

Questions and Answers Regarding Proposed Static Analyzable Memory Model

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

At the New Orleans meetings of the Open Group's Real-Time and Embedded Forum during the week of Oct. 18, several concerns, issues, and questions were raised regarding the memory model previously proposed by Nilsen for the safety-critical Java specification. This document outlines the issues that were raised by representing each in the form of a question, with responses to each question provided by the author of this document. Details of the proposed memory model are provided in appendices to this document.

1. Will programs targeted to the safety-critical profile's memory model run on a "regular RTSJ" machine?

According to the proposed safety-critical memory model, programmers must annotate methods, arguments, and certain fields to identify references that might refer to objects allocated within scoped memory. Consistency of annotations and usage is enforced by a special byte-code verifier. For safety-critical components deployed within a special safety-critical Java run-time environment, the intention is to provide very reliable and efficient implementations of the scoped-memory services, rivaling the performance and reliable memory allocation behavior of current-practice languages like Ada, C, and C++.

It is also possible to deploy these components with a conventional RTSJ run-time, using any one of the following alternative conventions. Regardless of which implementation technique is used, it is guaranteed that the safety-critical software components, when deployed on a conventional RTSJ virtual machine, will not throw any of the exceptions associated with typical scoped-memory usage errors (`IllegalAssignmentError`, `InaccessibleAreaException`, `MemoryAccessError`, `MemoryScopeException`, `ScopedCycleException`).

1. Code written to comply with the stringent static enforcement requirements of the proposed safety-critical Java standard will run as a traditional Java thread (using garbage-collected heap memory to emulate the stack of `ScopedMemory` regions).
2. If each `NoHeapRealtimeThread` has a dedicated `ScopedMemory` region, and `ScopedMemory` regions are not allowed to nest, and all of the code executed by a given thread satisfies the restrictive guidelines enforced by the safety-critical byte-code verifier, this thread will run reliably within the RTSJ run-time environment.
3. A more aggressive memory reclamation model uses nested `ScopedMemory` regions, one for each method that might need to allocate objects. The analysis performed by the safety-critical static byte-code analyzer is capable of determining the required size for each `ScopedMemory` region. A tool to facilitate deployment of safety-critical Java components on a traditional RTSJ platform would insert code to construct appropriately sized `ScopedMemory` regions within each method that so requires. Supporting methods that have the `@CallerAllocatedResult` attribute requires a technique such as one of the following:
 - a. For each invocation of a `@CallerAllocatedResult` method that is not tail recursive, surround the invocation with code patterned after the following template:

```

try {
    save current global caller-allocated-scope in local variable
    set global caller-allocated-scope to the desired allocation scope
      (the ScopedMemory region for this method)
    invoke the method
} finally {
    set the global caller-allocated-scope to the previous value, as represented by
      my "local variable"
}

```

When we speak of a tail-recursive invocation of a `@CallerAllocatedResult` method, we are describing a situation in which a `@CallerAllocatedResult` method U returns the result of a `@CallerAllocatedResult` method V . In this case, there is no need to modify the global caller-allocated-scope information within the method U .

In this model, the transformation tool that supports execution of safety-critical components on a traditional RTSJ platform would take responsibility for allocating all caller-allocated results in the scope represented by the value of the global caller-allocated-scope variable.

- b. For each `@CallerAllocatedResult` method, insert an additional implicit argument to represent the `ScopedMemory` region within which the method's result is to be allocated. Every invocation of a `@CallerAllocatedResult` method must pass the desired allocation scope as a parameter to the method. For each invocation, the desired scope was either passed in from the caller's context, or was newly constructed within this method.

In this model, the transformation tool that supports execution of safety-critical components on a traditional RTSJ platform would take responsibility for allocating all caller-allocated results in the scope represented by the method's implicit caller-allocated-scope parameter.

This is the model assumed in the code transformation examples provided in Appendix C.

2. What is the subset of the RTSJ that is required to support deployment of safety-critical Java components?

To deploy safety-critical components on a conventional RTSJ virtual machine, the RTSJ implementation must support, at minimum, the following services:

- The `MemoryArea` class must support `getMemoryArea()`, `newArray()`, and `newInstance()` operations.
- We must be able to instantiate the `LTMemory` class, which must derive (indirectly) from `MemoryArea`.
- We must be able to reference the `ImmortalMemory` class, which must derive from `MemoryArea`.

In this document, we focus on compatibility with the RTSJ scoped memory model. Discussion of compatibility between threading and synchronization models is treated separately.

The rules enforced by the safety-critical byte-code verifier assure that scoped objects residing in one thread's `ScopedMemory` stack never become visible to other threads. As long as all of the code invoked by a "safety-critical" thread has been verified by the safety-critical byte-code verifier, that software will function reliably without throwing any memory-misuse exceptions.

Note that the mixing of safety-critical software components with undisciplined RTSJ code within the same thread may introduce memory-misuse exceptions even into the safety-critical components. For example:

- If an RTSJ component uses the `MemoryArea.getMemoryArea()` method to obtain a reference to some outer-nested scope and then uses `MemoryArea.newInstance()` to allocate new objects within that outer-nested scope, this unanticipated memory allocation within that outer scope might cause subsequent “safety-critical” allocation requests to overflow the allocation region’s fixed size, resulting in an `OutOfMemoryError` exception.
- If an RTSJ component allocates certain objects within `ScopedMemory` regions and then invokes safety-critical components, passing references to these objects in parameters that were not declared with the `@Scoped` attribute, the preconditions for these safety-critical components will not be satisfied. Thus, it is possible (even likely) that the safety-critical component would perform operations that would result in `IllegalAssignmentError`.
- If an RTSJ component modifies certain linked data structures that had been created by the safety-critical Java components, and places into fields which are not identified with the `@Scoped` attribute references to objects allocated within `ScopedMemory` regions, the preconditions for the safety-critical components will not be satisfied. Thus, it is possible (even likely) that the safety-critical component would perform operations that would result in `IllegalAssignmentError`.

3. I find the requirement that `@CallerAllocatedResult` methods cannot return subclasses of the declared return type too restrictive. Is there any way to relax this requirement?

As originally drafted, we forbid methods that carry the `@CallerAllocatedResult` from returning subclasses of their declared type. It is straightforward to add an optional attribute to the `@CallerAllocatedResult` annotation. This attribute would be an array of `java.lang.Class`, representing types of subclasses that might be allocated in place of the declared return instance. A similar parameter can be added to the `@CallerAllocatedArrayResult` annotation, to allow the individual array elements to be subclasses of the declared array element type.

This proposed revision allows subclasses to be returned, but requires that the method’s annotation enumerate all subclasses that we anticipate returning. The byte-code verifier assures that only subclasses that were listed as attributes of the method’s `@CallerAllocatedResult` annotation are returned from this method or any of the methods that override it.

Note that “refactoring” is required whenever enhancements to existing code introduce a new subclass to be returned from a method that has the `@CallerAllocatedResult` annotation. In particular, the list of subclasses associated with that annotation must be augmented. While this introduces some inconvenience to object-oriented developers, one could argue that this is the only way to force programmers to take into consideration that their introduction of new subclass behaviors may have an impact on the memory resource needs of real-time superclass components for which guaranteed availability of all required memory resources must be demonstrated.

