priority ceiling lock. If the corresponding entry in this array is false, the method has not been proven not to block. Thus, it cannot be reliably invoked from within a context that holds a priority ceiling lock.

There are several constants defined in the StaticLimit class. These are listed below:

public enum DefaultAnalysisMode { CONSERVATIVE }
This is the default enumeration supplied as the value of the modes attribute of an @StaticAnalyzable annotation.

public static final long UNANALYZABLE_TIME
When used to represent the value of a @StaticAnalyzable execution_time attribute, UNANALYZABLE_TIME means the corresponding program component does not follow the guidelines that are required to support automatic analysis of worst-case execution time.

public static final long UNANALYZED_TIME
When used to represent the value of an @StaticAnalyzable execution_time attribute, UNANALYZED_TIME means the corresponding program component does follow the guidelines that are required to support automatic analysis of worst-case execution time, and thus the worst-case execution time is analyzable. However, due to limits of the development environment and/or run-time environment, this program component has not been analyzed.

public static final long UNANALYZABLE_SIZE_BYTES
When used to represent the value of a @StaticAnalyzable stack_bytes or heap_bytes attribute,

UNANALYZABLE_SIZE_BYTES means the corresponding program component does not follow the guidelines that are required to support automatic analysis of worst-case memory allocation needs.

public static final long UNANALYZED_SIZE_BYTES

When used to represent the value of an @StaticAnalyzable stack_bytes or heap_bytes attribute, UNANALYZED_SIZE_BYTES means the corresponding program component does follow the guidelines that are required to support automatic analysis of worst-case memory allocation needs. However, due to limits of the development environment and/or run-time environment, this program component has not been analyzed.

Figure 1 illustrates a sample class file within which the sort() method has been annotated to support static analysis. This program's annotations identify five different modes of operation. In the most general mode, the input array is of unknown size so the execution time cannot be analyzed. In the SMALL and BIG analysis modes, the input array is known to have fewer than 16 and 64 elements respectively. Given these size limits, it is straightforward to bound the execution time. If the size of the array is limited and the contents of the array is known to be entirely sorted except for the possibility that one element is in the wrong position, then the iteration counts can be tightened even further.

Byte Code Verification. The byte code verifier enforces the following rules with respect to the special annotations required to support static analysis of resource requirements:

1. None of the fields of the @StaticAnalyzable meta-data annotation are assigned by the developer except for modes and enforce_analysis. All of the other fields, which are named execution_time, stack_bytes, heap_bytes, and non_blocking, contain their default values.
2. If a given method is declared with the @StaticAnalyzable annotation, any subclass that overrides this method will also be declared with the @StaticAnalayzable annotation and the same modes attribute. Furthermore, the overriding method's enforce_analysis attributes will be true at least for the same analysis modes as the original method.
3. The enumeration class supplied as the argument to the modes attribute must have at least one element.
4. If a method includes any uses of StaticLimit assertions, the method includes a StaticAnalyzable annotation.
5. The arguments to all StaticLimit assertions are compile-time constants.
6. Any mode parameter passed to StaticLimit assertion enforcement is of the type corresponding to the enclosing method's StaticAnalyzable annotation.
7. If a method is annotated with the @StaticAnalyzable annotation, and the method's enforce_analysis attribute is true for a particular analysis mode, the byte-code verifier shall reject the program as illegal if it is not possible to derive upper limits for any of execution_time, stack_bytes, heap_bytes, or non_blocking because the program fails to restrict itself to the analyzable Java subset. In particular:
    a. The number of iterations for every loop is bounded by an appropriate IterationBound() or NestedIterationBound() assertion.
    b. Every method invoked from this method is itself annotated as @StaticAnalyzable and the corresponding execution mode of the invoked method has the enforce_analysis attribute set to true. In case this is a virtual method invocation, all of the subclasses in the class path must be likewise analyzable.
    c. No method invocation is recursive.

```
[1]   import javax.jsc.StaticAnalyzable;
[2]   import javax.jsc.Stackable;
[3]   import javax.jsc.StaticLimit;
[4]
[5]   public class BubbleSort {
[6]     public enum AnalysisModes
[7]       { UNBOUNDED,                        // can't analyze the most general case
[8]         SMALL,                               // array smaller than 16 elements
[9]         BIG,                                        // array up to 64 elements
[10]        SMALL_PRESORTED,               // small array only one element out of order
[11]        BIG_PRESORTED                    // big array only one element out of order
[12]    }
[13]    @StaticAnalyzable(enforce_analysis = { false, true, true, true, true },
[14]                modes = AnalysisModes.class)
[15]    public void sort(@Stackable int a[]) {
[16]      int i, j, k, t;
[17]      int len = a.length;
[18]      boolean sorted = false;
[19]
[20]      for (i = 0, k = len; !sorted && (i < len); i++) {
[21]        assert StaticLimit.IterationBound(AnalysisModes.SMALL, 16);
[22]        assert StaticLimit.IterationBound(AnalysisModes.BIG, 64);
[23]        assert StaticLimit.IterationBound(AnalysisModes.SMALL_PRESORTED, 2);
[24]        assert StaticLimit.IterationBound(AnalysisModes.BIG_PRESORTED, 2);
[25]        k--;
[26]        sorted = true;                                    // assume array is sorted
[27]        for (j = 0; j < k; j++) {
[28]          assert StaticLimit.NestedIterationBound(AnalysisModes.SMALL, 1, 120);
[29]          assert StaticLimit.NestedIterationBound(AnalysisModes.BIG, 1, 2016);
[30]          assert StaticLimit.NestedIterationBound(AnalysisModes.SMALL_PRESORTED,
[31]                                    1, 29);
[32]          assert StaticLimit.NestedIterationBound(AnalysisModes.BIG_PRESORTED,
[33]                                    1, 125);
[34]          if (a[i] < a[j]) {
[35]            assert
[36]              StaticLimit.NestedIterationBound(AnalysisModes.SMALL_PRESORTED,
[37]                                  1, 15);
[38]            assert
[39]              StaticLimit.NestedIterationBound(AnalysisModes.BIG_PRESORTED,
[40]                                  1, 63);
[41]            t = a[i];
[42]            a[i] = a[j];
[43]            a[j] = t;
[44]            sorted = false;                    // correct assumption if shown invalid
[45]          }
[46]        }
[47]      }
[48]    }
[49]  }
[50]
```

