Using Java in Safety-Critical Applications

ISO 26262 Certification for Real-Time Java Code

White Paper

Wolf-Dieter Heker

Rainer Koellner

Verocel, GmbH

TABLE OF CONTENTS

LIST OF FIGURES

1 Abstract

This document provides an analysis of the concerns with using Java, in particular real-time Java, in safety-critical applications. The focus is on hard real-time applications subject to certification based on ISO 26262. Both safety and timeliness are considered.

As ISO 26262-6:2011 is not specific on activities regarding object-oriented (OO) languages, this document refers to guidance provided for real-time applications in avionics, which have similar certification requirements. The avionics standard RTCA/DO-178C has an entire supplement on object-oriented technology and related techniques called RTCA/DO-332. DO-332 is used in this document to supplement the lack of detail for object-oriented technology in ISO 26262.

Furthermore, this document outlines a road map of necessary activities for using the *aicas* JamaicaVM implementation of real-time Java in a safety-critical environment.

2 References

- ISO 26262-6:2011 (E)
  Road vehicles — Functional safety,
  Part 6: Product Development at the Software Level
- RTCA/ DO-332, Object-Oriented Technology and Related Techniques Supplement to DO-178C and DO-278A; RTCA, Inc., December 13, 2011
- [HRTGC]
  Hard Real-time Garbage Collection in Modern Object-Oriented Languages: Fridtjof Siebert, aicas GmbH; 2002
- Java
  The Java® Language Specification Java SE 8 Edition; Oracle America, Inc. and/or its affiliates; 2015
- [RTSJ]
  Real-time and Embedded Specification for Java; Editor James J. Hunt, aicas GmbH
  Version 2.0, Draft 59, General Relativity Edition, March 20, 2017
- [SCJ]
  Safety-Critical Java Technology Specification, JSR 302, Editor Doug Locke, Version 0.110, Draft, February 2, 2017

3 Introduction

Though conventional Java is fine for applications that do not have a time bound, i.e., do not have real-time requirements, it should not be used for control applications. Therefore, this study confines itself to real-time Java. Real-time Java is defined as a Java implementation that conforms to the RTSJ specification and has a real-time garbage collector (GC).

Certification of real-time Java requires surmounting a number of challenges that procedural languages do not have. These challenges are identified below, along with a means of overcoming them. Since certification can only be done on a specific real-time Java implementation, the JamaicaVM has been chosen for this purpose. This means the study must address issues that are specific to JamaicaVM. This is rounded out with a proposal for verifying applications that run on JamaicaVM and its runtime environment, with special emphasis on memory management systems.

This study does not distinguish between safety levels; rather it assumes the highest assurance level: ASIL D.

Although there is ongoing work on specifying Safety-Critical Java, it was not considered here for three reasons. First, the specification is not yet complete. Second, it would severely restrict the language, as only a small number of standard classes are available. Finally, it only supports scoped memory, as stack-like memory management technique, which also severely restricts what Java libraries can be used. These restrictions drastically limit the compatibility with conventional Java, which is deemed unacceptable for the scope of this paper. Instead, the semantic refinements and additional APIs of the Real-time and Embedded Specification for Java ([RTSJ]) are considered as providing the extensions necessary for real-time response.

4 What is different with Real-time Java?

There are several differences between Java, both conventional and real-time, and procedural, such as C and Ada, that are usually used for safety-critical applications. Each of these is addressed in turn in this paper. Since C++, another object-oriented language is also considered appropriate for safety-critical applications; a comparison is also drawn with C++, highlighting pros and cons.

4.1 General aspects of OO

The OO paradigm is an excellent approach to mastering the complexity of applications. The idea of classes and object was designed to help reduce coupling between different parts of a system. The issues discussed below are essentially due to application complexity rather than due to the OO approach itself.

Note that these issues might even apply when OO paradigms are used for applications implemented in C where the dynamic binding is ‘hand-crafted’, using pointers to functions or switch/case constructions.

When an application needs to distinguish between different actions, an OO implementation will express this with a class hierarchy and dynamic binding. In a non-OO approach, the actions would be selected by equivalent switch/case construction or by using function pointers; OO is simply a more systematic approach which makes the need for consistency obvious in the class hierarchy. See the “Example for OO paradigm implemented by switch” provided in Appendix B.

4.1.1 Class Hierarchy

The class hierarchy must be consistent. While the language semantics make sure that any operation defined in a class is also available in all subclasses, the designer of the class hierarchy must make sure these operations are semantically consistent regarding pre-conditions and post-conditions in accordance with the Liskov Substitution Principle (LSP) (examples are also given in DO-332 or found in Wikipedia). In short, the Liskov Substitution Principle requires the following:

- Pre-conditions cannot be strengthened in a subtype;
- Post-conditions cannot be weakened in a subtype; and
- invariants of the supertype must be preserved in a subtype.

Adherence to the above criteria requires systematic testing (or analysis of the hierarchy).

4.1.2 Dynamic binding

Object-oriented applications typically make heavy use of dynamic binding, because using a class hierarchy without dynamic binding does not exploit the powers of OO. Thus, when looking at the source code, one does not know exactly which member function is being called. This makes the analysis of the source code more complicated, even for calls where only a single implementation is available, dynamic binding will often be used.

Even when the implementation of the class hierarchy is shown to be consistent, adhering to the LSP, the execution time can vary between different implementations of the same operation. This makes it harder to specify the execution time of operations in class hierarchies. Since dispatching is used instead of branching across various types, internal variation is replaced with variant across the methods that can be called as a result of the dispatch.

The coverage analysis for dynamically bound method calls should also demonstrate the correctness of dispatch tables.

Invoking interface methods requires a search process because of possible multiple inheritance[1]. The search process makes the determination of the execution time more complicated. In general, the dispatch time is proportional to the number of interfaces a class implements. This call overhead can be measured for common cases.

4.2 Memory Management

Java is a language which heavily relies on dynamic memory allocation and includes a garbage collector. Historically, safety-related applications have used little or no dynamic memory management (except for executions stacks which are well understood and easy to manage). Before going into details for Java, here are the possible levels of dynamic memory management:

- Using no dynamic memory allocation at all (or only during initialization).
  From a safety perspective, this is clearly the easiest way. But it severely restricts the application.

[1] Note that the implementation described in [HRTGC] 10.3.1.3 is NOT implemented in the current version of Jamaica. A linear search is performed, i.e., for an instance of a class that implements n interfaces, there are at most n comparisons to find the method table that corresponds to this interface. This n is typically small (1 or 2) and easy to determine for every class.

- Specialized memory management by the application.
  Most non-trivial applications need at least some dynamic memory management. Often, memory is pre-allocated in pools of objects of the same size and the application manages these pools explicitly (Object Pooling). This avoids the general problem of heap fragmentation; but when a pool exists for each object size, with dynamically sized objects like strings, this is impractical. But, it requires many object pools to be sized correctly. In fact, there is also fragmentation because free memory in one pool cannot be used by allocations for another pool. It is also susceptible to a program releasing an object too soon or not at all, which can result in object corruption or memory exhaustion respectively.
- Heap management by the application.
  Allocations and deallocations from the heap are performed under control of the application. While allocation is relatively easy to manage, it is often quite hard for applications to decide when a piece of memory can be deallocated. Errors in the deallocation are frequent and hard to find. The consequences of error in deallocation are either memory leaks or references to memory which is already used for a different object, with hard-to-predict consequences.
  Heap fragmentation is still an issue resulting in varying time required for allocation and deallocation.
- Implicit heap management by the runtime environment (garbage collection).
  Allocations are either explicit by the application or implicit through the use of certain language constructs. Deallocations are always implicitly performed by the garbage collector. The big advantage is that the application is freed from the burden of deciding when to deallocate an object[2]. The garbage collector is part of the runtime environment; it must be verified only once and can be used for many applications. Typically, the garbage collector also implements a solution for heap fragmentation.