4. How do I allocate memory for run-time exceptions?

It is possible to support allocation of memory for run-time exceptions using a generalization of the techniques used to identify `@CallerAllocatedResult` conventions. Suppose, for example, that we introduce a new `@CallerAllocatedException` annotation. We would need to impose certain restrictions on the methods that might be invoked from within contexts that establish `@CallerAllocatedException` contexts. Some of the particular issues that arise include the following:

- For reliable operation, a caller that volunteers to allocate particular exceptions within its scope needs an upper bound on the number of instances of this particular exception that will be allocated while its allocation context remains active. Conceivably, the same exception could be thrown and caught many times within inner contexts.
- For reliable operation, a caller that volunteers to allocate particular exceptions within its scope needs a limit on how much memory is required to represent the exception when it is thrown. Since the thrown exception's representation includes a stack backtrace, the amount of memory required to represent a dynamically allocated exception depends on the thread's maximum stack depth. For components that are `@StaticAnalyzable`, it is possible to bound the stack depth. But some components of a mission-critical deployment may not be `@StaticAnalyzable`. We need a solution that works both for code that is static analyzable and for code that is not.
- Even when we can "prove" that every thrown exception propagates to the context that corresponds to the `ScopedMemory` context from which it was allocated, that context would need to clear its allocation region before restarting whatever code had aborted with the thrown exception. This forces programmers to use awkward programming conventions in their implementations of fault isolation and recovery.

My personal preference is still to pre-allocate all built-in exceptions, to offer limited thread-local stack backtrace buffers for each thrown exception, and to encourage application developers to adopt similar practices for their user-defined exception handling. I believe this offers the best tradeoffs between ease of programming, efficiency and reliability of implementation, and expressive generality. There is very limited, if any, benefit that comes from dynamically allocation exception objects in safety-critical and mission-critical code.

Nevertheless, if the Real-Time and Embedded Forum feels it is essential to support dynamic allocation of exception objects, I am confident that the proposed annotation framework is sufficiently robust to support a broad spectrum of alternative programming conventions. The next step is for forum members to identify which classes of programming conventions they desire to support.

5. In what upwards-compatible ways can the safety-critical profile be extended to support the more general requirements of mission-critical development?

The driving force behind the design of the proposed safety-critical memory model is to combine the benefits of high performance and high reliability in the implementation of the safety-critical Java's run-time environment. These same benefits are of significant interest to developers of hard real-time mission-critical Java code as well. There are several generalizations of the proposed model which can be made available to developers of mission-critical code. Some of these generalizations are also relevant to safety-critical development:

- For reliable operation of RTSJ software, it is essential to know the required sizes of every `ScopedMemory` allocation context. It is also very important to assure that every required `ScopedMemory` context can be instantiated when it is required. The RTSJ specification does not constrain the representation of `ScopedMemory` objects, nor does it constrain the dynamic memory management techniques for allocation and deallocation of `ScopedMemory` contexts. Specifically, the RTSJ says:

“When a `ScopedMemory` area is instantiated, the object itself is allocated from the current memory allocation context, but the memory space that object represents is not. Typically, the memory for a `ScopedMemory` area might be allocated using native method implementations that make appropriate use of `malloc()` and `free()` or similar routines to manipulate memory.”

Since a traditional RTSJ application allocates and deallocates scopes in unpredictable orders, it is very difficult to guarantee in general that the memory region from which **ScopedMemory** contexts are to be allocated will not become fragmented. Thus, the reliability of this memory management technique is in question. An additional concern with the use of **malloc()** and **free()** to manage memory allocation contexts is the performance overhead, especially if a high percentage of method calls incurs the overhead of creating and destroying **ScopedMemory** contexts.

In the ideal implementation of safety-critical and mission-critical run-time environments, we would propose to encourage the nesting of **ScopedMemory** allocation contexts within outer **ScopedMemory** contexts. This significantly improves both reliability and performance. A single **ScopedMemory** context can serve to represent an entire software “subsystem” which is comprised of multiple threads. Within this initial context, we would allocate memory for the initial thread’s run-time stack and for each method’s **ScopedMemory** context as it is entered. If this software component is comprised of multiple threads, a **ScopedMemory** context is allocated (from within the primordial **ScopedMemory** context) for each of the threads that is spawned by the component’s root thread.

Code written to assume availability of this nested **ScopedMemory** service would run on a conventional RTSJ implementation using non-nested scopes. The only differences between code configured to run on a traditional RTSJ implementation and code configured for the enhanced “safety-critical” or “mission-critical” platforms, would be changes in the required sizes of the **ScopedMemory** regions, which sizes are generally non-portable across different RTSJ implementations anyway. The intent is for static analysis tools to automatically determine the required sizes of these regions. It would be straightforward for these tools to support the option of calculating **ScopedMemory** region sizes assuming system configurations for both nested and non-nested **ScopedMemory** contexts.

- To support reliable integration of nested **ScopedMemory** between a hierarchy of safety-critical or mission-critical threads comprising a software component, introduce a “standard” subclass of **NoHeapRealtimeThread**. For purposes of discussion, call it **MissionCriticalThread**. Provide a constructor for this class that identifies the inner-nested **ScopedMemory** context for this thread and encourage a style of development that allows the parent thread to pass **@Scoped** arguments to the thread’s constructor. These **@Scoped** arguments can be stored in thread-local **@Scoped** instance variables associated with the inner-nested thread. These references provide access to outer-nested scope-allocated data structures that are shared between the various threads that comprise this mission-critical application.

The byte-code verifier requires that construction of this special subclass of **NoHeapRealtimeThread** be accompanied by a **join()** operation to assure that the inner-nested thread terminates its execution prior to departing any outer-nested contexts. In other words, the byte-code verifier assures that any construction of this **NoHeapRealtimeThread** subclass mimics the following pattern:

```
@Scoped shared_object;    // refers to outer-nested ScopedMemory data structures
try {
    scope_id = new LTMemory(MAX_SIZE, MAX_SIZE);
    thread_id = new MissionCriticalThread(scope_id, shared_object);
} finally {
    thread_id.join();
}
```

By enforcing that an outer-nested **MissionCriticalThread** always waits for an inner-nested **MissionCriticalThread** to terminate before destroying any outer-nested **ScopedMemory** contexts, we

enable the following: (1) Very reliable and efficient creation and destruction of scoped-memory contexts, (2) Guaranteed compliance with the single-parent rule without the need for run-time checks, and (3) Guaranteed compliance with referential integrity rules without run-time checks (or with run-time checks only required within methods that specifically enable such by declaring the `@AllowCheckedScopedLinks` annotation).

- For any method that is `@StaticAnalyzable`, the compiler, static analysis tools, and class loader coordinate efforts to determine the maximum size of the `ScopedMemory` region from which this method allocates temporary objects. For methods that are not fully static analyzable, the programmer may provide an additional annotation of the form `@ScopedMemorySize(bytes = 3000)`. If a method is partially analyzable, the system will automatically determine the scope size for the cases that are analyzable, and will use the value of the `@ScopedMemorySize` annotation for all situations that are not analyzable. In the absence of both `@ScopedMemorySize` and `@StaticAnalyzable` annotations, the scope size is zero. Whenever the `ScopedMemory` size is not determined by static analysis, there is a possibility that demand for memory within this scope will exceed the scope's availability of memory. If this happens, the allocation request will throw an `OutOfMemory` error.

We should consider enhancing the parameterization of `@ScopedMemorySize` to support better portability and scalability across RTSJ implementations. Since each implementation will have different object alignment and padding overheads, it would be useful to use a generalization of the `SizeEstimator` class as part of the parameterization of a `@ScopedMemorySize` annotation.