Figure 1: Annotated Bubble-Sort Program

8. Each StaticLimit.InvocationMode() assertion specifies an analysis mode for the invoked method which is one of the enumeration constants associated with the @StaticAnalyzable annotation for the invoked method.
9. The modes attribute of the @StaticAnalyzable annotation has at least one element in its enumeration class.
10. Every occurrence of a StaticLimit.InvocationMode() assertion immediately precedes an invocation of a method.
11. For each of the analysis modes that are identified in the @StaticAnalyzable annotation, if the @StaticAnalyzable annotation identifies that the corresponding value of the enforce_analysis attribute is true, the byte code verifier assures that all annotations required to enforce the analysis are present. If not present, the byte code verifier rejects the program as illegal.
12. For any method that is annotated with the @StaticAnalyzable annotation, the byte-code verifier is also responsible for determining the value of each @StaticAnalyzable.non_blocking attribute. These are the rules for identifying a method that is known not to block:
    a. Every method invoked from this method must also be declared with the @StaticAnalyzable annotation.
    b. For every method invoked from this method, its corresponding @StaticAnalyzable.non_blocking attribute must be true.
    c. Any synchronized code that is invoked from within this method, or the methods that it might call, must belong to classes that implement the PCP interface.

Execution Considerations. There are several modes of operation for @StaticAnalyzable annotated program components. In particular:

1. To enforce assertions, we instrument the code as follows:
    a. Upon invocation of a @StaticAnalyzable method, we push onto a special validation stack the mode under which we intend to enforce assertions for the called method. Note that the invocation point does not necessarily include an InvocationMode assertion.
    b. At the beginning of each loop within a @StaticAnalyzable method, we initialize to zero one loop counter variable for each of the IterationCount() assertions that pertain to this loop.
    c. At the point of the corresponding IterationCount() assertions, we increment the count and check to validate that it is less than the specified bound.
2. If the programmer disables assertions for a class that contains a @StaticAnalyzable annotated method, no run-time instrumentation is present during execution of that method.
3. The program can run with assertions disabled. This means no checking is performed at run time to assure that the assertions are valid.
4. When assertions are disabled for some classes, but not for others, the treatment given to occurrences of StaticLimit.InvocationMode() assertions is determined by the assertion status of the callee method rather than of the caller method. This enables checking of StaticLimit assertions in the callee, even if the caller is not enforcing StaticLimit assertions.
5. If a @StaticAnalyzable program component runs with assertions enabled, but the method is invoked from a context that has assertions disabled, it will not be generally possible to validate assertions because the analysis mode for this method's execution is not known.

6. When enforcing assertions, a separate loop count is required for each IterationBound() assertion point, but not for each IterationBound() assertion. For any given execution of the method, only one analysis mode is active at a time. This means that if several IterationBound() assertions appear one after the other, all of the IterationBound() assertions in that cluster can usually be enforced with a single iteration count.
7. To support consistent execution-time analysis across all compliant implementations, we require that for any compliant safety-critical Java implementation, each byte code must be execution-time analyzable for any given static context. This means that instructions such as instance_of and invoke_interface must be execution-time analyzable.

5. Dynamic Memory Management

Because the safety-critical Java environment is intended to run without garbage collection, we do not provide any facility for finalization of objects. In our draft API, java.lang.Object has no finalize() method. Perhaps, a more forceful clarification of our intent would be to provide an empty finalize() method and declare it to be final. Note that, in general, we do not expect safety-critical Java programs to be dynamically allocating and deallocating objects, except for objects that might be allocated and deallocated on a thread's run-time stack. Any special finalization code that might normally be associated with an object's finalize() method can instead be placed in a finally block that will execute when the context that holds a stack-allocated object is exited.

5.1. Stack Allocation

We do not support ScopedMemory in the safety-critical context because its use requires run-time checks that might result in run-time exceptions. The nature of the ScopedMemory protocols is such that proving the absence of run-time exception prior to deployment is computationally undecidable. Thus, it would not be appropriate to use these protocols in a safety-critical system.

Instead, we provide developers with the ability to allocate certain objects on a thread's run-time stack using protocols that can be verified safe at compile time. At the Belgium Open Group meeting, there was general consensus that provably safe stack allocation is desired. However, there was a desire expressed to extend the annotations originally proposed to support:

- The ability to constrain the parameterization of Java interfaces, requiring that implementations of the interface honor certain stack-allocation protocols; and
- The ability to allocate the memory for stack-allocated objects that are returned from certain methods in the stack activation frame of a caller of the method that "allocates", initializes, and returns the object. Furthermore, it was requested that the context within which the memory is allocated be at an arbitrary nesting distance from the syntactic allocation point.

Following is my revised proposal for how to annotate a Java program component to enable safe, efficient, and reliable stack allocation of Java objects.

Programmer Annotations to Support Stack Allocation. The following JDK 1.5 meta-data annotations enable reliable stack allocation of Java objects.

@StackableThis

This method and constructor annotation indicates that the method's treatment of its implicit this argument is consistent with the rules required to allow this to refer to an object that resides on the run-time stack of the currently executing thread.

@StackableResult
This method annotation indicates that the Object to be returned from this method may be allocated on the run-time stack. Besides placing certain restrictions on the body of this method, the presence of this annotation requires that the calling method set aside sufficient memory to hold the method's return result and pass the address of this allocation buffer implicitly to the method.

@StackableArrayResult
This method annotation may be used if a method expects to return an array of references. This annotation indicates that the array object to be returned from this method, along with all of the objects to be directly referenced by this array, may be allocated within the caller's stack activation frame. Besides placing certain restrictions on the body of this method, this presence of this annotation requires that the calling method set aside sufficient memory to hold the reference array to be returned from this method plus enough additional memory to represent each of the objects to be referenced from the returned reference array. The caller passes the address of this allocation buffer implicitly to the method.

@Stackable
This annotation may be applied to instance fields, method parameters, and local variables.

When applied to an instance field, @Stackable signals the intention to create linked data structures out of stack-allocated objects. The contents of a @Stackable field can only be fetched into local or parameter variables that are also declared to be @Stackable. In the most general case, writing to a @Stackable field requires a run-time check to ensure that if the value to be written refers to stack-allocated memory, the object that contains the @Stackable variable resides on the same stack in a more inner scope than the referenced object. In the special case that the enclosing object was just allocated in the current method's activation frame, the run-time check shall be avoided.

When applied to a parameter or local variable, it means the variable's contents can only be assigned to other local or instance variables that are also declared with the @Stackable annotation, or to instance fields that are declared with the @Stackable annotation after performing the appropriate run-time checks. Furthermore, it means that if a new object is created and its reference is directly assigned to one of these variables, the new object will be allocated on the run-time stack.