The following issues arise with dynamic memory allocation:

- Time required to allocate / free memory:
  - With Object Pooling, this is generally not an issue; the operations can easily be performed in constant time.
  - When using a common heap, whether managed by the application or the garbage collector, this is generally not the case. The allocation strategy determines the complexity of the operations.
  - When using a garbage collector, the time required for garbage collection is non-deterministic. First, it reduces the execution time remaining for the application. Second, there is typically some need for synchronization between the garbage collector and application code, which introduces a blocking time. This increases the latency for reactions to external events by the application.
- Out-of-memory conditions:
  Allocations can fail if no more memory is available. Application designers must

[2] Sometimes, applications can help the garbage collector by explicitly overwriting object references with NULL, so the garbage collector can deallocate the object. But such actions are easier to verify than the actual deallocation. In particular, a runtime error would be detected should the application erroneously try to use such a NULL reference.

estimate the memory usage of the application and design the application to react to failures of memory allocation.

- Fragmentation:
  Fragmentation describes the situation where unused memory cannot be used to satisfy application requests. It occurs when many small pieces of memory are free but none of them is large enough to satisfy a request. This form typically occurs when using a common heap.
  With object pooling, there is no fragmentation within a pool. But there is a similar problem when one pool is exhausted and memory from other pools cannot be used. With many pools, it becomes difficult to size each pool properly without wasting too much memory.

- Premature Release:
  No object for which a reference exists[3] may be released to the free list. This is part of the job of the garbage collector, to release only objects that are no longer referenced. For object pooling, this must be demonstrated for each application because returning objects to the pool is the job of the application.

- Memory Leak:
  A memory leak happens when the last reference to an object is dropped but not freed. In a garbage collected system, this is a condition under which an object is eligible for being collected, but for other techniques such as pooling this will eventually cause the program to run out of memory. This should not be confused with object hoarding, which is similar: an object is not lost, but a reference is maintained after it is no longer needed.

Java was designed for using an automatic garbage collector: there is no explicit use of pointers, each "new" operation allocates memory and there is no explicit deallocation feature. This makes it possible to implement an exact garbage collector, which can give guarantees to detect free memory. For languages like C and C++, garbage collectors cannot be exact due to language properties. An exact garbage collector must know precisely where all pointers are, both in the heap and in the stack. Since a C or C++ program can morph any integer into a pointer, add any constant to a pointer, and run off the end of arrays, there is no way to know where all references are for certain. For those languages, applications typically perform the deallocations, which is error-prone. As outlined in DO-332, OO.D.1.6.3 and OO.D.2.4.2.2.2, an exact automatic garbage collector addresses most of the issues with dynamic memory management.

The fragmentation and timing issues are still present for Java in general.
Section 6.4 and [HRTGC] show how these are addressed in the JamaicaVM implementation.

4.3 Multithreading

Many applications inherently need support for multiple threads. Even if not strictly necessary, multithreading often allows for simpler implementations. Essentially, application designers have three choices to implement multithreading.

- Create an application-specific scheduler.
  This enables tailoring to the specific needs of an application, but the implementation

[3] More precisely, we can exclude references for which it is known they will not be used to access the object.

effort makes it only suitable for very simple systems. Even then, there is a portability issue.

- Use a commercial-of-the-shelf (COTS) real-time operating system (RTOS), using the RTOS specific API in the application.
  This provides a full-featured RTOS with less implementation effort for the application; the verification of the RTOS is typically performed by the RTOS provider.
  As the RTOS and the programming language used by the application are defined separately, they are not tightly integrated. The application is typically restricted to a procedural interface for synchronization. Higher level concepts like monitors must be created explicitly at the application level.

- Use a programming language with built-in support for multithreading.
  This reduces the implementation effort for the application and improves portability; the verification of the runtime environment is typically performed by the provider of the runtime environment.
  This approach also provides seamless integration of the programming language and the multithreading concept. For example, the interaction between multithreading, memory management, and exception handling are precisely defined within the programming language. As these facilities are implemented together, there are more opportunities for optimizations. The language must then provide the scheduling paradigm needed by the application.
  Real-time Java and Ada, among others provide such an integrated multithreading concept for time-critical systems.

Conventional Java has several issues with real-time applications, e.g., undefined scheduling and synchronization behavior. For this reason, the discussion here focuses on Java with the extensions and semantic refinements of the Real-Time and Embedded Specification for Java, see [RTSJ]. With these concepts applied, real-time Java provides for priority preemptive scheduling with priority inversion avoidance.

When a safety-critical application to be verified is multithreaded, the timing characteristics play an important role. Key questions, besides the timing of the application code itself, must be answered.

- What is the execution time for context switches?
- What is the time between an event triggering a context switch and the actual occurrence of the context switch? This affects the reactivity of the application and is influenced by blocking times introduced by the application itself and the runtime environment.
  Blocking through the garbage collector requires particular attention. (See section 7.1.4).

Note: Multi-processor / multi-core systems are not considered in this paper.

4.4 Exception handling

Exceptions can be thrown implicitly or explicitly. Exception paths need special attention for coverage analysis and worst case execution time (WCET) calculations. The time required to locate an exception handler might include a search process that depends on the stack depth. This is specific to language and implementation.

WCET with exceptions might be much longer than for the non-exception path. If callers always assume the exception path is taken, this might lead to very bad WCET calculations.

When an exception is handled locally, the caller might not even be aware that it occurred, so the caller may incorrectly assume that the non-exception WCET applies.

The strategies for exception handling and WCET calculations must match. Defining these concepts is beyond the scope of this paper.

4.5 Class initialization

See [HRTGC] 10.3.1.2. Static initializers are executed upon first use of a class, which might be while the application is already in operational mode. Applications should devise a scheme for initialization. For example, classes can be explicitly initialized with the Class.forName method.

4.6 Dynamic Loading

Loading of classes is performed in two different contexts:

- Loading the code for a pre-configured application requires making sure that only approved configurations are loaded. This can be achieved by restricting the possible sources of code, e.g. to a single image created by a builder tool.
- Loading additional classes from dynamically computed load paths, possibly over the internet. This is a feature which is quite specific for Java.

The latter is called ‘Dynamic Loading’; it essentially creates a different software configuration.

The source of the dynamically loaded code must be trusted. Any new configuration must be verified. In other words, the system must have a means of verifying a configuration before actually loading. In particular, the new configuration might have different timing characteristics; the WCET for a method might be increased by loading an additional subclass.

The transition between configurations makes the system vulnerable. Loading should be restricted to certain application states or to low priority threads. Loading is a potentially slow operation with potentially long blocking times.

Note: Dynamic loading will not be considered in detail in this paper.

4.7 Interpretation versus Compilation

In the simplest environment, a Java compiler converts Java source code into Java Bytecode which then is interpreted by the Java Virtual Machine (VM). Java class libraries can also be distributed directly as Bytecode. As the Bytecode is standardized, any Java compiler can be used with any VM.

Many Java development environments also provide a means of compiling Bytecode into native code for improved speed. This capability comes in two flavors:

- Just in time (JIT) compilation:
  Based on execution statistics, the VM decides to compile part of the code while the application is running. This approach is not suitable for safety-critical applications for several reasons:
  - The need to verify the behavior of JIT compiler.
  - The inability to verify the resulting object code.
  - The effects of the compilation to the real-time behavior.

- Static compilation:
  When the application is built, selected parts of the code are compiled. This preferable in a verification context:
  - The compiler need not be verified because the resulting object code can be verified
  - The real-time performance is highly predictable

Hence, in the remainder of this paper, we disregard JIT compilation.

Compiled code has typically a higher memory demand, with a factor between 5 and 10. Therefore, only the most time critical methods are usually compiled, striving for a good balance of execution speed and code size.