Appendix A: Annotations to Support Static Analysis of Software Component Resource Needs

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 configuration options:

- 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 follows¹:

@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*:

```
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.

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

Represents whether the method is considered to be non-blocking for each of the identified modes

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.

of operation. There is one entry in this array for each of the enumeration constants in the `modes` attribute. If the corresponding entry in this array is `true`, the method is required 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 need not be proven not to block. Thus, it cannot be reliably invoked from within a context that holds a priority ceiling lock. If the value of this attribute is `false` for a particular mode of analysis, an overriding method may require the attribute to be `true`.

`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 analysis 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.

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.

```
[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] }
```

Figure 1: Declaration of **AnalysisModes** Enumeration

Figure 2 illustrates a sample class file within which the `sort()` method has been annotated to support static analysis. Figure 1 provides the declaration of the **AnalysisModes** enumeration that is referenced as an attribute of the **@StaticAnalyzable** annotation. 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 **@StaticAnalyzable** 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:

```

[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 },
[11]                      modes = AnalysisModes.class)
[12]     public static void sort(@Scoped int a[]) {
[13]         int i, j, k, t;
[14]         int len = a.length;
[15]         boolean sorted = false;
[16]
[17]         for (i = 0, k = len; !sorted && (i < len); i++) {
[18]             assert StaticLimit.IterationBound(AnalysisModes.SMALL, 16);
[19]             assert StaticLimit.IterationBound(AnalysisModes.BIG, 64);
[20]             assert StaticLimit.IterationBound(AnalysisModes.SMALL_PRESORTED, 2);
[21]             assert StaticLimit.IterationBound(AnalysisModes.BIG_PRESORTED, 2);
[22]             k--;
[23]             sorted = true;                // assume array is sorted
[24]             for (j = 0; j < k; j++) {
[25]                 assert StaticLimit.NestedIterationBound(AnalysisModes.SMALL, 1, 120);
[26]                 assert StaticLimit.NestedIterationBound(AnalysisModes.BIG, 1, 2016);
[27]                 assert StaticLimit.NestedIterationBound(AnalysisModes.SMALL_PRESORTED,
[28]                                                         1, 29);
[29]                 assert StaticLimit.NestedIterationBound(AnalysisModes.BIG_PRESORTED,
[30]                                                         1, 125);
[31]                 if (a[i] < a[j]) {
[32]                     assert
[33]                         StaticLimit.NestedIterationBound(AnalysisModes.SMALL_PRESORTED,
[34]                                                         1, 15);
[35]                     assert
[36]                         StaticLimit.NestedIterationBound(AnalysisModes.BIG_PRESORTED,
[37]                                                         1, 63);
[38]                     t = a[i];
[39]                     a[i] = a[j];
[40]                     a[j] = t;
[41]                     sorted = false;        // correct assumption if shown invalid
[42]                 }
[43]             }
[44]         }
[45]     }
[46] }
[47]

```

Figure 2: Annotated Bubble-Sort Program

- 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.
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 enforcing 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.
 - d. This method does not invoke `CountingSemaphore.P()` or `SignalingSemaphore.P()`, or `Mutex.lock()`.
 - e. This method does not invoke `Object.wait()`.

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.

Appendix B: Annotations to Support Static Enforcement of Scoped Memory Model

We recommend against direct use of `ScopedMemory` because its use requires extensive run-time checks that might result in run-time exceptions, compromising both performance and software reliability. Though the services we recommend can be implemented using a subset of the full `ScopedMemory` API, the use of `ScopedMemory` services shall be hidden from application code. Specific reasons for prohibiting particular `ScopedMemory` services are discussed here:

- We forbid invocations of `MemoryArea.enter()` and `MemoryArea.executeInArea()` because it is not tractable to statically analyze which memory area will “host” execution of the runnable code. Thus, it cannot be determined in general whether the memory area will have sufficient memory to satisfy the allocation needs of the runnable code.
- We forbid invocations of `MemoryArea.newArray()` and `MemoryArea.newInstance()` because it is not tractable to statically analyze which memory area will supply the memory for the new array or new instance. Thus, it cannot be determined in general whether the memory area will have sufficient memory to satisfy the allocation needs.
- We forbid invocations of `MemoryArea.getMemoryArea()` because there is no need for this service in the absence of the various `MemoryArea` operations that we are forbidding programmers to use. By eliminating this service, we enable various optimizations that allow hard real-time code to run much faster and in much smaller memory than would otherwise be possible.
- We forbid invocation of the virtual methods `getMaximumSize()`, `memoryConsumed()`, `memoryRemaining()`, and `size()` because we do not want program components to concern themselves with dynamic properties of non-local components. Program behavior that depends on the state of non-local scopes is very difficult, if not possible, to statically analyze. We may provide static-method replacements to allow software to obtain the results of calling these methods for the currently active scope only.
- We forbid construction of new `ScopedMemory` instances, because our analysis of total memory requirements and inter-scope relationships depends on being in full control of the sizes and nesting order of all `ScopedMemory` contexts.
- We forbid `setPortal()` and `getPortal()` invocations as this service encourages dynamic binding between program components. These dynamic relationships cannot be readily analyzed with static analysis tools.
- We forbid `getReferenceCount()`, `join()`, and `joinAndEnter()` because these are dynamic services, the behaviors of which cannot easily be analyzed with static analysis tools. Explicit programmer use of these services may interact in undesirable ways with the built-in use of these services to implement the hard real-time capabilities.

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

`@ScopedThis`

(applies to methods and constructors)

This 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. In other words, `this` is never assigned to a variable that would possibly live longer than the object itself.

Questions and Answers Regarding Proposed Static Analyzable Memory Model

When a constructor has this attribute, it means that the newly constructed object may reside within a `ScopedMemory` context. If a constructor is declared to comply with the `@ScopedThis` annotation, the constructor's initialization code executes in the same dynamic scope that represents the newly constructed object. Thus, any objects allocated within the constructor will reside in the same scope as `this`. On the other hand, if a constructor does not have the `@ScopedThis` annotation, a new scope is created for the allocation of temporary objects within the constructor.

`@Scoped`

(applies to instance and static fields, methods, method parameters, and local variables.)

When applied to an instance field, `@Scoped` signals the intention to create linked data structures out of stack-allocated objects. The contents of a `@Scoped` field can only be copied to other variables that are also declared to have the `@Scoped` attribute. In the most general case, writing to a `@Scoped` 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 `@Scoped` variable resides on the same stack in a more inner scope than the referenced object. In the special case that the object containing the `@Scoped` instance field was just allocated in the inner-most active memory scope, the run-time check shall be avoided.

Consider a scenario in which certain real-time classes are dynamically loaded into an outer-nested scope for use only by other components that are more inner-nested on the scope stack. In this scenario, it makes sense to declare that certain static variables associated with this class have the `@Scoped` attribute. The contents of any such variable can only be copied to other variables that are also declared to have the `@Scoped` attribute. In the most general case, writing to a `@Scoped` static 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 `@Scoped` variable resides on the same stack in a more inner scope than the referenced object. In the special case that the object containing the `@Scoped` static variable was just allocated in the inner-most active memory scope, the run-time check shall be avoided.

When applied to a method, this annotation denotes that the method may return a reference to an object that resides in scoped memory. In the most general case, a run-time check is required before returning from the method to assure that the value to be returned resides within a scope that is nested outside the scope of the returning method. In the special case that a reachability data-flow analysis within the method demonstrates that the return value was not allocated within this method or any method it calls, the run-time check shall be avoided. The byte-code verifier assures that any invocation of a method that has this annotation shall assign the result of the method to a variable that has the `@Scoped` attribute.

When applied to a parameter variable, the `@Scoped` attribute means the variable's contents can only be assigned to other local or parameter variables that are also declared with the `@Scoped` annotation, or to instance and static fields that are declared with the `@Scoped` annotation after performing the appropriate run-time checks.

The `@Scoped` annotation may also be applied to local variables for code documentation purposes. However, these annotations are not preserved in class files, so they do not directly affect byte-code verification or code generation. By default, every local reference variable is considered to have the `@Scoped` attribute. The byte-code verifier removes this attribute from a local variable only if its

analysis of the code reveals that (a) this variable's contents is copied to an outgoing argument which was not declared to have the `@Scoped` attribute, (b) this variable's contents is copied to an instance or static variable that does not have the `@Scoped` attribute, (c) this variable's contents is copied to another local or parameter variable that does not have the `@Scoped` attribute, or (d) this variable's contents is returned from this method, and the method does not have the `@Scoped` attribute.

Every `new()` operation that assigns its result directly to a variable or outgoing parameter with the `@Scoped` attribute shall allocate its memory within the scope of the current activation frame. All other allocations shall use `ImmortalMemory`.

`@ScopedArray`

(applies to instance and static variables, parameters, and methods)

This annotation applies only to entities that represent arrays of references. In general, this attribute denotes that the array's elements may themselves refer to objects residing in scoped memory. A `@ScopedArray` variable may only be assigned to other `@ScopedArray` variables. An element of a `@ScopedArray` variable may only be assigned to `@Scoped` variables. Assignments to an element of a `@ScopedArray` object may require a run-time check, depending on the context.

When applied to a method, the `@ScopedArray` annotation denotes that the result returned from the method is an array that may reside in scoped memory, and each entry within the array may also reside in scoped memory.

`@ScopedPure`

(applies to methods and constructors)