Byte Code Verification. To support safe, reliable, and efficient stack allocation of memory, the byte code verifier enforces the following restrictions:

1. If a new operation returns its result directly to a variable that is declared to be @Stackable, the corresponding constructor must have been declared with the @StackableThis annotation.
2. Only reference variables may be characterized as @Stackable.
3. If a method has attribute @StackableThis, the method promises not to assign this to an instance field of any Java object unless the reference to the corresponding object was also declared with the @Stackable annotation and a run-time check has verified that the object was allocated in a scope that is more inner-nested than the scope of this object. Consider the following special case:
    a. If the object to which we desire to assign the value of this was just allocated within this method's activation frame, then no run-time check is required. We are assured that if this was stack allo-

cated, the new object is more inner nested than this. (There is also the possibility that this does not refer to a stack-allocated object. That is acceptable.)

4. If a method's parameter is declared with the @Stackable annotation, the method promises not to assign the value of this argument to an instance field of any Java object unless the reference to the corresponding object was also declared with the @Stackable annotation and a run-time check has verified that the object was allocated in a scope that is more inner-nested than the scope of the object referenced from the @Stackable reference argument. Consider the following special case:
    a. If the object to which we desire to assign a copy of the @Stackable reference argument's value was just allocated within this method's activation frame, then no run-time check is required. In this case, we are assured that if the object referenced from the @Stackable argument variable was stack allocated, the newly allocated object is not in a more outer nested context than the context of the object referenced from the @Stackable argument variable. (There is also the possibility that the @Stackable argument variable does not refer to a stack-allocated object. That is acceptable. Note that the analysis corresponding to @Stackable arguments is slightly different than the analysis corresponding to a @StackableThis. The main difference in treatment derives from the fact that the values of argument variables may be overwritten, but overwriting the contents of this is prohibited.)
5. If a method declares one of its local variables with the @Stackable annotation, the method promises not to assign the value of this variable to an instance field of any Java object unless the reference to the corresponding object was also declared with the @Stackable annotation and a run-time check has verified that the object was allocated in a scope that is more inner-nested than the scope of the object referenced from the @Stackable reference argument. Consider the following special case:
    a. If the object to which we desire to assign a copy of the @Stackable local variable's value was just allocated within this method's activation frame, then no run-time check is required. In this case, we are assured that if the object referenced from the @Stackable local variable was stack allocated, the newly allocated object is not in a more outer nested context than the context of the object referenced from the @Stackable local variable. (There is also the possibility that the @Stackable local variable does not refer to a stack-allocated object. That is acceptable.)
6. A method that declares @StackableResult must satisfy the following conditions:
    a. For each return statement within the method, it is reached by exactly one allocating expression that defines the value returned by this expression.
    b. The allocating expression is of one of the three following forms:
        i. result_variable = new <object>(); where <object> represents the declared return type for this method, or
        ii. result_variable = new <object>[N]; where <object>[] represents the declared return type for this method and the immediately following statement is an assertion of one of the following two forms:

```
assert StaticLimit.ArrayLength(result_variable, Max);
```

or

```
assert StaticLimit.ArrayLength(result_variable, Enum mode, Max);
```

where Max represents the maximum number of elements in the allocated array; or

iii. result_variable = methodCall(); where methodCall() is declared to return the same type as this method and it also has the @StackableResult attribute.

In the case that the return type is an array, the method must be declared with the @StaticAnalyzable annotation. Insofar as stack allocation is concerned, it is not necessary for the enforce_analysis attributes to be true. However, it is essential that every allocating expression provide enough StaticLimit.ArrayLength() assertions to fully constrain the array size for every possible execution mode corresponding to this method. This requirement is enforced by the byte code verifier.

c. The allocating expression dominates the return statement. This means that every control path from this method's entry point to this particular return statement must pass through the identified allocating expression.

7. A method that declares @StackableArrayResult must satisfy the following conditions:

a. For each return statement within the method, it is reached by exactly one allocating expression that defines the value returned by this expression.

b. The allocating expression is of one of the three following forms:

i. result_variable = new <object>[N]; where <object>[] represents the declared return type for this method and the immediately following statement is an assertion of one of the following two forms:

```
assert StaticLimit.ArrayLength(result_variable, Max);
```

or

```
assert StaticLimit.ArrayLength(result_variable, Enum mode, Max);
```

where Max represents the maximum number of elements in the allocated array; or

ii. result_variable = methodCall(); where methodCall() is declared to return the same type as this method, and it has the @StackableResult attribute; or

iii. result_variable = methodCall(); where methodCall() is declared to return the same type as this method, and it also has the @StackableArrayResult attribute; or

In the case that the allocating expression is of the form described in paragraph 5.b.i, the enclosing method must be declared with the @StaticAnalyzable annotation. Insofar as stack allocation is concerned, it is not necessary for the enforce_analysis attributes to be true. However, it is essential that every allocating expression provide enough StaticLimit.ArrayLength() assertions to fully constrain the array size for every possible execution mode corresponding to this method. This requirement is enforced by the byte code verifier.

c. The allocating expression dominates the return statement. This means that every control path from this method's entry point to this particular return statement must pass through the identified allocating expression.

d. In the case that the allocating expression is of one of the first two forms (5.b.i or 5.b.ii), there must exist a for loop of the following form which dominates the return statement relative to the allocating expression:

```
for (int i = 0; i < result_variable.length; i++) {

    // arbitrary code, not shown
    result_variable[i] = <allocating-expression>;

    // more arbitrary code, not shown
}
```

We require that the body of this loop not make any assignments (or use the increment/decrement operators) that would modify the value of the loop's counting variable i. Within this loop, there is a single assignment to the expression result_variable[i]. This assignment is dominated by the first instruction of the loop's body and this expression dominates the last instruction in the loop's body. In other words, every iteration of the loop executes the allocation expression and corresponding assignment exactly once. The allocating expression of one of the following two forms:

i. result_variable[i] = new <object>(); where <object> represents the declared element type for the result_variable array, or

ii. result_variable[i] = methodCall(); where methodCall() is declared to return the same type as the declared element type for the result_variable array and it has the @StackableResult attribute.