	Speed	Code size
Bytecode:	slow	small
Compiled code:	fast	large

The VM is responsible for actions supporting the synchronization and garbage collection approach. While it is relatively easy to verify the VM to perform these actions, this is not as easy with the compiled code. This will be discussed in more detail in section 7.5.

5 Requirements for Real-time Java Implementations

For using Java in real-time applications, an implementation must address two issues.

- Provide precisely define real-time scheduling and synchronization algorithms, thus avoiding the pitfalls of the original Java specification. Two approaches are available:
  - use the Safety-Critical Java subset
    accepting severe restrictions on the language (stack-based scoped memory instead of heap memory, restrictions on thread use), or
  - use the Realtime and Embedded Specification for Java ([RTSJ])
    enabling full use of Java and even extensions to its capabilities.
- Avoid indeterminate blocking times due to the garbage collector. Simple Java implementations stop the whole application for a complete garbage collection cycle. More sophisticated Java implementations execute the garbage incrementally. Essentially, there are two mechanisms:
  - Run the GC in a separate thread with a scheduling scheme that given the GC enough execution time but does not hinder the application. The GC thread scheduling parameters have to be adapted for the application.
  - Work-based garbage collectors perform garbage collection in incremental steps. Each time the application allocates memory, the application thread also executes a specific amount of garbage collection work. The more allocations, the more GC work is done.

The following table gives an overview of the possible combinations and their properties regarding the above criteria:

Java Type		Garbage Collection approach		
Name	Short Description	Stop the world	Incremental, Parallel Threads	Incremental, Work based
Conventional Java	OpenJDK like	Undefined Scheduling Unacceptable blocking	Undefined Scheduling	Undefined Scheduling
Safety-Critical Java	Restricted Java without GC	Reduced functionality GC not provided		
Realtime Java	Java with RTSJ extensions & Deterministic GC	Unacceptable blocking	Full functionality GC threads to be configured	Full functionality GC well distributed, easy to configure

6 Using Real-time Java with ISO 26262 and DO-332

The ISO 26262-6:2011 standard, section 5.4.6, gives examples of programming languages which might be used, including Java:

> *The selected programming language (such as Ada, C, C++, Java, Assembler or a graphical modelling language) supports the topics given in ...*

Three of these languages are object-oriented. Therefore; the intent is clearly that such OO languages, and in particular Java, can be used.

The standard does not go into details for certification using OO languages or Java in particular. Requirements/recommendations which have particular implications when using OO languages are listed below.

- ISO 26262-6:2011, 5.4.6 and 7.4.3, call for abstraction, modularity, encapsulation and runtime error handling which are well supported by Java. With the real-time specification, support for embedded real-time software is also covered.
- ISO 26262-6:2011, 7.4.17, calls for upper bounds for execution time and storage space. While this is not specific for OO languages, it is of particular importance with dynamic binding and garbage collection.
  This requirement maps to DO-332 objective OO.6.3.4f.
- ISO 26262-6:2011, 8.4.4, calls for limited use of pointers in Level D applications. Java does not use pointers explicitly whereas C and C++ do.
- ISO 26262-6:2011, 10.4.6, requires metrics for the call coverage but does not explicitly mention dynamic binding. The objective includes demonstrating the absence of unintended functionality, however. This corresponds to structural coverage objectives of DO-178C, 6.4.4.2.
- For configurable software, ISO 26262-6:2011, 5.4.3, requires Annex C to be applied. This is of particular importance when dynamic loading is used.

As the guidance for certification of object-oriented code given in ISO 26262-6:2011 is quite vague, it is reasonable to compare ISO 26262-6:2011 to DO-178C and DO-332 and then look for additional guidance in DO-332, since DO-332 provides more detailed guidance for using object-oriented technology.

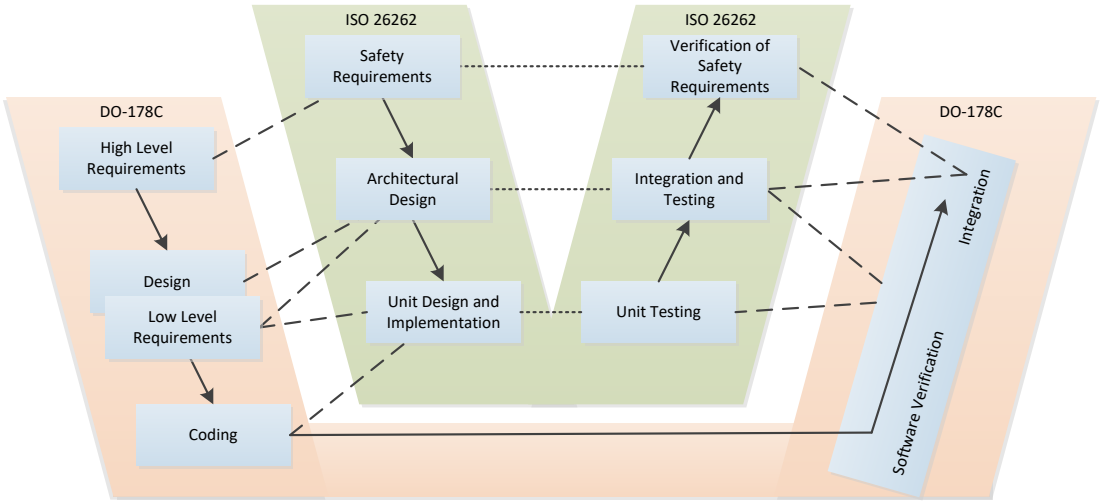
6.1 Comparison of ISO 26262-6:2011 to DO-178C

The safety certification standard for avionics, RTCA/DO-178C, provides details for certification of object oriented programming that are missing in ISO-26262. There is an entire supplement to DO-178C, called RTCA/DO-332, which covers certification of object-oriented technology and related techniques. Since DO-178C has a similar general approach to certification as ISO-26262, it is reasonable to use DO-332 as a reference. Here is a quick overview of the framework from ISO 26262-6:2011 and DO-178C with DO-332 in a tabular form:

ISO 26262	DO-178C + DO-332
Targets automotive sector High-level requirements are derived from the vehicle	Targets avionic software High-level requirements are derived from the aircraft
ISO 26262-6:2011 covers S/W	DO-178C covers S/W DO-332 is for Object Oriented S/W
Both focus on integrated safety through a defined development process	
Recommends actions Objectives are more implicit	Defines objectives and leaves it to the applicant to plan actions to achieve the objectives.
Objectives are comparable Based on requirements and their verification / test	
Functional safety manager role	Certification authority (FAA / EASA)
Safety assessment based on artifacts produced during product development	
Applies to system, hardware, and software	Software only (system and hardware are handled in other standards)
Applies to production & operation of the product	Production out of scope (Covered by other standards)
Defines life cycle phases based on V-Model	Development process not specified. Suggests waterfall model, can be matched to V-Model.
Re-use of components based on field experience	Artifacts for previously developed software must comply to standards
Both start with requirements that are refined into a design and then an implementation. Both are based on testing at various integration levels.	

The following diagram compares development activities for ISO 26262-6:2011 and DO-178C:

Figure 6-1 V-model Verification Diagram



As illustrated by Figure 6-1, the processes are comparable, although DO-178C allows for more flexibility.

The similarities between both frameworks are strong enough to justify the use of DO-332 as additional guidance. DO-332 covers certification of object-oriented code (albeit for use in avionic systems) in much more detail than ISO 26262-6:2011. Not all of the objectives given in DO-332 are necessarily required for verification according ISO 26262-6:2011. But applying the more detailed framework of DO-332 seems to cover the ISO 26262-6:2011 objectives. On the other hand, with ISO 26262-6:2011 being quite vague, none of the DO-332 objectives can easily be dismissed, at least for ASIL D.