This annotation is short-hand to denote that all reference arguments, including `this`, have the `@Scoped` attribute. If any of the arguments refers to an array of references, that argument is considered to have the `@ScopedArray` attribute. `@ScopedPure` further implies that this method, and any method that overrides it, does not have the `@AllowCheckedScopedLinks` attribute. `@ScopedPure` does not imply that the method's result is `@Scoped` or `@ScopedArray`. Neither does it imply that the result is caller allocated (See `@CallerAllocatedResult` and `@CallerAllocatedArrayResult`).

`@CallerAllocatedResult` (subclasses = { <list of sub-class types> })

This method annotation indicates that the `Object` to be returned from this method may be allocated within the allocation context of the caller. 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. The optional list of sub-classes specifies the allowed sub-class types that might be returned from this method. The caller must set aside sufficient memory to represent the largest of these sub-class representations.

`@CallerAllocatedArrayResult` (subclasses = { list of sub-class types })

This method annotation may be used to annotate a method that returns an array of references. This annotation indicates that the array object to be returned from this method, along with all of the objects that are 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, the 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. The optional list of sub-classes specifies the allowed sub-class types that might be assigned to individual array elements. The caller must set aside sufficient memory to represent the worst-case in which every array element is the the largest of the enumerated sub-class types.

@ScopedMemorySize (bytes = <N>)
(applies to methods and constructors)

For methods that are not fully analyzable, this annotation allows programmers to specify the desired size of the method's **ScopedMemory** allocation context, measured in bytes. If a method's body is fully or partially analyzable, the size of the method's allocation context will be determined analytically for all modes that are analyzable. For modes that are not analyzable, the size of the scope will be determined by annotation.

@AllowCheckedScopedLinks
(applies to methods and constructors)

By default, the byte-code verifier rejects any code that would require a run-time check in order to assure integrity of references between objects allocated in **ScopedMemory** regions. There is a class of algorithms, however, for which static analysis of the code cannot prove that all assignments will be legal. If developers find it necessary to use these algorithms, they must annotate any method that might contain an assignment operation that needs to be checked with this annotation. With this annotation present, the byte-code verifier will allow checked assignment operations. To further draw attention to the risks associated with inclusion of code that might include an illegal assignment operation, the byte-code verifier requires that any method with this annotation is declared to throw **IllegalAssignmentException**. Note that this is different than **IllegalAssignmentError**, which is an unchecked exception.

Constructors. Constructors require special implementation techniques because they are sometimes called upon to initialize objects within outer-nested scopes (when invoked to instantiate a caller-allocated result). To support the desired generality, all of the temporary objects allocated within a constructor are allocated in the same **ScopedMemory** context as the object being constructed (represented by **this**). If the constructor invokes other methods, and those methods allocate temporary memory, that temporary memory is allocated in the local context of that method, and will generally be discarded before the constructor returns. If a constructor invokes a method that is declared to support caller-allocated results, and the constructor assigns the result of that method to a **@Scoped** variable, the allocation context for the invoked method is the same as the allocation context for **this**.

Static Initializers. When a class is first loaded and/or initialized, certain static initialization code must execute. In the safety-critical and mission-critical run-time environments, much of this initialization can be performed prior to deployment, as controlled by an intelligent static linker. For any static initialization code that must be deferred until execution time, the run-time environment establishes a distinct **ScopedMemory** context for execution of each body of static initialization code associated with a class. Thus, temporary objects allocated within the **ScopedMemory** context will be discarded following execution of each block of static initialization code.

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 **@Scoped**, the corresponding constructor must have been declared with the **@ScopedThis** annotation.
2. Only reference variables may be characterized as **@Scoped** or **@ScopedArray**.
3. An array that is declared to have the **@ScopedArray** attribute is considered to also have the **@Scoped** attribute.
4. Insofar as byte-code verification is concerned, there are no annotations associated with local variables. A local variable of type array of references is considered to be **@ScopedArray** until shown otherwise. A local variable of other reference type is considered to be **@Scoped** until it is shown otherwise. Local variables are shown not to have the scoped attribute by one of the following:
 - a. If the variable's value is copied to an instance or static field, or to an outgoing argument, or returned from the method and the corresponding recipient of the copy is not labeled with the **@Scoped** or **@ScopedArray** attribute, this variable is known not to have the corresponding attribute.
 - b. If an element of a reference array is copied to an instance or static field, or to an outgoing argument, or is returned from the method and the corresponding recipient of the copy operation is not labeled with the **@Scoped** or **@ScopedArray** attribute, this reference array is known not to have the **@ScopedArray** annotation. It may still be **@Scoped**. That question must be addressed separately.
5. If a method has attribute **@ScopedThis**, the implicit **this** argument is treated as a **@Scoped** variable within the body of the method.
6. Incoming arguments, instance fields, and static fields that are declared to have the **@Scoped** attribute are treated as having the **@Scoped** attribute.
7. A return statement is treated as an assignment to an implicit outgoing return parameter. The return parameter is considered to have the **@Scoped** attribute only if the method is declared to have the **@Scoped** or **@ScopedArray** attribute.
8. The byte-code verifier forbids the contents of a **@Scoped** variable from being assigned to a local variable, an instance or static field, an outgoing parameter, or returned from the method unless the destination of the assignment operation is also considered to have the **@Scoped** attribute.
9. The byte-code verifier forbids the contents of a **@ScopedArray** array element from being assigned to a local variable, an instance of static field, an outgoing parameter, or returned from the method unless the destination of the assignment operation is also considered to have the **@Scoped** attribute.
10. Assignment of a **@Scoped** value to an instance or static field that has the **@Scoped** attribute requires in the most general case a run-time check to assure that the assigned value refers to an object that is more outer nested on the scope stack than the object that contains the field to be assigned. The run-time 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 reference sharing protocol.

The byte-code verifier rejects any code that requires a run-time check unless the method that contains the code was declared to have the `@AllowCheckedScopedLinks` attribute. The following special cases do not require run-time checks and shall be allowed by the byte-code verifier even within methods that do not have the `@AllowCheckedScopedLinks` attribute:

- a. If the object that contains the field to be assigned was just allocated within this method's inner-most `ScopedMemory` context, we are assured that any values assigned to this field reside in the same or outer-nested contexts. This conclusion is based on reaching definitions analysis within the body of code for the current method only.
 - b. If the value to be assigned was copied from another reference field (or array element) of the same object, we are assured that the assigned reference value refers to an object residing in the same or outer-nested context. This conclusion is based on reaching definitions analysis within the body of code for the current method only.
 - c. If the value to be assigned was copied from another reference field (or array element) of an object that was reachable from from the object that is being assigned to, we are assured that the assigned reference value refers to an object residing in the same or outer-nested context. This conclusion is based on reaching definitions analysis within the body of code for the current method only. [This rule may need some further clarification to precisely characterize the set of programs that is considered to be "legal".]
 - d. If a reference value being returned from a method was passed in as an argument to the method, or was reachable from one of the objects passed in as arguments to the method, we are assured that the returned value is visible in the scope of the caller's method. This conclusion is based on reaching definitions analysis within the body of code for the current method only.
11. Because of support for caller-allocated results, constructors cannot in general assume the object being constructed was allocated from the inner-most nested `ScopedMemory` context. Some constructors may contain code that is only considered legal if the object being constructed is allocated from within the most inner-nested scope. The byte code verifier shall reject any use of such a constructor in a context that does not guarantee to be executing in the inner-most nested scope. Specifically, this refers to constructors that require the `@AllowCheckedScopedLinks` annotation to be generally applicable to all situations. If this annotation is missing, it may be that the constructor's code is legal (and does not require run-time assignment checks) only when constructing an object in the inner-nested scope. In this case, the byte-code verifier shall allow this constructor's use only in such contexts.
12. A method that declares `@CallerAllocatedResult` 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 `@CallerAllocatedResult` 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.

13. A method that declares `@CallerAllocatedArrayResult` 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 `@CallerAllocatedResult` attribute; or
- iii. `result_variable = methodCall();` where `methodCall()` is declared to return the same type as this method, and it also has the `@CallerAllocatedArrayResult` attribute; or

In the case that the allocating expression is of the form described in paragraph 13.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 (13.b.i or 13.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 is 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 `@CallerAllocatedResult` attribute.
14. If a method is annotated with the `@ScopedPure` annotation, any method that overrides this method also has the `@ScopedPure` annotation.
 15. If a method is annotated with the `@ScopedThis` annotation, any method that overrides this method also has the `@ScopedThis` annotation.
 16. If a method is annotated with the `@CallerAllocatedResult` annotation, any method that overrides this method also has the `@CallerAllocatedResult` annotation. The overriding method may subset from the original method's set of allowed subclass types for the return result.
 17. If a method is annotated with the `@CallerAllocatedArrayResult` annotation, any method that overrides this method also has the `@CallerAllocatedArrayResult` annotation. The overriding method may subset from the original method's set of allowed subclass types for the return result.
 18. If one of the parameters to a method is annotated with the `@Scoped` annotation, any method that overrides this method also declares the same parameter to have the `@Scoped` annotation.

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