8. If a method is annotated with the @StackableThis annotation, any method that overrides this method also has the @StackableThis annotation.
9. If a method is annotated with the @StackableResult annotation, any method that overrides this method also has the @StackableResult annotation.
10. If one of the parameters to a method is annotated with the @Stackable annotation, any method that overrides this method also declares the same parameter to have the @Stackable annotation.
11. If an instance variable is declared with the @Stackable annotation, we enforce the following:
    a. The value of this variable is only copied to other local, parameter, or instance variables that are also declared with the @Stackable annotation.
    b. A run-time check protects each write to this variable. This check consists of the following:
        i. Check to see whether the value to be written holds a reference to a stack-allocated object. If not, no further checking is required.
        ii. If so, check to see whether the object to be overwritten was also stack allocated. If not, throw a run-time exception because this violates the sharing protocol.
        iii. Otherwise, check to make sure the object to be overwritten does not reside in a context that is more outer nested than the object whose reference is to be assigned. If it does, throw a run-time exception because this violates the sharing protocol.
12. Given a @Stackable reference to an array of references, each element of the array is treated as if it had been declared as a @Stackable instance variable. This means that the byte code verifier and the run-time environment will collaborate to enforce the restrictions described in paragraph 11 immediately above.

Implementation Suggestions. There are several suggestions for how to efficiently implement the desired capabilities, provided below:

1. Broadly speaking, there are several different classes of programs that perform stack-memory allocation:
   a. For most safety-critical code, it is inappropriate to depend on any run-time enforcement of object sharing rules. Thus, any practice that would require a run-time check to enforce compliance must be prohibited. There should be a byte-code verifier option that forces programmers to use only this subset that can be verified at compile time.
   b. For hard real-time code that is not safety-critical, it is possible to provide more general capabilities by allowing memory management operations that can only be enforced with run-time checks. There should be a byte-code verifier option that allows programmers to use these more general capabilities without performing run-time checks.
   c. For some systems that require run-time checks to enforce compliance with memory management restrictions, it is sometimes desirable to elide the run-time checks after the code has been thoroughly reviewed and tested in order to get better performance and/or smaller memory footprint. We probably should support a build-time option that instructs the AOT compiler to omit generation of the code that performs run-time checks to enforce proper usage of stack-allocated objects.
2. Note that the rules described here allow unbounded stack growth during the execution of a given method. This is because the method might stack-allocate an arbitrary number of objects within a control loop. To prevent this, programmers should exercise care in the design of their stack-based allocation requests. Also, programmers can annotate a method with the `@StaticAnalyzable` annotation, and set the `enforce_analyzable` attributes to `true`. Given that the stack activation frame must be allowed to grow during execution, we expect to use a combination of frame pointer and stack pointer registers, and a stack allocation technique patterned after the C `alloca()` service.
3. Even though certain aspects of a given program component may suggest that a particular new Java object request can be satisfied from the run-time stack, the object will only be stack allocated if all of the required conditions are satisfied. For example, if a method is declared with the `@StackableResult` annotation, but the method's return result is assigned to a variable that does not have the `@Stackable` annotation, the return result will not be stack allocated. Nevertheless, the caller is responsible for setting aside the memory that will hold the method's return result. In this case, the caller sets aside the required memory in the "heap", and it passes the heap adderss to the called method rather than passing the method a pointer to its stack-allocated buffer.
4. Even if assertions may be turned off for a particular method that is declared with the `@StackableResult` or `@StackableArrayResult` annotation, the code generator must insert a check to make sure that the buffer preallocated to hold the result object(s) is large enough to hold the returned object(s). If static analysis of the source code is able to prove that the preallocated buffer is always large enough to hold the return result, then this check may be omitted.

5.2. Sample Programs

The following illegal programs illustrate some of the reasons that a program would be rejected by a byte-code verifier. The program illustrated in Figure 2 is illegal because none of the allocating steps dominates the return statement. In Figure 3, this program violates two of the requirements for legal programs:

1. The return statement is reached by two different allocation expressions.
2. One of the two allocation expressions does not dominate the return statement.

```
[1]    public @StackableResult Object foo(boolean a) {
[2]      @Stackable Object result;
[3]
[4]      if (a) {
[5]        result = new Object();
[6]      }
[7]      ...                                              // other code not shown
[8]
[9]      if (!a) {
[10]       result = new Object();
[11]     }
[12]     return result;
[13]   }
```

Figure 2: Illegal Program with Multiple Allocating Expressions Reaching Return Statement

```
[1]    public @StackableResult Object baz(boolean a) {
[2]      @Stackable Object result;
[3]
[4]      result = new Object();
[5]      if (a) {
[6]        result = new Object();
[7]      }
[8]      ...                                              // other code not shown
[9]      return result;
[10]   }
```

Figure 3: Illegal Program for which Allocating Expression Does Not Dominate Return Statement

The Complex class (See Figure 4) illustrates the use of stackable declarations to allocate certain structured data objects on the stack. Given this definition of Complex, the following code fragment allocates all of its Complex objects on the run-time stack, requiring no allocation from the heap.

```
public void foo() {
     Complex a, b, c, d;

     a = new Complex(3.5, 4.2);
     b = new Complex(5.7, 2.1);
     c = a.add(b);
     d = c.multiply(b);
}
```

The FFT class (See Figure 5) makes use of the Complex class to do fast-Fourier transforms. The math is omitted from this example. The emphasis is on passing of stack-allocated arrays containing references to stack-allocated objects.

6. Locks and Synchronization

At the Belgium meeting, the Open Group decided the following:

```
[1]   package samples;
[2]
[3]   import javax.jsc.*;
[4]
[5]   public class Complex {
[6]     public float real, imaginary;
[7]
[8]     public @StackableThis Complex(float r, float i) {
[9]       real = r;
[10]      imaginary = i;
[11]    }
[12]
[13]    public @StackableResult @StackableThis Complex add(@Stackable 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 @StackableResult @StackableThis Complex
[22]        multiply(@Stackable Complex arg) {
[23]      float r, i;
[24]
[25]      r = this.real * arg.real - this.imaginary * arg.imaginary;
[26]      i = this.real * arg.imaginary + this.imaginary * arg.real;
[27]      return new Complex(r, i);
[28]    }
[29] }
[30]
[31]
```

Figure 4: Annotated Source Code for Class Complex

1. The safety-critical Java standard will not support the `synchronized` statement. The byte code verifier will reject as illegal any program that contains a `synchronized` statement.
2. To eliminate the need to analyze inter-task blocking behaviors, the safety-critical Java subset will not support any blocking mechanisms. In particular:
   a. The safety-critical Java standard will not support the `wait()` or `notify()` methods.
   b. The safety-critical Java standard will not support priority-inheritance-style locking. It will only support the immediate priority ceiling protocol.
   c. The safety-critical Java standard will not support any explicit locking mechanisms. There will be no counting or signaling semaphore and no explicit `Mutex` class in the safety-critical Java subset.
   d. Priority ceiling locks may nest only if the ceiling priorities associated with inner-nested locks are strictly greater than the ceiling priorities of all outer nested locks.[1]
3. A preference was expressed to use the `javax.realtime.MonitorControl` mechanism to specify the ceiling priority of each allocated object. A concern was raised regarding the scalability implications of this approach. It was agreed that we would discuss further at the next meeting.