6.2 Objectives and Activities for Object Oriented Software

The following condensed table is based on the tables given in DO-332, annex OO.C. It contains a subset of the objectives and activities which were changed in DO-332 with respect to DO-178C, i.e., which are OO specific. Furthermore, the more general objectives for Planning, Quality Assurance, and Certification Liaison Process are omitted. The middle column in the table provides a short summary of the OO-specific activities that apply to DO-332. References in the last two columns are to sections below the table.

#	DO-178C / DO-332 Objectives	DO-178C / DO-332 Activities	Short summary of OO specific Activities	Remarks for application & Java libraries	Remarks for Java VM and runtime environment, including garbage collector
1	**Software Planning Process**				
2	**Software Development Process**				
2-1	High-level requirements are developed. [5.1.1.a]	5.1.2 (a,b,c,d,e,f,g,j) OO.5.5.a	Requirements to class methods should also trace to overriding definitions in subclasses.	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A
2-3	Software architecture is developed. [5.2.1.a]	OO.5.2.2 (a,d,h,i,j,k,l)	**h**: Class hierarchy **i**: locally type consistent **j**: strategy for memory management **k:** strategy for exception management **l**: reusing components: derived requirements & functionality to deactivate	**h, i**: See 6.3 Class Hierarchy, Tracing and Type Consistency **j**: See 6.4 Strategy for memory management **k**: See 6.5 Overloading and type conversion vulnerabilities **l**: See 6.7 Reusing components	**j**: See 6.4 Strategy for memory management **k**: See 6.5 Overloading and type conversion vulnerabilities others N/A
2-4	Low-level requirements are developed. [5.2.1.a]	OO.5.2.2 (a,e,f,g,i) 5.2.3 (a,b) 5.2.4 (a,b,c) OO.5.5.(b,d)	OO.5.2.2 I: locally type consistent OO.5.5.d: traces to overriding method declarations in subclasses	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A

#	DO-178C / DO-332 Objectives	DO-178C / DO-332 Activities	Short summary of OO specific Activities	Remarks for application & Java libraries	Remarks for Java VM and runtime environment, including garbage collector
2-5	Derived low-level requirements are defined and provided to the system processes, including the system safety assessment process. [5.2.1.b]	OO.5.2.2 (b,c, l)	I: reusing components: derived requirements & functionality to deactivate		N/A
2-6	Source Code is developed. [5.3.1a]	5.3.2 (a,b,c,d) OO.5.5.c	Trace data include traces to overriding method declarations in subclasses	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A
3	**Verification of Outputs of Software Requirements Process**		OO.11.14, Verification results include: Local type consistency Dynamic memory management		
4	**Verification of Outputs of Software Design Process**		OO.11.14, Verification results include: Local type consistency Dynamic memory management		
4-8	Software architecture is compatible with high-level requirements. [OO.6.3.3.a]	OO.6.3.3	Architecture does not conflict with HLRs: exception management and memory management. Trace data include traces to overriding method declarations in subclasses	See 4.4Exception handling See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A
4-9	Software architecture is consistent. [OO.6.3.2.b]	OO.6.3.3	Trace data include traces to overriding method declarations in subclasses	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A
4-10	Software architecture is compatible with target computer. [OO.6.3.3.c]	OO.6.3.3	Trace data include traces to overriding method declarations in subclasses	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A
4-11	Software architecture is verifiable. [OO.6.3.3.d]	OO.6.3.3	Trace data include traces to overriding method declarations in subclasses	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A

#	DO-178C / DO-332 Objectives	DO-178C / DO-332 Activities	Short summary of OO specific Activities	Remarks for application & Java libraries	Remarks for Java VM and runtime environment, including garbage collector
4-12	Software architecture conforms to standards. [OO.6.3.3.e]	OO.6.3.3	Trace data include traces to overriding method declarations in subclasses	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A
4-13	Software partitioning integrity is confirmed. [OO.6.3.3.f]	OO.6.3.3	Trace data include traces to overriding method declarations in subclasses	See 6.3 Class Hierarchy, Tracing and Type Consistency	N/A
5	**Verification of Outputs of Coding & Integration Process**		OO.11.14, Verification results include: Local type consistency Dynamic memory management		
5-1	Source Code complies with low-level requirements. [OO.6.3.4.a]	OO.6.3.4	Overloading and type conversion vulnerabilities, OO.D.1.3.1 & OO.D.1.4.1	See 6.5 Overloading and type conversion vulnerabilities	N/A
5-2	Source Code complies with software architecture. [OO.6.3.4.b]	OO.6.3.4	Overloading and type conversion vulnerabilities, OO.D.1.3.1 & OO.D.1.4.1	See 6.5 Overloading and type conversion vulnerabilities	N/A
5-3	Source Code is verifiable. [OO.6.3.4.c]	OO.6.3.4	Overloading and type conversion vulnerabilities, OO.D.1.3.1 & OO.D.1.4.1	See 6.5 Overloading and type conversion vulnerabilities	N/A
5-4	Source Code conforms to standards. [OO.6.3.4.d]	OO.6.3.4	Overloading and type conversion vulnerabilities, OO.D.1.3.1 & OO.D.1.4.1	See 6.5 Overloading and type conversion vulnerabilities	N/A
5-5	Source Code is traceable to low-level requirements. [OO.6.3.4.e]	OO.6.3.4	Overloading and type conversion vulnerabilities, OO.D.1.3.1 & OO.D.1.4.1	See 6.5 Overloading and type conversion vulnerabilities	N/A
5-6	Source Code is accurate and consistent. [OO.6.3.4.f]	OO.6.3.4	Overloading and type conversion vulnerabilities, OO.D.1.3.1 & OO.D.1.4.1	See 6.5 Overloading and type conversion vulnerabilities	N/A

#	DO-178C / DO-332 Objectives	DO-178C / DO-332 Activities	Short summary of OO specific Activities	Remarks for application & Java libraries	Remarks for Java VM and runtime environment, including garbage collector
6	**Testing of Outputs of Integration Process**		OO.11.14, Verification results include: Local type consistency Dynamic memory management OO.11.21, Trace data include traces to overriding method declarations in subclasses		
6-1	Executable Object Code complies with high-level requirements. [6.4.a]	6.4.2 OO.6.4.2.1 6.4.3 6.5	Constructors to properly initialize object state. Initial state consistent with class requirements	See 6.3 Class Hierarchy, Tracing and Type Consistency	
6-3	Executable Object Code complies with low-level requirements. [6.4.c]	6.4.2 OO.6.4.2.1 6.4.3 6.5	OO.11.14, Verification results include: Local type consistency Dynamic memory management	See 6.3 Class Hierarchy, Tracing and Type Consistency	See 7.4.1 Verification of the Garbage Collector
7	**Verification of Verification Process Results**		OO.11.14, Verification results include: Local type consistency Dynamic memory management	See 6.3 Class Hierarchy, Tracing and Type Consistency	
7-OO 10	Verify local type consistency [OO.6.7.1]	OO.6.7.2		See 6.3 Class Hierarchy, Tracing and Type Consistency	

#	DO-178C / DO-332 Objectives	DO-178C / DO-332 Activities	Short summary of OO specific Activities	Remarks for application & Java libraries	Remarks for Java VM and runtime environment, including garbage collector
7-OO 11	Verify use of dynamic memory management is robust [OO.6.8.1]	OO.6.8.2. (a,b,c,d,e,f,g)	**a**: exclusivity **b**: allocations succeed when free memory available **c**: memory reclaimed before needed **d**: sufficient memory at any time **e**: reference consistency **f**: atomic object moves **g**: bounded time operations	**d, g**: See 6.4 Strategy for memory management most items are addressed jointly by application / memory configuration / runtime & compiler options and the runtime environment	**a, b, c, e:** See 6.4 Strategy for memory management F: N/A See 7.4.1 Verification of the Garbage Collector
8	**Software Configuration Management Process**				
9	**Software Quality Assurance Process**				
10	**Certification Liaison Process**				

6.3 Class Hierarchy, Tracing and Type Consistency

The class hierarchy must be developed based on the high-level requirements.

