Towards Certification of Java Applications for Safety Critical Projects

A.Andy Walter[1]

1:aicas GmbH, Haid- und Neu-Straße 18, 76131 Karlsruhe, Germany

Abstract:

The increasing complexity of embedded systems also effects the area of safety critical applications. Modern development tools and approaches which are common for the development of desktop applications are not available for ADA. At the same time, using unsafe languages such as C and C++ bares a high risk in any embedded application.

A good successor for ADA would be Java: the well-defined language eliminates a good deal of potential errors inherently, which should ease certification efforts for safety critical applications. Unfortunately, the old certification standard DO-178B does not really cover the usage of object oriented languages. The certification process is hindered by a formalism, which neglects the benefits and safety that object oriented languages in general and Java in particular could introduce for the usage in safety critical applications. E.g., usage of automated memory management is not possible with reasonable effort, such that currently, developers of safety critical applications undertake enormous efforts of doing their own memory management, e.g., by using object pools, which is not only more effort, but also more dangerous than using a provably correct automated tool would be.

Luckily, the succeeding certification standard DO-178C will make certification of Java technology, including the use of virtual machine technology and garbage collection, easier.

Keywords: DO-178B, Safety Critical Java, JSR 302, Certification

1. Introduction

Ada has been the preferred language for safety critical applications, but this is beginning to change. The number of Ada developers is diminishing, while the complexity of applications is increasing. C and C++ are poor alternatives to Ada, and many ideas from the Ada community have made their way into the realtime Java specifications.

Though strongly related to standard Java technology such as J2SE and J2EE, realtime Java is really a different beast. Realtime Java sets itself apart by having much stronger threading semantics and a means of avoiding timing anomalies due garbage collection, ideally while maintaining the reference consistency automatic object deallocation ensures. In the past, reference consistency was maintained by disallowing or severely limiting dynamic memory management. This approach works well for state machine like tasks, but not for more complex applications. The up and coming Safety Critical Java standard (JSR 302) provides some more flexibility than currently tolerated by providing a stack like approach to memory allocation and deallocation. This will enable the Java language to be used at the highest criticality levels in the near term, but does not address increasing complexity well.

New work on object oriented technology in SG-5 of the SC 205 / WG 71 Plenary to update the DO-178 standards, will make certification of Java technology, including the use of virtual machine technology and garbage collection, easier.

This talk gives an overview about what error types can be completely avoided by Java in general, as well as it's strengths in combination with code verification. Important Java standards, such as the realtime Specification for Java (JSR 1 and JSR 282) and Safety Critical Java (JSR 302), are outlined, as well as proposed changes from SG-5 for object oriented technology. New garbage collection technology will also be covered. This should give the attendee a good background in the state-of-the-art of realtime Java Technology and safety certification.

2. The Realtime Specification for Java (RTSJ)

The Realtime Specification for Java[2], which was published by the Java Community in 2001 as JSR 1[4] and JSR 282[?], has been widely accepted as the standard library extension for realtime Java Virtual Machines and is used by most of the realtime Java applications today.

Other than realtime, RTSJ also provides helpful

standard classes for the general usage in embedded applications. Its extensions range from asynchronous event handlers - a more convenient way of responding to events from outside the Java world, such as interrupts and timer events - to a safe access for specific memory regions such as memory mapped I/O. With RTSJ, even device drivers can be written completely in Java and can easily be ported to other OSes and to other RTSJ compatible Java Virtual Machines. But RTSJ alone is not sufficient for certification.

Ordinary Java applications can be interrupted by Java's automated memory management system, called Garbage Collection, any time, as shown in Illustration 1. Obviously, this is not suitable for realtime applications.

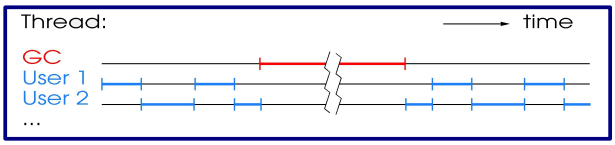

Illustration 1: Ordinary Java Threads can be interrupted by the Garbage Collection at any time.