```

[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 3: 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 4: Illegal Program for which Allocating Expression Does Not Dominate Return Statement

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 (e.g., forbidding use of the `@AllowCheckedScopeLinks` annotation).

- b. In large bodies of safety-critical and mission-critical code, system designers may desire to enforce the restriction that none of this code allocates from `ImmortalMemory`. A byte-code verification option should be available to allow this restriction to be selectively enforced for particular classes and methods, using byte-code verification tools to perform the enforcement.
 - 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. This might apply both to array subscript checking and assignment checking. 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

```
[1] package samples;
[2]
[3] import javax.jsc.*;
[4]
[5] public class Complex {
[6]     public double real, imaginary;
[7]
[8]     public @ScopedThis Complex() {
[9]         real = imaginary = 0.0;
[10]    }
[11]
[12]    public @ScopedThis Complex(double r, double i) {
[13]        real = r;
[14]        imaginary = i;
[15]    }
[16]
[17]    public @CallerAllocatedResult @ScopedPure Complex add(Complex arg) {
[18]        double r, i;
[19]
[20]        r = this.real + arg.real;
[21]        i = this.imaginary + arg.imaginary;
[22]        return new Complex(r, i);
[23]    }
[24]
[25]    public @CallerAllocatedResult @ScopedPure Complex subtract(Complex arg) {
[26]        double r, i;
[27]
[28]        r = this.real - arg.real;
[29]        i = this.imaginary - arg.imaginary;
[30]        return new Complex(r, i);
[31]    }
[32]
[33]    public @CallerAllocatedResult @ScopedPure Complex multiply(Complex arg) {
[34]        double r, i;
[35]
[36]        r = this.real * arg.real - this.imaginary * arg.imaginary;
[37]        i = this.real * arg.imaginary + this.imaginary * arg.real;
[38]        return new Complex(r, i);
[39]    }
[40]
[41]    public @CallerAllocatedResult @ScopedPure Complex multiply(double s) {
[42]        return new Complex(real*s, imaginary*s);
[43]    }
[44]
[45]    public @CallerAllocatedResult @ScopedPure Complex conjugate() {
[46]        return new Complex(real, -imaginary);
[47]    }
[48]
```

Figure 5: First Part of Annotated Source Code for Class **Complex**

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.

```
[49] public @ScopedPure Complex cloneInPlace(Complex source) {
[50]     real = source.real;
[51]     imaginary = source.imaginary;
[52]     return this;
[53] }
[54]
[55] // turn this complex number into its conjugate
[56] public @ScopedPure Complex conjugateInPlace() {
[57]     imaginary = -imaginary;
[58]     return this;
[59] }
[60]
[61] public @ScopedPure Complex addInPlace(Complex arg) {
[62]     real += arg.real;
[63]     imaginary += arg.imaginary;
[64]     return this;
[65] }
[66]
[67] public @ScopedPure Complex subtractInPlace(Complex arg) {
[68]     real -= arg.real;
[69]     imaginary -= arg.imaginary;
[70]     return this;
[71] }
[72]
[73] public @ScopedPure Complex multiplyInPlace(Complex arg) {
[74]     real = real * arg.real - imaginary * arg.imaginary;
[75]     imaginary = real * arg.imaginary + imaginary * arg.real;
[76]     return this;
[77] }
[78]
[79] public @ScopedPure Complex multiplyInPlace(double s) {
[80]     real *= s;
[81]     imaginary *= s;
[82]     return this;
[83] }
[84] }
```

Figure 6: Second Part of Annotated Source Code for Class **Complex**

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 `@CallerAllocatedResult` annotation, but the method's return result is assigned to a variable that does not have the `@Scoped` 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 address to the called method rather than passing the method a pointer to its stack-allocated buffer.

```
[1] public void foo() {  
[2]     Complex a, b, c, d;  
[3]  
[4]     a = new Complex(3.5, 4.2);  
[5]     b = new Complex(5.7, 2.1);  
[6]     c = a.add(b);  
[7]     d = c.multiply(b);  
[8] }
```

Figure 7: Sample Stack Allocating Code

4. Even if assertions may be turned off for a particular method that is declared with the `@CallerAllocatedResult` or `@CallerAllocatedArrayResult` 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.