Class constructors must properly initialize objects; the initial state must be consistent with the requirements for the class.

Low-level requirements tracing to class members must also trace to the overriding members in all subclasses. For example, given a low-level requirement 1 that applies to method foo in class A and class B is a subclass of A which overrides foo, requirement 1 applies to both A.foo and B.foo. Both A.foo and B.foo must refer back to requirement 1, though B.foo may have additional requirements that do not apply to A.foo.

The testing strategy must verify local type consistency. This can be done by applying tests for a superclass to all of its subclasses. Formal methods can also be used to show that overridden methods conform to the requirements of all methods overridden.

This is a responsibility for the application and the class libraries.

See DO-332 OO.1.6.1.2, OO.D.1.1.3.

6.4 Strategy for memory management

With Java, most of the dynamic memory management aspects are covered by the runtime environment, which provides an exact garbage collector. Most decisions regarding dynamic memory management are implied by selecting Java as programming language and by selecting a specific Java implementation. In DO-332, this dynamic allocation with exact garbage collection for recycling memory is referred to as *automatic heap allocation*.

The verification results should describe how the chosen Java implementation supports the overall goals of the application and how it matches the high-level requirements. This applies in particular to the real-time properties of the garbage collector and how it affects the timing of the application.

Application designers must determine the maximum amount of memory used by the application; the heap size must be set to a value such that the used memory never exceeds a selected fraction of the heap size. That fraction determines the allocation overhead for garbage collection for that application. In other words, an application which requires more memory must have a larger absolute overhead to maintain the same fractional overhead.

These items from DO-332 OO.6.8.2 are then addressed as follows:

- **a**: exclusivity
  e: reference consistency
  These items are granted by the GC approach. Memory is only reused after the GC determined there are no references to the piece of memory to be reused.
- **b**: allocations succeed when free memory is available
  In general, the Java implementation is responsible for avoiding heap fragmentation, e.g. by moving objects.
  For JamaicaVM , this item is granted by the distributed object model which avoids fragmentation issues at the time of allocation.
- **c**: memory reclaimed before needed
  This is granted by the GC approach *provided* the application does not use more than the selected fraction of the heap size.

- **d**: sufficient memory at any time
  The application must not use more than designed amount of memory. Typically, the heap size is set to a larger value and the application must not use more than a selected fraction of the heap size.
- **f**: atomic object moves:
  The GC is responsible for moving objects.
  For JamaicaVM , this is not applicable because objects are never moved with JamaicaVM.
- **g**: bounded time operations:
  This would be a problem for non-real-time Java implementation, due to the unpredictable timing of the garbage collector.
  For real-time Java, allocation itself is a constant time operation. The amount of time spent with garbage collection (GC overhead) is determined by the GC parameters.

The situation can also be described in a table, similar to the one from OO.D.1.6.3:

Technique	Activities (DO-332 OO.6.8.2)						
	a	b	c	d	e	f	g
Object pooling	X	X	X	X	X	N/A	OK
Stack allocation	X	OK	OK	X	X	N/A	OK
Scope allocation	OK	OK	OK	X	X	OK	OK
Manual heap allocation	X	X*	X	X	X	N/A	OK
Automatic heap allocation	**OK**	**OK**	**OK**	**X**	**OK**	**OK**	**OK**
RT Java with JamaicaVM	**OK**	**OK**	**OK**	**X**	**OK**	**N/A**	**OK**

With the entries meaning:

X = to be addressed by the application

OK = addressed by memory management infrastructure

N/A = not applicable,

* = difficult to ensure by either application or memory management infrastructure

Clearly, the automatic heap allocation with garbage collection puts the least burden on the application side.

Note that with JamaicaVM, fragmentation and atomic moves are not an issue, due to non-contiguous object allocation. Furthermore, the work based GC algorithm shows graceful

degradation: if the application uses slightly more than the designed heap fraction, the time used for incremental garbage collection is slightly increased.

See DO-332 OO.D.1.6 for more details.

Java also uses stack allocation, but JamaicaVM only uses this for local variables. The sizes of runtime stacks for each thread have to be determined as for any other programming language. Stack overflow is detected automatically by the JamaicaVM runtime causing an exception to be thrown.

6.5 Overloading and type conversion vulnerabilities

Overloading ambiguity occurs when implicit type conversions occur when selecting an acceptable match. This could lead to unintended best match selection by the compiler. For Java, boxing and unboxing conversions as well as the use of variable arity methods have to be considered.

Different programmers might unintentionally choose the same name for semantically different operations.

Coding standards should address the issues listed in DO-332 OO.D.1.3.3.

This is a responsibility for the application and the class libraries.

See DO-332 OO.D.1.3.1, OO.D.1.4.1, OO.D.1.3.3.

6.6 Strategy for exception management

The mechanisms to throw, propagate and catch exceptions are defined by the Java programming language.

The verification results should describe how the chosen Java implementation supports the overall goals of the application and how it matches the high-level requirements.

The coding guidelines should describe in which situations exceptions should be used and in particular, where exception handlers should be placed (and where not). This is important for the timing behavior of the application. Exceptions are part of the class requirements and must be included in the verification of type consistency.

There is a particular issue with exception handling regarding timing that can be illustrated by an example:

> If an application works with a container which might become empty, it can test whether the container is empty *before* trying to retrieve an element. As an alternative, it can just try retrieving an element and catch the exception thrown in case the container was empty.

Functionally, both schemes work fine, but the timing might be quite different:

- The test for an empty container is usually an operation with a small and easy to determine execution time.
- Throwing and catching an exception can be much slower and the execution time is harder to predict. The execution time might even be influenced by non-local properties.

  With JamaicaVM, the overhead of throwing an exception is determined by creation of the exception object and the search for the corresponding handler. Unless preallocated exception objects are used, creation of an exception requires allocation for the exception object itself, and of the stack trace that is stored in the exception and

depends on the call depth when the exception is created. The search for an exception handler is determined by the number of the stack frames to be removed until the exception handler is found, see [HRTGC] 10.3.3.

With the exception approach, the average execution time for accessing an element might be smaller; the worst case execution time is likely to be worse, however. In Java, it is not possible to remove all exception handling, but one must understand their effect on WCET. In general, exceptions should be used sparingly, where code readability considerations overweigh the extra time required for generating an exception in the case of failure.

6.7 Reusing components

Reuse of components typically applies to class libraries, be it libraries defined in the application domain or those belonging to the Java runtime environment.

Only Java libraries which come with verification artifacts should be considered as components; in other words, libraries where requirements and a complete set of corresponding tests are available. Other libraries have to be treated like parts of the application.

The following issues should be addressed, for more details, see DO-332 OO.D.2.3.1.

a. Components developed outside the system context might not fully satisfy the intended functionality or they may provide more functionality than needed.

b. Life cycle data may be developed to different standards.

c. Error management for components might follow a different policy than required by the system. Wrappers might be needed. See also section 6.6.

d. Resource management might be different (heap, stack, processor cycles, and synchronization).

- Memory management:
  As Java includes a complete, exact garbage collector, stack and heap management are addressed within the runtime environment. The stack and heap usage by the component must nevertheless be documented.

- Execution time:
  Unless the component was verified in the Java environment, timing information is likely to be missing or to be inadequate, due to the work-based GC approach. Even if provided, execution lime limits for the component might be specified in a form which is incompatible with the approach selected for the application.

e. Presence of deactivated code: Parts of the reused component might not be needed in the given system.

f. Integration testing, including data coupling analysis and control coupling[4] analysis may be difficult without internal knowledge of the component.

The reused component may be available as source code, Java Bytecode or as object code. If source code is not available, this makes the analysis more complicated. This affects in particular item f above; the timing analysis is also affected.