RTSJ provides an acceptable, but inconvenient way for running realtime Java applications even on Java VMs without a realtime Garbage Collector, as shown in Illustration 2. Realtime Threads can have higher priorities than the Garbage Collector. Consequently, those Threads are not allowed to access any objects that might be affected by the Garbage Collector. Instead, RTSJ introduces scoped memory: Having entered a memory scope explicitly, new allocated objects will be stored in this scope. Scopes are hierarchic, as shown in Illustration 3. Scopes are left in the reverse order of which they were entered, such that assignments of objects to variables in a higher level store are illegal. In C or C++, this would result in a dangling pointer, which can be very difficult to locate, while in RTSJ, an Exception would be thrown.

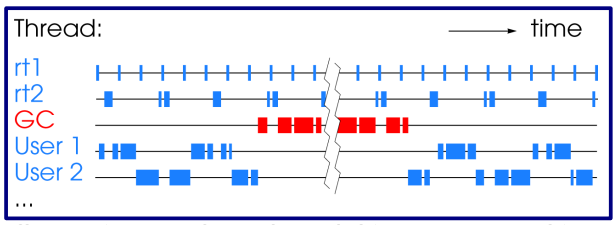

Illustration 2: Thread Model in RTSJ: Realtime Threads have higher priority than the Garbage Collection

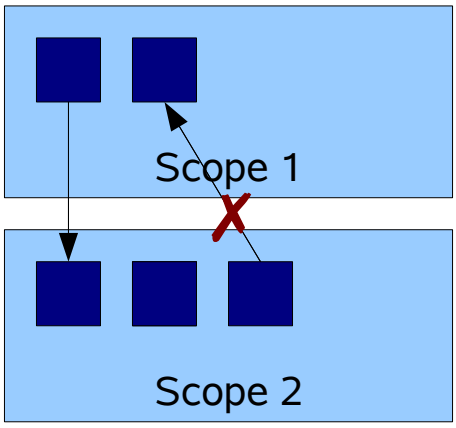

Illustration 3: Scoped memory in RTSJ prevents illegal assignments, which could lead to dangling pointers on leaving a scope, by throwing an exception at the time of the assignment already.

3. Realtime Garbage Collection

An automated memory management for realtime applications needs to guarantee an upper bound of time for each operation which is executed. Higher priority threads must be able to interrupt a running thread within a well-defined time, even if memory management is currently taking place. For this, the memory management (Garbage Collector) should not run in a dedicated thread, but at allocation time of new objects only: [9] and [10] describe JamaicaVM's garbage collector, which guarantees an upper bound of time for each object allocation and is deterministic. Illustration 4 illustrates this timing behaviour: In this example, the highest priority thread rt1 does not allocate any objects. Consequently, it is easy to prove that this thread by no means can be affected by the garbage collector. Nevertheless, even high priority, realtime threads can safely allocate objects with this approach.

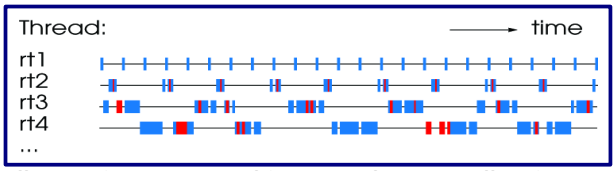

Illustration 4: A realtime Garbage Collection must be deterministic and incremental, such that higher priority threads can interrupt it any time.

4. Safety Critical Java: Towards Java Certification

In 2006, a new working group of the Java Community started developing a Java subset for the usage in safety critical applications. This standard, which will be published as JSR 302[?], is being designed to be within the limitations of current DO-178B certification practise. Safety Critical Java (SCJava) aims for Level A, the highest criticality level of DO-178B. In order to achieve this, only a very limited subset of the standard Java classes and the RTSJ will be part of SCJava. One of the restrictions is that no garbage collection is used - which means removing one of the strongest Java features compared to Ada, C, and C++. Instead, Safety Critical Java is based on RTSJ's Scoped Memory approach. This is a significant improvement over the usage of memory pools, which is common in safety critical applications. Due to the increasing complexity of safety critical applications, this kind of manual memory management is becoming more time-consuming as well as more error-prone. Even more complexity of applications could be reduced by the usage of a realtime garbage collector rather than ScopedMemory, if it can be proven that the garbage collection is both deterministic and realtime capable.