Discussion. In the absence of semaphores and of Java's wait()/notify() services, we discussed at the Belgium Open Group meeting how one would go about waiting for a particular condition to be satisfied. The suggestion was that condition handling is best handled in the safety-critical Java domain by using an asynchronous event to represent the condition, and using an asynchronous event handler to service the condition. By adopting this approach, analysis of the schedulability of condition handling code becomes a natural extension of the asyncronous event handling schedulability analysis. Note that maximum trigger frequencies can be specified for each sporadic triggering event.

A desire was expressed at the Belgium meeting to not require any kind of syntactic marker to distinguish classes that intend to synchronize with immediate priority ceiling protocol vs. those that intend to synchronize with priority inheritance. Given that the safety-critical Java subset assumes that all synchronization is implemented with immediate priority ceiling protocol, it would appear this is a straightforward issue.

An issue arises, however, when one generalizes the capabilities of the safety-critical Java environment to certain hard real-time mission-critical applications. Because many such systems are much more dynamic than typical safety-critical systems and are likely to exploit Java features such as dynamic class loading, it will be important to allow such systems to use priority inheritance rather than the priority ceiling protocol for certain locks.

In other respects, we desire hard real-time mission-critical Java development to be the same as safety-critical Java development. In particular, we still desire the ability to annotate certain methods as @StaticAnalyzable. Assume that we intend to use a queue-free implementation of the immediate priority ceiling protocol. This requires that no thread block while it holds an immediate priority ceiling protocol lock. Now, suppose we want to verify through static analysis that no thread will attempt to block while holding an immediate priority ceiling protocol lock. In order to perform this analysis, we need to syntactically distinguish between Java synchronization statements that implement priority inheritance and those that implement immediate priority ceiling lock.

Assume further that we desire to reuse software between the safety-critical domain and the hard real-time mission-critical domain. For these reasons, we at Aonix recommend:

1. That there be a syntactic marker rather than a run-time mechanism to distinguish between synchronized statements that use the immediate priority ceiling lock and those that use priority inheritance.
2. That the syntactic marker be associated with the locks that require "special handling" rather than the locks that represent default or traditional handling. In particular, it is the priority ceiling locks that must be handled differently than the priority inheritance objects. The special handling they require consists of:
    a. Defining the priority ceiling for the lock, and
    b. Assuring that the code that executes while holding the lock does not block
3. That this syntactic marker and its associated semantics be inherited by all sub-classes.

1. This particular requirement was not decided, or even discussed, at the Belgium Open Group meeting. At that meeting, we did agree to structure the safety-critical Java subset in such a way that we would not preclude future support for multiprocessor architectures. To assure non-blocking behavior on a uniprocessor system, it is sufficient to require that inner-nested priority ceiling locks have priority no less than, but possibly equal to, the priority of all outer nested locks. However, supporting consistent semantics on multiprocessor machines requires that inner-nested locks be strictly greater than outer-nested locks. Otherwise, threads acquiring priority ceiling locks might deadlock.

```
[1]  package samples;
[2]
[3]  import javax.jsc.*;
[4]
[5]  public class FFT {
[6]    @Stackable Complex input[];
[7]    int array_len;
[8]
[9]    public @StackableThis FFT(@Stackable Complex in_array[]) {
[10]
[11]     array_len = in_array.length;
[12]
[13]     // Note that the following line requires a run-time check.  In
[14]     // particular, if this object was not stack-allocated, but
[15]     // in_array was stack-allocated, the following line would create a
[16]     // pointer from a heap object to a stack-allocated object.
[17]     //
[18]     // In the most general case, any assignment to a @Stackable
[19]     // instance variable will require a run-time check.
[20]     //
[21]     // On the other hand, in certain circumstances, the run-time check
[22]     // can be removed by an optimizing compiler.  For example, if we
[23]     // know from the context within which this constructor was called
[24]     // that the memory for this new object is actually on the run-time
[25]     // stack, then we would not need to emit any run-time checks.
[26]     input = in_array;
[27]   }
[28]
[29]   @StaticAnalyzable(enforce_analysis = {false})
[30]   public final @StackableThis @StackableArrayResult Complex[] doAnalysis() {
[31]     @Stackable Complex result[];
[32]     float computed_real, computed_imaginary;
[33]
[34]     // initialize these variables to silence compiler below
[35]     computed_real = computed_imaginary = (float) 0.0;
[36]
[37]     result = new Complex[array_len];
[38]     assert StaticLimit.ArrayLength(result, 1024);
[39]     for (int i = 0; i < array_len; i++) {
[40]
[41]       // complex arithmethic not shown here, produces computed_real and
[42]       // computed_imaginary values to be assigned to result[i] below.
[43]
[44]       result[i] = new Complex(computed_real, computed_imaginary);
[45]     }
[46]     return result;
[47]   }
[48] }
[49]
```

Figure 5: Annotated Source Code for Class FFT

Perhaps the best compromise between the fully static approach and the fully dynamic (RTSJ) approach is the following:

1. The type of synchronization lock for a newly allocated object is determined by the value returned at the time the new object is instantiated by invocation of:

```
public static jsc.realtime.MonitorControl getMonitorControl();
```

2. At the time a new object is instantiated, if getMonitorControl() returns an instance of jsc.realtime.PriorityCeilingEmulation but the object to be instantiated does not implement the javax.jsc.PCP interface, the constructor terminates by throwing IllegalStateException rather than allocating and initializing the new object.
3. We prohibit the practice of dynamically changing the base monitor control behavior for existing objects by not supporting the following method:

```
public static MonitorControl setMonitorControl(Object obj, MonitorControl policy)
```

4. In the safety-critical Java domain, the only allowed subclass of MonitorControl is PriorityCeilingEmulation.
5. Clarify, as in the Jan. 25 unofficial draft revision of the RTSJ 1.0 that setMonitorControl(MonitorControl policy) and getMonitorControl() apply only to subsequently constructed objects. They have no effect on objects already in existence at the time of their invocation.

The draft API which accompanies this document assumes we are using this compromise solution.