[4] ISO 26262-6:2011 does not use the term “control coupling”, but it requires metrics for the call coverage. Full call coverage implies that all possible calls have been verified, which means control coupling was verified.

Though integration testing is always required, other retesting can only be avoided where the same object code or the same bytecode and interpreter are used. For static compilation, this means compiling the same set of methods with the same settings and interpreting the remaining set. WCET is dependent not only on the hardware, but also what code is compiled and what code is interpreted. Technically, it is possible to extend the class hierarchy by introducing application-specific subclasses of classes defined within reused components. First, this requires a good understanding of the component, including access to source code and documentation. Second, it requires special attention when verifying the class hierarchy and type consistency as described in section 6.3.

7 Verification with JamaicaVM

This section describes what's to be done when using JamaicaVM, taking into account the specific implementation details.

Note that this section does NOT list all those activities which are required regardless of the use of Java. We assume the reader to be aware of the verification activities applicable for non-object-oriented implementations, e.g. using the C programming language.

7.1 Verifying JamaicaVM Applications

Coding guidelines must be established, in particular covering the aspects from sections 6.5 and 6.6.

Verify the class hierarchy and type consistency as described in section 6.3.

For compiled parts of the application, the issues described in section 7.5 apply.

7.1.1 Storage Requirements

Establish a scheme to determine boundaries for storage requirements. Define the required heap size and the minimum amount of unused heap memory.
The JamaicaVM environment provides tools for measuring the memory usage.

Consider monitoring the storage used and define the behavior of the application in case the storage limit should be exceeded.

7.1.2 Execution Time Limits

Establish a scheme to determine boundaries for execution time:

- Will the execution time be calculated or measured?
- Will it be determined for the application as a whole, per subsystem or at a method level?
- How will the variability of execution times for overridden member methods in a class hierarchy be accounted for?
  Worst case execution times can be considered as post-conditions. The LSP implies that subclasses cannot exceed the timing limits defined for their superclass. This might not apply strictly for all class hierarchies and using this approach implies the WCET is specified on the function level.

Decide whether the GC overhead will be configured statically or adjusted dynamically.

Make sure any time measurements are done in a configuration with worst case garbage collection overhead; even if the application is going to dynamically adjust the GC overhead to the heap usage, measurements should be made with a static setting corresponding to the situation with maximum expected heap usage.

JamaicaVM supports two array layouts: linear (contiguous) and tree arrays. For arrays that are not pre-allocated, time measurements should be made with the optimization for contiguous array allocation switched off. This optimization depends on being able to allocate an array in a contiguous set of -blocks in memory, which cannot be guaranteed, in particular after the system has been running a long time and memory is nearly exhausted. See [HRTGC] 7.3.3.

7.1.3 Exception handling scheme

Establish a scheme for exception handling, including the consequences of exceptions regarding execution time, as discussed in section 6.6.

The scheme to be used must be compatible with existing class libraries to be used. So, in practice, there may be little choice.

Even if rarely expected in safety-critical Java code, specify whether the code is expected to be executable with runtime checks suppressed.

7.1.4 Guaranteed response times

Establish a scheme to guarantee required response times. Select the compiler and runtime options to match the expected response times.

A real-time Java implementation must ensure that the garbage collector does not interfere with the running program and that changes by one are seen immediately by the other. This can be done either with locks and read barriers or with well-defined synchronization points. In general, using synchronization points is more efficient for a very small latency penalty.

JamaicaVM restricts garbage collection and scheduling actions to so-called synchronization points. At these synchronization points, the heap must be in a consistent state, allowing for incremental garbage collection by other threads. Between synchronization points, there is no need to guarantee the GC invariants and there is more opportunity for optimizations. The distance between synchronization points is such that the time to run from one to the next is much smaller than the time for the OS to switch contexts.

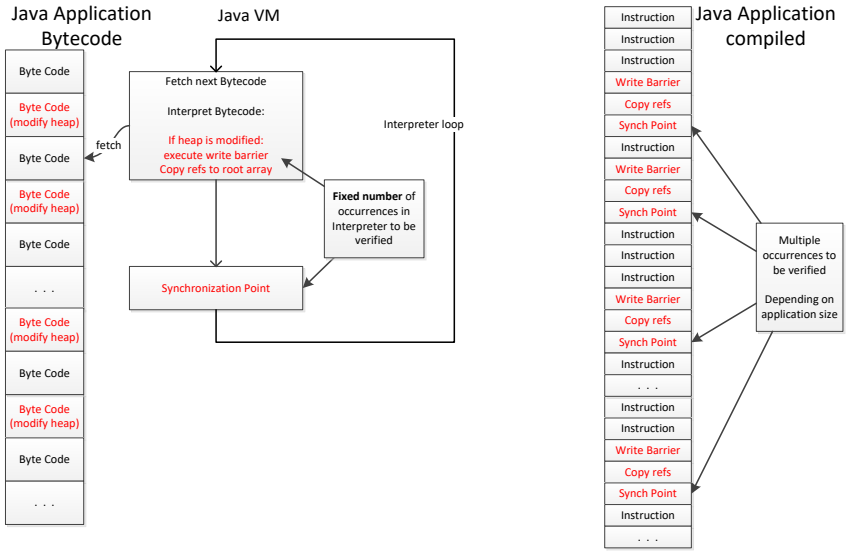
Clearly, the maximum distance between synchronization points is critical for response times:

- The JamaicaVM bytecode interpreter inserts synchronization points at defined time intervals.
- The JamaicaVM garbage collector is interruptible between the processing steps for memory blocks, due to explicitly programmed synchronization points.
- For compiled code, be it application code or libraries, the JamaicaVM compiler inserts synchronization points[5]. See also section 7.5.

Figure 7-1 illustrates the difference between interpreted and compiled code.

[5] The application can select the distance between sync points. If the value is different from the one which was assumed when compiling libraries (and the one implemented in the VM, see 7.4), the larger value determines the responsiveness of the application as a whole. See *jamaicavm_8.0_manual, section 14: Option* ***-threadPreemption=****n*

Figure 7-1 Synchronization points in interpreted versus compiled code



7.1.5 Interoperability between application code and libraries

With JamaicaVM, there are several compiler and runtime options which affect the overall properties of the resulting application. The interaction of these options must be understood.

The application defines the parameters for the garbage collector, in particular the maximum fraction of the heap size used by the application and the amount of GC overhead per allocation. This must be taken into account when determining the execution time of library functions.

If library code is interpreted rather than compiled, changes in runtime options will also affect library code (its execution time, in particular).

7.1.6 Interoperability with Native Code

Java enables the use of native code, written in programming languages like C. This causes a couple of problems regarding memory management, as the native code cannot be expected to cooperate with the garbage collector like the Java code does. Hence, native code must not access any memory which is subject to garbage collection except through functions defined by the Java Native Interface, JNI. Any memory allocated by native code must be separate from the heap used by Java. Furthermore, native code will have different behavior regarding scheduling than Java code: native code will not include explicit synchronization points.

JamaicaVM supports JNI, which does not allow direct access to Java memory; any access has to be performed through calls of JNI routines. With JamaicaVM, native code is executed in parallel to Java threads, with scheduling enabled.

Thus, the execution time of native code does not affect the response time of the application, nor does it interfere with the garbage collector. See [HRTGC] 5.8.2.

7.2 Verifying Third Party Libraries

Libraries which are delivered with verification artifacts according to the applicable standard and the required safety level can be considered as components as described in section 6.7.

Other libraries must be verified together with the application, the previous section applies. The following additional issues should be considered:

- If libraries are delivered without source code, creating verification evidence is quite difficult, if not impossible.
- If source code comes with legal restrictions, it becomes more difficult to remove unused parts (in order to reduce cost) or to correct issues found during verification.
- Missing documentation about the internals of the library makes verification more difficult.
- Verifying third party code is more difficult than verifying your own code.