Despite of its limitations, SCJava is a good first step towards the certification of Java applications according to DO-178B, Level A. Object Orientation, reusability, and platform independence are good reasons to prefer SCJava over Ada or C for the use in safety critical applications.

5. DO-178C: A New Certification Standard

The biggest problem for the DO-178B certification of Java applications is that this old standard from 1992 does not consider Object Orientated Technology (OOT). While OOT helps software developers in writing reliable and portable applications with much less effort than imperative languages do, it also introduces a number of vulnerabilities. Those need to be identified and addressed in the certification process of safety critical applications. SG-5 of the SC-205/WG-71 plenary is currently working on a new set of objectives, activities, guidance, and guidelines for the certification of OOT software. These will become part of DO-178C[?], the successor to DO-178B. Unlike its predecessors, the new standard finally considers OOT explicitly. For each new feature that is introduced by OOT, the certification process has to prove, that it doesn't affect the application's safety.
The good news for Java developers is that most of the general vulnerabilities of OOT do not apply to Java, either because of the language specification or because of the automated memory management.

For safe OOT applications, it needs to be proven that the following vulnerabilities can not occur:

a.) Inheritance and Redefinition
Inheritance is a core feature of Object Orientation. The encapsulation of code and data provides a safe means for shared code within an application or even reusing it for other applications. Nevertheless, care should be taken to maintain compatibility of subclasses with their parents. In C++, static dispatch and multiple inheritance can make this difficult.

Since a subclass is also a subtype, a general problem with inheritance is maintaining type compatibility. Any subclass should be substitutable for its superclass. Liskov's Substitution Principle defines type compatibility formally. This can be stated succinctly using preconditions, postconditions, and invariants: compared with those of its superclass, no precondition may be strengthened, and no postcondition or invariant may be weakened in any subclass. Java Annotations are an appropriate way of making such preconditions and
postconditions part of the application. The verification activities for any class should also be performed on its subclasses. Vice versa, any class should be verified using its own *and* its parent's verification processes.

Another vulnerability is introduced with static dispatch: which implementation of an overwritten method is called, depends on the *declared* type of the object. This is misleading and confusing: despite a method having been overwritten, the original method might be called on a subclass. The behaviour of C++ is even worse than that: static and dynamic dispatch coexist and can even be mixed within the same class, such that some methods are dispatched statically (which is the default in C++) while others are declared *virtual* and dispatched dynamically. In Java, method dispatch is always virtual, such that, the method to be called will always be decided based on the *real* type of an object, not on the *declared* type.

Multiple inheritance introduces another vulnerability: it is often not clear which implementation of a method is called. Java avoids this confusion by allowing multiple inheritance only for interfaces, but not for implementations. While there is only one possible method implementation to be called, care needs to be taken with contradicting specifications introduced by different interfaces which declare the same method. However, multiple inheritance on the implementation level, as supported by C++, is a significantly complexer problem. In Java, it is sufficient to make sure that all methods are conforming to all of their declarations.

b.) Parametric Polymorphism
Parametric polymorphism, which is available in Java as Generics or in C++ as Templates, is a type-consistent means to write reusable code whenever sub-typing is not possible or convenient. An example is a List, each of which is should only have a certain type of element, but should be usable for all element types.