Even this compromise solution introduces a reliability/scalability issue. In particular, it would be desirable to prove through static analysis of a component's source code that the software will not throw a CeilingViolationException. In order to perform this analysis, the ceiling priority of each lock must be statically determined, rather than set at run time. Clearly, there are certain situations in which this is practical, and there are other situations in which this may not be practical because, for example, the same class definition is used in multiple contexts, each one requring a different priority ceiling lock. It would be desirable to offer an additional mechanism whereby programmers would have the option to statically specify their priority ceiling. A convention could be established whereby an object implementing the PCP interface would have the option of providing the following declaration within the class definition:

```
public static final int CEILING_PRIORITY = 87;
```

At the time a new PCP object is instantiated, the run-time environment would consult the current MonitorControl policy to ensure that it is PriorityCeilingEmulation with ceiling priority equal to the value of this variable, if it is present. If this variable is not present, there would be no restriction on the ceiling priority for the newly instantiated PCP object.

Byte Code Verification. The special capabilities described in this secion require the following additional byte code verification checks:

1. Within the safety-critical subset, only objects that are declared to implement the PCP interface are allowed to have synchronized methods.
2. No object that implements the PCP interface is allowed to invoke this.wait() or this.notify().
3. Within the safety-critical subset, there are no synchronized statements, only synchronized methods.

4. Every synchronized method of an object that implements PCP is declared with the @StaticAnalyzable annotation. The byte-code verifier requires that the value of the @StaticAnalyzable.non_blocking attribute be true for all identified analysis modes.
5. If we choose to adopt a convention for allowing static specification of priority ceiling protocol, we would want to require that for any PCP object that provides a declaration of CEILING_PRIORITY, all of its subclasses also specify CEILING_PRIORITY with the same priority. Further, we would want to require that inner-nested priority ceiling synchronization use a higher ceiling priority than outer nested synchronization, or at least encourage the use of a lint-like tool to warn developers whenever it is not possible through static analysis of the program to prove that the program will not throw a CeilingViolationException.

7. Initialization of Class Variables

In traditional Java, class variables are to be initialized "immediately before first use". This requires a run-time check and hinders efficient translation of programs for native execution. Further, it introduces certain race conditions in which the initial values of particular class variables (even the values of certain final variables) depend on the sequence in which classes are accessed (and initialized).

For the safety-critical Java standard, all class variables shall be initialized prior to run time. A smart linker will perform a dependency analysis on all class variables and will initialize variables according to a topological sort of the dependency chain. In the best of cases, all class variables are initialized in the static load image prepared by the intelligent linker. When this is not possible, certain class variables will be "scheduled" by the smart linker to be initialized during system startup, prior to execution of any Java threads. In all cases, the order of the initialization sequence is fully determined by the static linker.

Any dependency cycles will be identified at link time. Certain restrictions must be imposed on developers who are writing initialization expressions for class variables in order to facilitate this semantics.

The following restrictions are recommended. We expect that these instructions will be enforced by the byte-code verifier.

1. Each static (class) variable must be initialized exactly once, outside of any looping or conditional control structures, either with an assignment that is part of the variable's declaration or with an assignment statement within the body of a static initializer block.
2. In either case, all initialization code is restricted according to the following rules:
   a. The code is not allowed to perform any side effects (no synchronization, no I/O, no increment or decrement operations, and no assignments) except for the following:
      i. A single assignment appearing outside of any looping control structures must be provided for each class variable.
      ii. Assignments are allowed to the instance fields of objects that were located in the current evaluation context and which are not reachable from any class variable (i.e. temporary objects).
      iii. Assignments are allowed to the local variables of a static initialization context.
   b. Only final methods may be invoked from within initialization code.
3. Since all methods invoked from within initialization contexts are declared final, it is possible to generate special "translations" of this code. Within an initialization context, all assignments to class and instance variables are treated as if the variables had been declared volatile. This means no code that

syntactically precedes the assignment may be reordered to be executed following the assignment. The most straightforward implementation of initialization uses a simple byte-code interpreter. There is little need to accelerate the initialization code.

4. For each declared class, the compiler separates the class variable initialization declarations into one expression for each variable to be initialized. The linker analyzes the byte code of the assignment expression to determine which other class variables must be initialized prior to evaluation of this expression (the dependencies).
5. For each static class initialization statement, the compiler analyzes the body of the class initialization statement to determine which class variables it initializes and which class variables it depends on. The static class initialization statement is treated as an indivisible unit. This block of code will be scheduled for execution after all dependencies have been resolved. Once executed, any class variables that are defined by this initialization statement will likewise be treated as resolved.
6. When analyzing dependencies, the linker makes no attempt to analyze the dependencies represented by native methods occuring in the initialization expressions. Programmers may supply special annotations to force an ordering on execution of these initialization blocks. The following annotation is supported:

```
class Foo {

    @StaticDependency(Foo.class, "global_C")
    public static int global_A = nativeMethod1();

    @StaticDependency(Foo.class. "global_A")
    public static int global_B = nativeMethod2();

    public static int global_C = nativeMethod3();

}
```

   These annotations require that the initialization expression for Foo.global_C (nativeMethod3()) be executed before the initialization expression for Foo.global_A (nativeMethod1()), which itself must be executed before the initialization expression for Foo.global_B (nativeMethod2()).

7. Because most hard real-time development will involve the use of cross-development tools, it will not be possible in general for a static linker to execute the native code at link time. However, there may be development environments that are able to emulate the deployment platform, and thus might try to run native initialization code in the emulated environment rather than waiting until the system starts up. In the case that a developer knows it would not generally be appropriate to execute certain class initialization (either native code or Java code) prior to run time, the developer may annotate certain variables using the following annotation:

```
@InitializeAtStartup
```

   This annotation indicates that the initialization code associated with the class variable that immediately follows must not be executed until the deployed system begins to execute. In case initialization of other variables depend on the initialization of this variable, execution of the initialization code for those other variables must also be delayed until startup time.

8. For each static initializer block, the linker examines the byte code of the static initializer to determine which other class variables must be initialized prior to evaluation of this expression (the dependencies)

and which class variables are initialized by execution of this static initializer block. If this block does not initialize any class variables, the block is considered dead and a warning is issued by the compiler.