7.3 Verifying the JamaicaVM Libraries

The term JamaicaVM Libraries is herein used for libraries provided with the JamaicaVM development environment for which aicas is willing to assume responsibility for creating verification artifacts. What this library contains must be defined for each project. Other libraries should be considered as third-party libraries, even if provided with JamaicaVM.

A subset of the available libraries must be selected and verified.

Coding guidelines must be established, in particular covering the aspects from sections 6.5 and 6.6.

Verify the class hierarchy and type consistency as described in section 6.3.

In contrast to third-party libraries, there is a better chance to provide useful timing information which can be integrated into the timing analysis of the application. However, the current approach is to leave timing to the application.

The current approach is to let the user decide which parts of the code are compiled. In case the object code is also delivered with JamaicaVM, the compiler options used to compile libraries must not introduce compatibility problems with applications. It may be possible to provide the binaries for libraries for a limited number of variants, with different compiler option settings and to allow the application developer to select the best-suited variant.

For compiled parts of the libraries, the issues described in section 7.5 apply.

7.3.1 Storage Requirements

Establish a scheme to determine boundaries for storage requirements. For libraries, this must be done in a way such that the application developers can integrate the results into their environment:

- Provide formulas to calculate the storage requirements, or
- Provide information how to measure the maximum storage requirements.

7.3.2 Execution Time Limits

Establish a scheme to determine execution times. For libraries, this must be done in a way such that the application developers can integrate the results into their environment:

- Provide formulas to calculate the execution times, or
- Provide information how to measure the maximum execution times.

In particular, the influence of the GC overhead per allocation needs to be addressed.

7.3.3 Exception handling scheme

Establish a scheme for exception handling, including the consequences of exceptions regarding execution time, as discussed in section 6.6.

For new development, the decisions made here must not unduly restrict the exception handling approach used by applications. Class interfaces should be designed to allow using them without being forced to work with exception handlers for normal operations. Exceptions should only be used for handling situations that represent real errors. See the example given in section 6.6.

For the vast majority of classes delivered with JamaicaVM, there is no option to change the class interface design, but the interface of the Java standard edition class libraries is used exactly as is done by OpenJDK, etc. Hence, application designers must adapt their exception handling strategy accordingly.

7.4 Verifying the JamaicaVM Runtime Environment

The runtime environment, namely the JamaicaVM, consisting of the Java interpreter, the garbage collector, the class loader and the support code, need to be verified as any software included in a safety-critical system. Here, we concentrate on the specific requirements affecting memory management and real-time characteristics:

(1) Insertion of synchronization points.

(2) Insertion of write barriers.

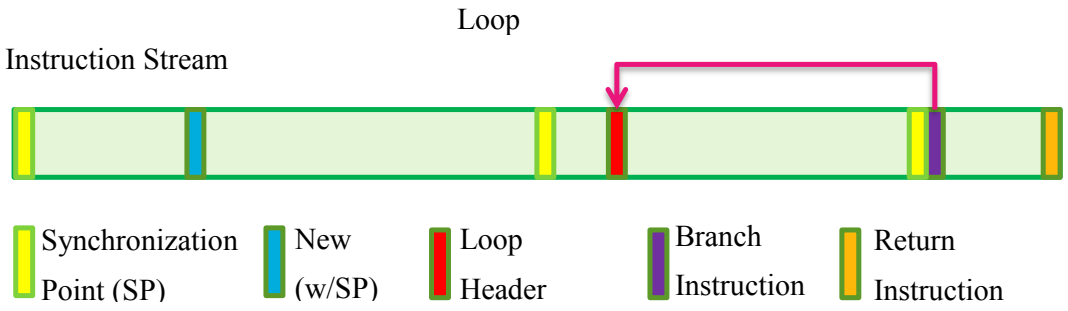
(3) Copying references from stack to root arrays.

In support of point (1), the Java interpreter contains code to execute a synchronization point after executing every single bytecode. All other code in the runtime environment, including the garbage collector, contains synchronization points which make sure the time between executing synchronization points never exceeds the designed limit. Every allocation (new instruction) contains a synchronization point. Figure 7-2 illustrates a method of implementing synchronization points. This limit needs to be precisely documented[6]. The intent is to make the limit small compared to the context switch time. Since context switch time also varies with the CPU speed, the limit in terms of maximum number of instructions can be fixed. In any case, it is not application dependent.

It seems feasible to create a relatively simple tool which partially verifies this property on the object code, pointing out the locations where synchronization points might be missing, or where a manual analysis is necessary. Such a tool could also be used to verify the compiler output, see section 7.5.

[6] The distance between synchronization points within the runtime system, including the interpreter, cannot be changed easily; it is fairly small, but platform dependent. Assuming that all applications will contain code which is interpreted, it does not make much sense to use a different limit for compiled code.

Figure 7-2: Placing Synchronization Points into the compiled code



The VM is responsible for implementing items (2) and (3). No attempt is made for optimizations, as interpretation of Bytecode is relatively slow anyway. Any code in the runtime environment that deals with references into the heap must also be verified to adhere to the protocol.

7.4.1 Verification of the Garbage Collector

The verification of the garbage collector is not much different from other code of the runtime environment.

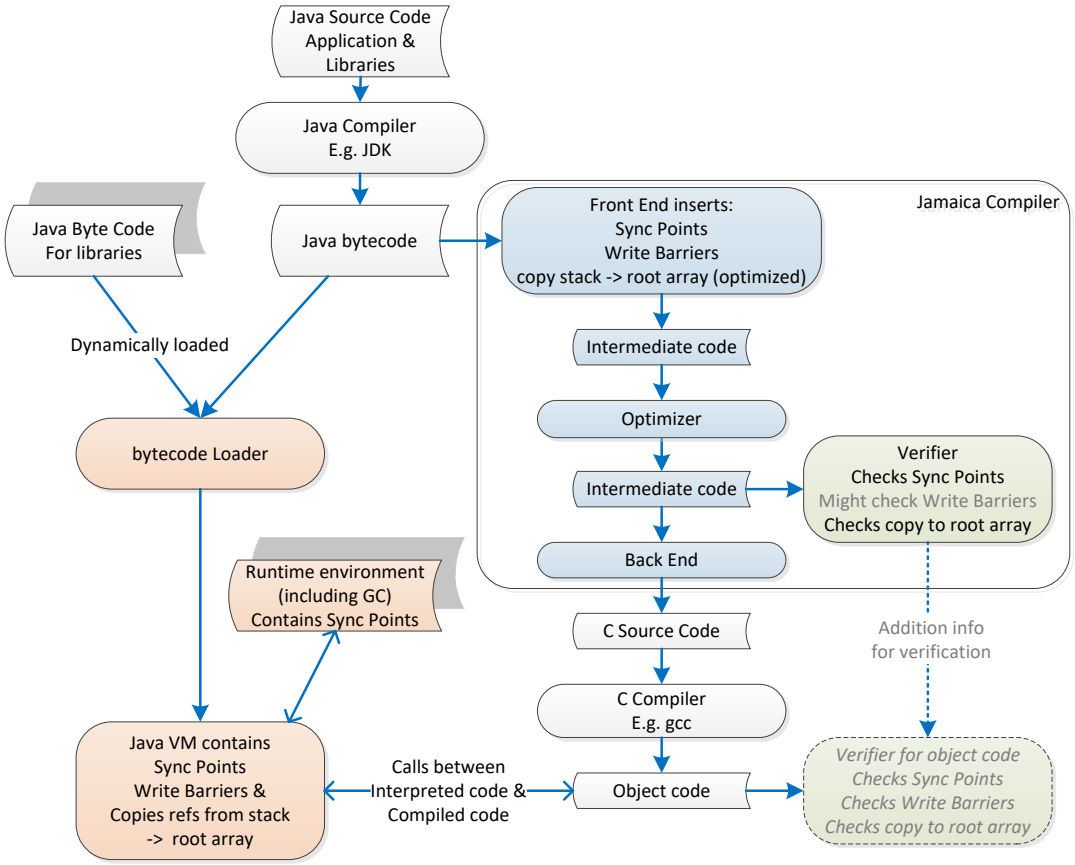
The properties of the garbage collector regarding correctness of the GC algorithm (e.g. GC invariants) become functional requirements. The most interesting aspect seems to be the garbage collector progress. The claims made in [HRTGC] 9.5 need to be transformed into high-level requirements and verified using the material present in [HRTGC], taking into account that the implementation has been optimized since [HRTGC] was written.