Parametric polymorphism is a strong feature of Object Oriented languages. For certification, each unique instantiation of a parametrised type or combination of types needs to be verified. Generics in Java are type safe, and consistency is checked at compile time as far as possible.

c.) Type Conversion
OO Type conversion, as well as in imperative programming languages, is sometimes necessary, but can cause unexpected behaviour. The strong type system of Java tries to detect type inconsistencies at compile time as far as possible, and throws an exception at runtime in all other cases. While throwing an exception is preferable over working with corrupted data (like C or C++ would do), certification requires a static proof of correctness. Static type checking, e.g., based on Data Flow Analysis, can help to ensure the correct typing of an application. Theoretically, Data Flow Analysis can also be used for analysing C or C++ programs, but the result is less reliable than in Java on any language that does not prohibit pointer arithmetics.

d.) Overloading
When used with care, overloading can improve readability and code maintenance. Along with implicit type conversion, which is common in Object Oriented programming languages, Overloading may cause ambiguity. In order to avoid this, guidelines should address the cases in which overloading is allowed and discourage the use of implicit type conversion.

e.) Exception Management
Most OO languages support throwing and handling exceptions instead of returning from a method with an ordinary return value. While Exceptions are a safe and convenient way to deal with exceptional situations, they introduce another vulnerability: exceptions might leave the application in an inconsistent state in case an unexpected exception is handled very low in the call stack or even not handled at all.
Java supports both checked and unchecked exceptions. Unless a method declares throwing a checked exception, it must be handled whenever this exception might be thrown within the scope of this method. The need to handle checked exceptions explicitly makes their usage safe. Unchecked exceptions are used for runtime errors such as division by zero, bounds checks, and range checks, and usually mainly thrown implicitly by the VM in situations, which would cause the system to either crash or run in an inconsistent state if this situation had occured in a programming language without exception management.

aicas has just released a Data Flow Analysis based tool, Veriflux, which can prove that all unchecked exceptions are handled by the application and detect all occurrences of unhandled unchecked exceptions. This approach ensures highest safety for critical applications.

f.) Dynamic Memory Management
Complex tasks often require temporary data. There are several possibilities to deal with the allocation and deallocation of this data, the safest and most convenient of which is an automated garbage collector. SG-5 has determined the following vulnerabilities related to dynamic memory management. In Java, all but Heap memory exhaustion can be avoided by a realtime capable automated garbage collector.

f 1.) Ambiguous References: Objects overlap in the same memory region. This can be avoided by an allocator which ensures exclusivity, provided that pointer arithmetics is not possible (as it is in Java) or not used (which would be necessary, but hard to ensure, e.g., in C or C++). Ambiguous references can also be a consequence of the deallocation of an object that was still in use.

f 2.) Fragmentation Starvation: A new object cannot be allocated due to memory fragmentation. To prevent this, memory should be organised in a way that ensures all allocations will succeed, provided that the system has enough of free memory. Many Java garbage collectors can ensure this. C applications, especially in Embedded Systems, are often vulnerable for fragmentation starvation.

f 3.) Deallocation Starvation: Garbage objects may not be freed, or at least not fast enough, causing the application to run out of memory. Some Java garbage collectors can guarantee that they reclaim memory fast enough to prevent this.

f 4.) Heap Memory Exhaustion: The Heap may be insufficient for the application to run. The application needs to ensure that all simultaneously live objects fit into the available memory. Again, data flow analysis is a safe approach to ensure this.

f 5.) Premature Deallocation: Objects may be removed although they are still in use. This causes dangling pointers, which is a common problem in C or C++ applications. However, all Java garbage collectors should be able to avoid this vulnerability.

f 6.) Lost Update and Stale Reference: In order to prevent fragmentation starvation, some systems move objects to a different location. Those need to make sure that modifications of an object which is being moved also effect the new location. Similarly, any read access to this object should return the new data of the modified object. Non-moving garbage collectors or one that move objects atomically are safe to use here.

f 7.) Indeterministic Allocation or Deallocation: Dynamic memory management could interrupt the application unexpectedly. Most automated garbage collectors that prevent the other vulnerabilities in this section, cannot guarantee determinism. For example, the Boehm garbage collector, which is commonly used in C++ applications, is vulnerable in this respect. The garbage collector of the JamaicaVM Java Virtual Machine guarantees determinism and avoids all other vulnerabilities in this section, with the exception of Heap memory exhaustion.