Consider, for example, the following class declarations:

```
[1]    class Foo {
[2]      static int x = 5;
[3]      static float z = x + (new Baz()).code1();
[4]      static Baz b = new Baz(z);
[5]      static double sines[];
[6]      static double cosines[];
[7]
[8]                                  // Source adapted from Flanagan, "Java in a Nutshell", 1996
[9]      static {
[10]       double x, delta_x;
[11]       double local_sines[] = new double[1000];
[12]       double local_cosines[] = new double[1000];
[13]       int i;
[14]       delta_x = (Math.PI/2)/(1000-1);
[15]
[16]            // assignments to local_sines[i] and local_cosines[i] are allowed because these
[17]             // local variables refer to locally allocated objects that are not yet visible to
[18]                                                                      // any class variables.
[19]       for (i = 0; x = 0.0; i < 1000; i++, x+= deltax) {
[20]         local_sines[i] = Math.sin(x);
[21]         local_cosines[i] = Math.cos(x);
[22]       }
[23]
[24]       sines = local_sines;
[25]       cosines = local_cosines;
[26]
[27]               // no further modification of the local_sines or local_cosinesis permitted
[28]                      // because these objects are now visible to "global" class variables
[29]     }
[30]
[31]       etc ...
[32]   }
[33]
[34]   class Baz {
[35]     static float y = (float) Foo.x + Foo.b.code2();
[36]     float my_float;
[37]
[38]     Baz() {
[39]       my_float = Math.PI;
[40]     }
[41]
[42]     final float code1() {
[43]       return my_float;
[44]     }
[45]
[46]     final float code2() {
[47]       return (float) 0.693147;
[48]     }
```

```
[49] }
[50]
[51] class Math {
[52]   static double PI = 3.141519;
[53]
[54]   static final native double cos(double x);
[55]   static final native double sin(double x);
[56]
[57]     etc ...
[58] }
[59]
```

This results in the creation of 6 entries in the initialization dependency graph:

1. Foo.x = 5: no dependencies
2. Foo.b = new Baz(z): depends on Foo.z.
3. Foo.z = x + (new Baz()).code1(): depends on Foo.x, Math.PI.
4. Foo.sines and Foo.cosines are initialized by the static initialization code, lines 9 through 29 inclusive, which depends on Math.PI. Note that execution of this static initialization code depends on the execution of two native methods. Assume, for the purposes of illustration, that the development tools do not have the ability to emulate execution of these native methods at static link time. Thus, we will defer execution of this particular initialization code until run time, even though there is no @InitializeAtStartup annotation that would require it.
5. Baz.y = (float) Foo.x + Foo.b.code2(): depends on Foo.x and Foo.b.
6. Math.PI = 3.141519: no dependencies.

Based on the analysis described above, the JRTK linker builds a dependency graph that relates the various executable blocks together, as shown in Figure 6. The initialization evaluator implements the following algorithm:

1. The Executable_Initializations list represents all of the expressions for which all dependencies have already been resolved (i.e. the value of the dependencies field equals zero). In Figure 6, the expressions on this list are linked through the field illustrated in the top right corner of each expression node. Initially, only the two nodes for which dependencies equals zero are on this list.
2. Initialization proceeds by removing the leading entry from the Executable_Initializations list.
   a. If the corresponding initialization expression can be executed prior to system startup, execute the code associated with that entry, and then decrement the dependencies count for each of the dependants of the evaluated expression. If the new value of the dependencies count after performing the decrement operation equals zero, add this node to the Executable_Initializations list.
   b. Otherwise (this initialization code cannot be executed prior to system startup), place this entry on a different list, named the Startup_Initializations list.
3. Continue this process until the Executable_Initializations list is empty.
4. Now, remove the leading entry from the Startup_Initializations list. Schedule the corresponding initialization code to executed "next" in the startup sequence. Then decrement the dependencies count for each of the dependants of the evaluated expression. If the new value of the dependencies count after performing the decrement operation equals zero, add this node to the Startup_Initializations list.