Sample Programs

Figures 3 and 4 serve to illustrate some of the reasons that a program would be rejected by a byte-code verifier. The program illustrated in Figure 3 is illegal because none of the allocating steps dominates the return statement. The program in Figure 4 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.

The `Complex` class (See Figure 5 and Figure 6) illustrates the use of `@Scoped` declarations to allocate certain structured data objects on the stack. Given this definition of `Complex`, the code fragment shown in Figure 7 allocates all of its `Complex` objects on the run-time stack, requiring no allocation from the heap.

The `ComplexFFT` class (See Figures 8 through 12) makes use of the `Complex` class to do fast-Fourier transforms. This code was adapted from a publicly available C++ template math library. We provide this as an example of the sort of high-performance computation that might be required in real-time digital audio, sonar, and radar applications. Note that temporary objects are allocated at several levels within this software hierarchy. All of these temporary objects can be stack allocated.

```
[1] package samples;
[2]
[3] import java.lang.Math;
[4]
[5] import javax.jsc.Scoped;
[6] import javax.jsc.ScopedArray;
[7] import javax.jsc.ScopedThis;
[8] import javax.jsc.ScopedPure;
[9]
[10] import javax.jsc.PreallocatedExceptions;
[11]
[12] import javax.jsc.StaticLimit;
[13] import javax.jsc.StaticAnalyzable;
[14]
[15] /**
[16]  * This sample code translated/derived from C++ template library
[17]  * available at http://www.jjj.de/fft/cplxfft.h
[18]  */
[19]
[20] public class ComplexFFT {
[21]     int N, log2N; // these define size of FFT buffer
[22]     @ScopedArray Complex w[]; // array [N/2] of cos/sin values
[23]     @Scoped int bitrev[]; // bit-reversing table, in 0..N
[24]     @Scoped double fscales[]; // f-transform scalings
[25]     @Scoped double iscales[]; // i-transform scales
[26]
[27]     static final int MAX_ARRAY_SIZE = 4096;
[28]     static final int LOG_MAX_ARRAY_SIZE = 12;
[29]     static final int INNER_LOOP_BOUND = 1+2+4+8+16+32+64+128+256+512+1024+2048;
[30]     static final int INNER_INNER_LOOP_BOUND =
[31]         (MAX_ARRAY_SIZE / 2) * (LOG_MAX_ARRAY_SIZE - 1);
[32]
```

Figure 8: First Part of Annotated Source Code for Class **ComplexFFT**

```

[1] public @StaticAnalyzable @ScopedPure ComplexFFT(int size, double scalef1,
[2]           double scalef2, double scalei1, double scalei2) {
[3]     int i,j,k;
[4]     double t;
[5]
[6]     // Our ability to analyze this code assumes array length < MAX_ARRAY_SIZE
[7]     assert (size <= MAX_ARRAY_SIZE);
[8]
[9]     fscaler = new double[2]; assert (StaticLimit.ArrayLength(fscaler, 2));
[10]    iscaler = new double[2]; assert (StaticLimit.ArrayLength(iscaler, 2));
[11]    fscaler[0] = scalef1; fscaler[1] = scalef2;
[12]    iscaler[0] = scalei1; iscaler[1] = scalei2;
[13]
[14]    for (k = 0; ; ++k){
[15]        if ((1<<k) == size) break;
[16]        if (k==14 || ((1<<k) > size)) // size not a power of two
[17]            throw PreallocatedExceptions.IllegalArgumentException;
[18]    }
[19]    N = size;
[20]    log2N = k;
[21]
[22]    bitrev = new int [N];
[23]    assert (StaticLimit.ArrayLength(bitrev, MAX_ARRAY_SIZE));
[24]    if (k > 0) {
[25]        w = new Complex [N / 2];
[26]        assert (StaticLimit.ArrayLength(w, MAX_ARRAY_SIZE / 2));
[27]    }
[28]    else
[29]        w = null;
[30]
[31]    bitrev[0] = 0;
[32]    for (j = 1; j < N; j = j * 2){
[33]        assert StaticLimit.IterationBound(LOG_MAX_ARRAY_SIZE);
[34]        for (i = 0; i < j; ++i){
[35]            assert StaticLimit.NestedIterationBound(1, INNER_LOOP_BOUND);
[36]            bitrev[i] <<= 1;
[37]            bitrev[i+j] = bitrev[i]+1;
[38]        }
[39]    }
[40]
[41]    if (k > 0) {
[42]        k = (1<<(k-1));
[43]        for (i = 0; i < k; ++i) {
[44]            assert StaticLimit.IterationBound(MAX_ARRAY_SIZE / 2);
[45]            t = (double) (bitrev[i<<1]) * Math.PI / (double) k;
[46]            w[i] = (new Complex (Math.cos(t), Math.sin(t))).conjugateInPlace();
[47]        }
[48]    }
[49] }

```

Figure 9: Second Part of Annotated Source Code for Class **ComplexFFT**

```
[1] // perform forward fft on buffer
[2] public @StaticAnalyzable @ScopedPure final void fft(@ScopedArray Complex buf[])
[3] {
[4]     fft_func( buf, true );
[5] }
[6]
[7] // perform reverse fft on buffer
[8] public @StaticAnalyzable @ScopedPure final void ifft(@ScopedArray Complex buf[]) {
[9]     fft_func( buf, false );
[10] }
[11]
[12] public @StaticAnalyzable @ScopedPure final int length() {
[13]     return N;
[14] }
[15]
[16] public @StaticAnalyzable @ScopedPure
[17]     void hermitian(@ScopedArray Complex buf[]) {
[18]
[19]     int i, j;
[20]     if (N <= 2) return; // nothing to do
[21]     i = (N>>1)-1; // input
[22]     j = i+2; // output
[23]
[24]     // This code is copied verbatim, Kelvin thinks this should be i >= 0.
[25]     while (i > 0) {
[26]         assert StaticLimit.IterationBound(MAX_ARRAY_SIZE / 2);
[27]         buf[j].cloneInPlace(buf[i]);
[28]         buf[j].conjugateInPlace();
[29]         --i;
[30]         ++j;
[31]     }
[32] }
```

Figure 10: Third Part of Annotated Source Code for Class **ComplexFFT**

```

[1] private @StaticAnalyzable @ScopedPure
[2]     void fft_func(@ScopedArray Complex buf[], boolean fwd_flag) {
[3]     int i, j, k;
[4]     int buf0_offset, buf2_offset;
[5]     Complex z1, z2, zw;
[6]     double sp[], s;
[7]
[8]     z1 = new Complex();
[9]     z2 = new Complex();
[10]    zw = new Complex();
[11]
[12]    sp = fwd_flag ? fscales : iscales;
[13]    s = sp[0]; // per-pass scale
[14]    if (log2N == 0) { // only 1 element !
[15]        s = sp[1]; // final scale only
[16]        buf[0].multiplyInPlace(s);
[17]        return;
[18]    }
[19]
[20]    k = N / 2;
[21]    if (log2N == 1)
[22]        s *= sp[1]; // final scale
[23]
[24]    buf2_offset = k;
[25]    for (i = 0; i < k; ++i) { // first pass is faster
[26]        assert StaticLimit.IterationBound(MAX_ARRAY_SIZE / 2);
[27]        z1.cloneInPlace(buf[i]).addInPlace(buf[buf2_offset + i]);
[28]        z2.cloneInPlace(buf[i]).subtractInPlace(buf[buf2_offset + i]);
[29]        buf[i].cloneInPlace(z1.multiplyInPlace(s));
[30]        buf[buf2_offset + i].cloneInPlace(z2.multiplyInPlace(s));
[31]    }
[32]
[33]    if (log2N == 1) return; // only 2!
[34]    k = k / 2; // k is N/4 now
[35]    for (; k != 0; k >>= 1) {
[36]        assert StaticLimit.IterationBound(LOG_MAX_ARRAY_SIZE - 1);
[37]        if (k == 1) { // last pass - include final scale
[38]            s *= sp[1]; // final scale
[39]        }