7.5 Verification of Compiler Output

When Java Bytecode is compiled rather than interpreted by the Jamaica VM, the compiler takes over the responsibility for requirements (1), (2) and (3) as listed in section 7.4.

Figure 7-3 shows the simplified structure of the JamaicaVM compiler, ignoring the rest of the builder. The white boxes represent third-party tools. The orange part is for purely interpreted Java Bytecode; the blue part is the Java compiler. The green part is about verification; the dashed part does not currently exist, it is just a proposal.

Figure 7-3: Certifying the Output of the Compiler[7]



The compile chain currently works as follows:

- Java source code is compiled into Bytecode with any Java compiler. This step is not aware of the JamaicaVM runtime environment.
- The front end of the JamaicaVM compiler transforms Java Bytecode into Intermediate Code, implementing requirements (1), (2), and (3).
- The intermediate code is optimized, in particular regarding implementation of (3).
- On the intermediate code, several checks are performed regarding (1), (3).
- The back end of the JamaicaVM compiler then transforms intermediate code into C source code.
- The C code is compiled with a COTS C compiler (gcc). The verification of the compiler is not required (and not considered feasible).

Even if the Intermediate Code is fully verified, the last two steps could theoretically introduce errors.

It is an open issue to decide whether the verification performed on the Intermediate Code is sufficient and can be qualified under the requirements of ISO 26262-6:2011. What is the tool confidence level of the compiler chain in general and for the verification of the three requirements discussed here in particular? While many errors in the compiler tool chain can

[7] Dotted lines represent elements still to be done.

be detected by requirements-based testing of the resulting object code, the issues listed below are much harder to test:

- A missing synchronization point would just increase the latency for context switches.
- An error in inserting write barriers or copying references from the stack to root arrays would result in a memory management error, which may (but often will not) lead to deallocation of an object still in use. This is hard to find through testing; if an error is detected, it still remains unclear which piece of code is incorrect.

Essentially, there are two ways to approach the verification of requirements (1), (2), and (3):

- Create a convincing argument for trusting the requirements are still satisfied by the object code.
- Develop a tool that can verify the above issues by analysis of the object code, possibly helped by additional data from the verification on the intermediate code.
  Note that a verification tool for (1) would also help in the verification of the runtime environment.

A compiler which directly compiles Java Bytecode into object code is planned. Ideally, this compiler would include a verification step working on object code.

8 Potential Problems

See [HRTGC] 10 for more issues not addressed by ISO 26262-6:2011 or DO-332.

9 Conclusion

The OO aspects of Java require OO specific verification activities as for any OO language, as expected.

A lot of libraries are available, from third parties as well as delivered with the Java implementation in use. In any case, using these libraries requires planning for their verification and the integration of the verification artifacts into the application's evidence.

The use of a garbage collector makes life relatively easy from an application perspective. The major part of the verification effort for memory management is shifted to the implementation of the Java runtime environment.

When Java is used for real-time applications, a deterministic timing behavior is required, including a defined response time to external events. This is addressed by using:

- The semantic refinements and additional APIs of the Real-time and Embedded Specification for Java ([RTSJ]).
- A real-time garbage collector, providing incremental garbage collection and a defined response time, as included in JamaicaVM.

Verifying the Java bytecode interpreter and the garbage collector is not particularly difficult.

The most difficult part is the verification of the 'global' requirements, which support the GC strategy and the real-time behavior and apply to all Java code. For interpreted code, the interpreter takes over this responsibility; the verification is relatively straight forward. For compiled code, the verification is harder to accomplish, the compiler might have to be adapted. Although section 7.4 is specific for JamaicaVM, other real-time Java environments are expected to have similar requirements.

The following steps need to be addressed

- Verification of the runtime environment, including
    - the Java Bytecode interpreter (virtual machine) and
    - the garbage collector
    - the class loader
- Determine a strategy to verify compiled code regarding the 'global' requirements.
    - Can we trust the compiler?
    - Is a checking tool feasible?
- Define set of Java libraries to be used by applications
    - Who is responsible for their verification?

Further work needs to be done to investigate dynamic class loading and using multicore or multiprocessor platforms.

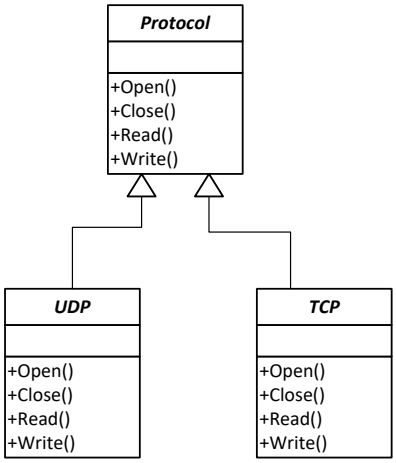
Appendix A Acronyms

API	Application Programmers Interface
ASIL	Automotive Safety Integrity Level
CPU	Central Processing Unit
COTS	commercial of-the-shelf
EASA	European Aviation Safety Agency
FAA	Federal Aviation Administration
GC	garbage collector
HRTGC	Hard Real-time Garbage Collection
JNI	Java Native Interface
JIT	Just in time (compilation)
LSP	Liskov Substitution Principle
OO	Object Oriented / Object Orientation
PDS	Previously Developed Software (a DO-178C term)
RTSJ	Real-time and Embedded Specification for Java
RTOS	Real-time Operating System
SP	Synchronization Point
VM	Virtual Machine
WCET	worst case execution time

Appendix B Example for OO paradigm implemented by switch

In a hierarchy like the one given in Figure B-1, the inheritance mechanism will make sure that each subclass provides all four operations.

Figure B-1 Example Hierarchy



The above could be implemented directly in an OO language like Java.

In an equivalent C implementation, the distinction between protocols will be expressed by a switch construction like the one shown in Figure B-9-2:

Figure B-9-2 Implementation by switch construct in C

```
void open ()
{
   switch (protocol_type)
      {
      case UPD:
         {
         /* handle UDP */
         }

      case TCP:
         {
         /* handle TCP */
         }

      default:
         {
         /* handle errors */
         }
      }
}
```

```
void close ()
{
   switch (protocol_type)
      {
      case UPD:
         {
         /* handle UDP */
         }

      case TCP:
         {
         /* handle TCP */
         }

      default:
         {
         /* handle errors */
         }
      }
}
```

```
void read ()
{
   switch (protocol_type)
      {
      case UPD:
         {
         /* handle UDP */
         }

      case TCP:
         {
         /* handle TCP */
         }

      default:
         {
         /* handle errors */
         }
      }
}
```

```
void write ()
{
   switch (protocol_type)
      {
      case UPD:
         {
         /* handle UDP */
         }

      case TCP:
         {
         /* handle TCP */
         }

      default:
         {
         /* handle errors */
         }
      }
}
```

The switch constructs in all functions have to be maintained and kept consistent. When a new operation is added, it must support all existing protocols. When a new protocol is added, all

operations must be adapted. The latter also makes configuration control harder; adding a new protocol must not introduce errors in existing protocols.

In either implementation, we must verify that all protocols behave consistently.