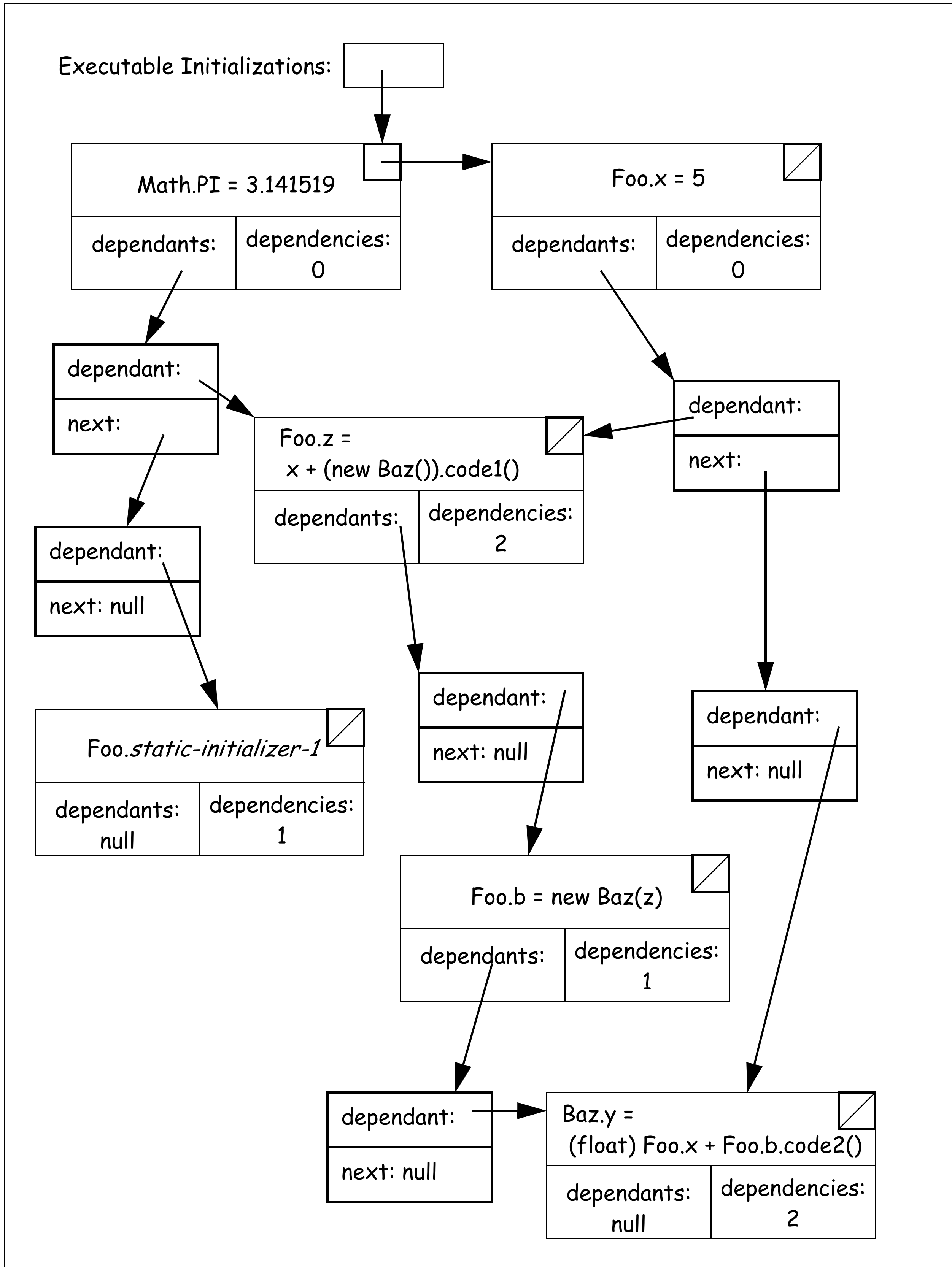

Figure 6: Dependency Graph for Topological Sort

5. Continue this process until the Startup_Initializations list is empty.
6. Upon emptying the Executable_Initializations and Startup_Initializations lists, ask if all expressions have been evaluated. (Keep a count of the total number of expressions to be evaluated and decrement this count each time an expression is evaluated.) If unevaluated expressions remain to be processed, the original program had a dependency cycle in it. The linker issues an error message. The static load image cannot be constructed.

In this particular example, the likely initialization sequence would be as follows:

1. Initialize Math.PI
2. Initialize Foo.x
3. Initialize Foo.z
4. Initialize Foo.b
5. Initialize Baz.y
6. Commit the initial values of the above static class variables to the static load image. Arrange for the startup code to:
    a. Execute Foo.static-initializer-1 before invoking the main method.

Starting up a Safety-Critical Java Application. When a safety-critical Java program begins execution, it first executes whatever sequence of initialization instructions was deferred until startup time. It does this in a single thread which runs at whatever priority was assigned by the host “operating system” when the safety-critical Java environment was “invoked”. This behavior will defer from one operating environment to the next. We consider this aspect of the system startup to be Implementation Defined.

After initialization of static variables completes, the safety-critical Java environment startups up a main NoHeapRealtimeThread running at priority 28 and arranges for this thread to execute the main method of the “initial class”, which should be declared with the following signature:

```
public void main(String args[]);
```

At the time this method begins to execute, this is the only thread running in the safety-critical Java environment. The initial class is specified as part of the configuration of the safety-critical Java environment, using Implementation Defined methods.

Discussion. Note that the @StaticDependency and @InitializeAtStartup annotations are associated with particular variables rather than the initialization expressions themselves. Furthermore, recognize that it is certainly possible to hide all of the initialization of static class variables in native code that is invisible to the smart linker, and thus cannot be analyzed. It is the developer’s responsibility to to coordinate with the smart linker by arranging his or her code in ways that can be analyzed. There are various ways to do this.

Suppose, for example, that you’ve got a single native method that is going to directly initialize an assortment of static class variables. Call these:

```
public static int native_initialized_1, native_initialized_2, native_initialized_3;
```

One way to force the initialization code to execute is by introducing a new placeholder variable, as in:

```
@InitializeAtStartup
private int native_placeholder = the_native_method_that_initializes_everybody();
```

Now, introduce annotations to force this native method to be executed by rewriting the declarations of the first three variables:

```
@StaticDependency(c = TheClassName.class, field = "native_placeholder")
public static int native_initialized_1;

@StaticDependency(c = TheClassName.class, field = "native_placeholder")
public static int native_initialized_2;

@StaticDependency(c = TheClassName.class, field = "native_placeholder")
public static int native_initialized_3;
```

At the Belgium meeting of the Open Group, there was some disagreement as to whether the initial main thread should run at the maximum or minimum priority. The choice is a bit arbitrary, as either default behavior can be easily overriden by calling RealtimeThread.currentRealtimeThread.setPriority() as the first request within this main method. The benefit of the approach currently drafted is that this gives the safety-critical Java code greater assurance that its main code will not be starved by other higher priority activities running outside the safety-critical Java environment.

Byte Code Verification. The following additional byte-code verification is required to support the capabilities described in this section:

1. For each use of the @StaticDependency annotation, the identified class is available as part of the current class path and the named field is a static class variable associated with the identified class.
2. For each use of the @StaticDependency and @InitializeAtStartup annotations, the associated field declaration describes a static class variable.

1. Each static (class) variable is initialized exactly once, either with an assignment that is part of the variable's declaration or with an assignment statement within the body of a static initializer block.
2. In either case, all initialization code is restricted according to the following rules:
    a. The code is not allowed to perform any side effects (no synchronization, no starting of threads, no I/O, no increment or decrement operations, and no assignments) except for the following:
        i. A single assignment appearing outside of any looping control structures must be provided for each class variable.
        ii. Assignments are allowed to the instance fields of objects that were located in the current evaluation context and which are not reachable from any class variable (i.e. temporary objects).
        iii. Assignments are allowed to the local variables of a static initialization context.
    b. Only final methods may be invoked from within initialization code.

8. Memory Model

JSR-133 describes refinements to the Java Memory Model in order to support scalable and portable development of multi-threaded code running on modern hardware. See http://www.cs.umd.edu/~pugh/java/memoryModel/PostPublicReview.pdf. We intend to adopt the recommendations of JSR-133 with the following modifications:

1. The safety-critical byte-code verifier prohibits exposure of `this` to other objects from within a constructor. In other words, a constructor cannot pass the variable `this` to a method of another object. The reason for this prohibition is that if another thread gains access to this object before its constructor has completed initialization of the object, it is possible that other threads might perceive an incorrect value for this object's `final` variables. This restriction is sufficient, though not necessary, to prevent these sorts of data integrity errors.
2. JSR-133 permits code optimizers to remove and/or rearrange synchronization that is considered "useless" according to certain subtle rules. We intend to forbid this practice under the assumption that the real-time programmer carefully designed his locking strategies and the optimizer may not fully appreciate the real-time issues that are at play.

Byte Code Verification. To support these restrictions, we recommend that the byte code verifier enforce the following properties:

1. Within an object's constructor, the value of the `this` variable cannot be assigned to any other variable and cannot be passed as a parameter to any other method. This prohibition also disallowed invocation of virtual methods on the object currently being constructed, because we do not want `this` to be passed as an implicit argument to those methods either.

9. Foreign Language Interfaces

We expect that it will be important for hard real-time Java programs to efficiently and reliably interface with programs written in other languages such as C, C++, and Ada. However, we propose not to standardize the interface at this time. JNI is too large and overly complex. Much of the overhead and complexity of JNI is not necessary in the more limited safety-critical Java environment, because these Java objects are not subject to garbage collection and they are never going to be relocated.

Our recommendation for the moment is to allow use of the `native` keyword as in traditional Java, and to allow each vendor to define their own multi-language native-method interfacing techniques.