```

Figure 11: Fourth Part of Annotated Source Code for Class **ComplexFFT**

```

[1]     buf0_offset = 0;
[2]     for (j = 0; buf0_offset < N; ++j) {
[3]         assert StaticLimit.NestedIterationBound(1, INNER_LOOP_BOUND);
[4]         zw.cloneInPlace(w[j]);
[5]         if (!fwd_flag)
[6]             zw.conjugateInPlace();
[7]         buf2_offset = buf0_offset + k;
[8]         for (i = 0; i < k; ++i) {           // a butterfly
[9]             assert StaticLimit.NestedIterationBound(2,
[10]                INNER_INNER_LOOP_BOUND);
[11]             z1.cloneInPlace(zw).multiplyInPlace(buf[buf2_offset + i]);
[12]             z2.cloneInPlace(buf[buf0_offset + i]).addInPlace(z1);
[13]             buf[buf2_offset + i].
[14]                 cloneInPlace(buf[buf0_offset + i]).subtractInPlace(z1).multiplyInPlace(s);
[15]             buf[buf0_offset + i].cloneInPlace(z2).multiplyInPlace(s);
[16]         }
[17]         buf0_offset += (k * 2);
[18]     }
[19] }
[20]
[21] for( i = 0; i < N; ++i ){
[22]     assert StaticLimit.IterationBound(MAX_ARRAY_SIZE);
[23]     j = bitrev[i];
[24]     if( i <= j ) continue;                // don't do these
[25]     z1.cloneInPlace(buf[i]);
[26]     buf[i].cloneInPlace(buf[j]);
[27]     buf[j].cloneInPlace(z1);
[28] }
[29] }
[30] }

```

Figure 12: Fifth Part of Annotated Source Code for Class **ComplexFFT**

Appendix C: Transformation of Safety-Critical Code to Traditional RTSJ Code

A safety-critical (or mission-critical) Java program written according to the restrictive set of guidelines described in this document can be translated into a traditional RTSJ program using straightforward transformations. These transformations do not represent the most efficient or most reliable deployment. Nevertheless, they serve to demonstrate theoretical equivalency of approaches, and provide a mechanism to allow safety-critical and mission-critical developers to develop and test their code on traditional RTSJ platforms. Note, in comparing the original code with the transformed code for the `Complex` and `ComplexFFT` classes, the expressive power and programmer convenience afforded by use of the proposed safety-critical annotations.

```
[1] package samples;
[2]
[3] import javax.jsc.*;
[4] import javax.rtsj.LTMemory;
[5] import java.lang.Class;
[6]
[7] public class XComplex {
[8]     public double real, imaginary;
[9]
[10]    private final static java.lang.reflect.Constructor complex_constructor;
[11]
[12]    static
[13]    {
[14]        Class double_type;
[15]        java.lang.reflect.Constructor x;
[16]
[17]        x = null;
[18]        try {
[19]            double_type = java.lang.Double.TYPE;
[20]            // Note: auto creation of temporary array for argument list
[21]            x = XComplex.class.getConstructor(double_type, double_type);
[22]        } catch (Throwable t) {
[23]            System.exit(-1);
[24]        } finally {
[25]            complex_constructor = x;
[26]        }
[27]    }
[28]
[29]    public XComplex() {
[30]        real = imaginary = 0.0;
[31]    }
[32]
[33]    public XComplex(double r, double i) {
[34]        real = r;
[35]        imaginary = i;
[36]    }
[37]
```

Figure 13: First Part of RTSJ Transformation of `Complex` Class

```

[1] private class $setup_add implements Runnable {
[2]     XComplex method_object;
[3]     LTMemory mem_context;
[4]     XComplex arg;
[5]     XComplex result;
[6]     $setup_add(XComplex method_object, LTMemory mem_context, XComplex arg) {
[7]         this.method_object = method_object;
[8]         this.mem_context = mem_context;
[9]         this.arg = arg;
[10]    }
[11]    public void run() {
[12]        result = method_object.$$add(mem_context, arg);
[13]    }
[14] }
[15] public XComplex add(LTMemory mem_context, XComplex arg) {
[16]     $setup_add f;
[17]     LTMemory inner_scope;
[18]     final int SCOPE_SIZE = 256;    // determined by static analysis
[19]     f = new $setup_add(this, mem_context, arg);
[20]     inner_scope = null;
[21]     try {
[22]         inner_scope = new LTMemory(SCOPE_SIZE, SCOPE_SIZE, f);
[23]     } finally {
[24]         for (;;) {
[25]             try {
[26]                 if (inner_scope != null)
[27]                     inner_scope.join();    // join() should return immediately
[28]                 break;
[29]             } catch (InterruptedException x) {
[30]                 continue;    // retry join() until it completes successfully
[31]             }
[32]         }
[33]     }
[34]     return f.result;
[35] }
[36] private XComplex $$add(LTMemory mem_context, XComplex arg) {
[37]     double r, i;
[38]     Object args[];
[39]
[40]     r = this.real + arg.real;
[41]     i = this.imaginary + arg.imaginary;
[42]
[43]     args = new Object [2];
[44]     args[0] = new Double(r);
[45]     args[1] = new Double(i);
[46]     return (XComplex) mem_context.newInstance(complex_constructor, args);
[47] }
[48]

```

Figure 14: Transformation of the **Complex add()** method

Consider, for example, translation of the `Complex` class shown in Figures 5 and 6. Excerpts of its translation to traditional RTSJ code are shown in Figures 13 and 14. The code shown in lines 9 - 27 of Figure 13 was inserted as part of the transformation process. This code is required to initialize the static `complex_constructor` variable, which is used in the transformations of the various methods that have caller-allocated results.

Figure 14 provides the transformation of the `add()` method. The original code is represented by lines 15, 37, and 39 through 42. Note that an extra argument has been inserted into the argument list for this method (see line 15). This argument, which represents the `ScopedMemory` context in which to allocate the caller-allocated result, is passed implicitly by every invocation of the `add()` method. In other words, the transformation process inserts this allocation-context argument into every invocation of a method that is declared to return a caller-allocated result.

The original implementation of the `add()` method has been copied into the body of the `$$add()` method in the transformed code. Note that what used to be a simple constructor invocation has been replaced with an invocation of the outer-nested `LTMemory` object's `newInstance()` method. Note also that the conventions of the Java reflection API require allocation of several temporary objects in order to parameterize the invocation of `newInstance()`. These temporary objects serve no useful purpose after the constructor has completed initialization of the new `XComplex` object. Thus, their memory will be reclaimed upon return from `$$add()`.

The private `$setup_add` class provides a mechanism to establish a new allocation context within which temporary objects can be allocated by the `$$add()` method. All of the memory for these temporary objects will be reclaimed upon return from `$add()`. Note that the RTSJ's convention to require each scope transfer to pass through the `run()` method of a `Runnable` class forces us to copy arguments and results into and out of intermediate instance variables. We do not show the translations of the `subtract()`, two `multiply()`, and `conjugate()` methods as each of these translations is in the same general form as the translation of `add()`. The translations of the `cloneInPlace()`, `conjugateInPlace()`, `addInPlace()`, `subtractInPlace()`, and two `multiplyInPlace()` methods are all identical to the original implementations of these methods. Since these methods do not allocate memory, no special transformations are necessary.

Treat these transformations as theoretical proofs that the safety-critical and mission-critical Java standards are equivalent in capability to a subset of the full RTSJ. Further, this exercise serves to prove that, with appropriate transformational tools, it is possible to develop and test "critical" components on any compliant RTSJ implementation. At the same time, it is important for programmers to realize that the ideal implementation of `@Scoped` allocation within the safety-critical and mission-critical domains is much more efficient than what is represented by these transformations. Given the restrictive guidelines that govern programmer behavior in the safety-critical and mission-critical profiles, it is possible to implement very lightweight `ScopedMemory` services in which memory allocation regions are part of each method's run-time stack's activation frame. The resulting efficiency will be comparable to stack allocation in languages like C, C++, and Ada.

Consider also the translation of the `ComplexFFT` class into a `XComplexFFT` class for execution in a traditional RTSJ environment. As with the transformation of `Complex`, `XComplexFFT` also introduces a declaration and static initialization expression for the `complex_constructor` variable. This is shown in Figure 15. Note the use of this variable at line 44 of Figure 17.

The translated constructor is shown in Figures 16 and 17. In a typical scenario, memory for the construction of a new `XComplexFFT` object is allocated within the inner-most `ScopedMemory` context. However,

```

[1] private final static java.lang.reflect.Constructor complex_constructor;
[2] static
[3] {
[4]     Class double_type;
[5]     java.lang.reflect.Constructor x;
[6]
[7]     x = null;
[8]     try {
[9]         double_type = java.lang.Double.TYPE;
[10]         // Note: auto creation of temporary array for variable-length argument list
[11]         x = XComplex.class.getConstructor(double_type, double_type);
[12]     } catch (Throwable t) {
[13]         System.exit(-1);
[14]     } finally {
[15]         complex_constructor = x;
[16]     }
[17] }
[18]
[19] private class $setup_XComplexFFT implements Runnable {
[20]     XComplexFFT method_object;
[21]     int size;
[22]     double scalef1, scalef2, scalei1, scalei2;
[23]
[24]     $setup_XComplexFFT(XComplexFFT method_object, int size,
[25]         double scalef1, double scalef2,
[26]         double scalei1, double scalei2) {
[27]         this.method_object = method_object;
[28]         this.size = size;
[29]         this.scalef1 = scalef1;
[30]         this.scalef2 = scalef2;
[31]         this.scalei1 = scalei1;
[32]         this.scalei2 = scalei2;
[33]     }
[34]
[35]     public void run() {
[36]         method_object.$$XComplexFFT(size, scalef1, scalef2,
[37]             scalei1, scalei2);
[38]     }
[39] }

```

Figure 15: Declarations in XComplexFFT Translation of ComplexFFT

in cases where the constructor is called to allocate the result to be returned from a method that is declared with the `@CallerAllocatedResult` annotation, the object `this` will have been allocated in a more outer-nested `ScopedMemory` context. Therefore, in the transformed constructor, each of the `new` operations performed within the constructor is directed to allocate memory from the same memory scope that holds the object referenced by the constructor's `this` argument.

The constructor for `XComplexFFT` contains a loop (lines 38 through 45 of Figure 17) that initializes the instance variable `w`. Each iteration through the loop constructs a new `XComplex` object. Each construction

```

[1] public XComplexFFT(int size,
[2]                     double scalef1, double scalef2,
[3]                     double scalei1, double scalei2) {
[4]     $setup_XComplexFFT f;
[5]     LTMemory inner_scope;
[6]     final int SCOPE_SIZE = 256;    // determined by static analysis
[7]
[8]     f = new $setup_XComplexFFT(this, size, scalef1, scalef2, scalei1, scalei2);
[9]     inner_scope = null;
[10]    try {
[11]        inner_scope = new LTMemory(SCOPE_SIZE, SCOPE_SIZE, f);
[12]    } finally {
[13]        for (;;) {
[14]            try {
[15]                if (inner_scope != null)
[16]                    inner_scope.join();
[17]                break;
[18]            } catch (InterruptedException x) {
[19]                continue;    // retry join() until it completes successfully
[20]            }
[21]        }
[22]    }
[23] }
[24]

```

Figure 16: Transformed Constructor Code for **ComplexFFT**

of an **XComplex** object requires the allocation of a temporary two-element array and two **Double** objects. All of the memory for these temporary objects will be reclaimed when control returns from the constructor.

Since the constructor for **XComplexFFT** requires allocation of temporary objects, the transformation of the constructor introduces a **Runnable** class to wrap the invocation of the constructor. This allows us to establish a new **ScopedMemory** context for the purpose of allocating temporary objects that are needed only during execution of the constructor. The wrapper class is named **\$setup_ComplexFFT**. It is shown in Figure 15.

There are no transformations for the **fft()**, **ifft()**, **length()**, and **hermitian()** methods because none of these methods allocate memory. The **fft_func()** method allocates three temporary objects (on lines 8 through 10 of Figure 11). Thus, its implementation is transformed according to the patterns already demonstrated for the **Complex.add()** method and **ComplexFFT** constructor. Its translation is not shown.

```

[1] private void $$XComplexFFT(int size, double scalef1, double scalef2,
[2]                             double scalei1, double scalei2) {
[3]     int i, j, k;
[4]     double t;
[5]
[6]     fscales = (double []) MemoryArea.getMemoryArea(this).newArray(Double.TYPE, 2);
[7]     iscales = (double []) MemoryArea.getMemoryArea(this).newArray(Double.TYPE, 2);
[8]     fscales[0] = scalef1; fscales[1] = scalef2;
[9]     iscales[0] = scalei1; iscales[1] = scalei2;
[10]
[11]     for (k = 0; ; ++k){
[12]         if ((1<<k) == size) break;
[13]         if (k==14 || ((1<<k) > size)) // size not a power of 2
[14]             throw PreallocatedExceptions.IllegalArgumentException;
[15]     }
[16]     N = size;
[17]     log2N = k;
[18]     bitrev = (int [])
[19]         MemoryArea.getMemoryArea(this).newArray(java.lang.Integer.TYPE, N);
[20]     if (k > 0) {
[21]         w = (XComplex [])
[22]             MemoryArea.getMemoryArea(this).newArray(XComplex.class, N/2);
[23]     }
[24]     else
[25]         w = null;
[26]
[27]     bitrev[0] = 0;
[28]     for (j = 1; j < N; j = j * 2){
[29]         for (i = 0; i < j; ++i){
[30]             bitrev[i] <<= 1;
[31]             bitrev[i+j] = bitrev[i]+1;
[32]         }
[33]     }
[34]
[35]     if (k > 0) {
[36]         Object args[];
[37]         k = (1<<(k-1));
[38]         for (i = 0; i < k; ++i) {
[39]             t = (double) (bitrev[i<<1]) * Math.PI / (double) k;
[40]             args = new Object[2];
[41]             args[0] = new Double(Math.cos(t));
[42]             args[1] = new Double(Math.sin(t));
[43]             w[i] = (XComplex) MemoryArea.getMemoryArea(this).
[44]                 newInstance(complex_constructor, args).conjugateInPlace();
[45]         }
[46]     }
[47] }
[48]

```

Figure 17: Transformed Helper Function for **ComplexFFT** Constructor