Various techniques exist to deal with those vulnerabilities, as shown in Tabular 1: manual heap allocation, which is commonly used in Embedded C applications that do not need to be certified, leave most of them to be dealt with by the application developers.

Technique	Ambigous References	Memory Fragmentation	Memory Holes	Not Enough Memory	Premature Deallocation	Lost Update / Stale References	Undeterministic Allocation / Deallocation
Manual Allocation	✗	?	✗	✗	✗	N/A	✔
Object Pooling	✗	✗	✗	✗	✗	N/A	✔
Stack Allokation	✗	✔	✔	✗	✗	N/A	✔
Scope Allokation	✔	✔	✔	✗	✗	N/A	✔
Automated Memory Management	✔	✔	✔	✗	✔	✔	✔

✔ = managed by system, ✗ = manually ensured in application
N/A = not applicable, ? = difficult to ensure

Tabular 1: Memory Management Techniques and problems related to them

Object pooling is only slightly better: rather than allocating and initialising a complex object from scratch when it is needed and destroying it again afterwards, the object could be taken from a prefilled object pool, which is much faster than the allocation. When the object is no longer needed, it is sent back to the object pool rather than destroyed. While object pools avoid the typical fragmentation of a *malloc*, they introduce a new kind of fragmentation: while free objects might still exist in some of the object pools, the pool holding the kind of objects to be allocated might be empty. Unlike manual heap allocation, this is at least some kind of fragmentation which the application developers have a chance for dealing with manually. An advantage of object pooling is the fast allocation time as long as there are enough objects left in the pool.
Stack allocation is used to store local object on the call stack, removing them automatically on method exit. While this lowers the danger of fragmentation, it also limits the extent to which frames can be shared between threads and is only usable for local objects. Scope based object management is slightly more flexible, because it allows for several threads to enter a certain frame simultaneously. At the same time, scope allocation causes a higher risk of fragmentation. The referencing rules of scoped memory prevent dangling references, which is not true for stack allication in languages such as C and C++.
Automated garbage collection is the most convenient and, if it solves the vulnerabilities identified by the SG-5, safest way to deal with dynamic memory. Care should be taken with the selection of the garbage collector: some cannot ensure that free memory can be detected early enough to avoid deallocation and fragmentation starvation or that an upper bound of allocation and deallocation time is not exceeded. The realtime Garbage Collector of JamaicaVM is able to guarantee all of this.

g.) Virtualisation

Virtualisation generally improves the portability and reusability of application code and typically reduces the complexity of applications. Java applications generally run in such a virtual environment. The main vulnerability here is that interpreted code is not sufficiently validated, because it is treated as data rather than code. DO-178C generally allows for virtualisation, but requires that each layer is verified independently, i.e., when certifying the interpreter, its input can be treated as data, but an additional verification of the interpreted code, in which the interpreter is treated as execution platform is also required.

6. Verification of Applications

Numerous sources for common programming errors can be removed by the usage of Java for critical applications: While they would entail severe runtime errors in C or C++ applications, which would be hard to locate and undeterministic, in Java applications they lead to a compiler error in the best case, or a clear runtime exception in the worst case. While this is helpful for locating bugs as early as possible, it is not sufficient as a proof of the correctness of the application.

Since pointer arithmetics is prohibited in Java, this language is well prepared for running static analysis based on formal methods: Dataflow Analysis can prove that certain types of errors do not occur in the application. Alternatively, locations can be found, for which such a proof was not possible. With this technique, the value set for each occurrence of a variable is determined. E.g., if this value set under no circumstances contains the value 0, dividing by this value is safe without potentially causing a DivisionByZeroException.

It is possible to run Dataflow Analysis on C or C++ applications as well, but the result is weaker: The Dataflow Analysis can not rely on determining the complete value set for all variables. Faulty pointer arithmetics, e.g., a wrong array access, could modify any value in the memory. Tabular 2 illustrates, when which error type can be detected.

	C	C++	Java	Java + Dataflow Analysis
Type Error	✘	✘	o	✔
Deadlocks	✘	✘	✘	✔
Dangling Pointers	✘	✘	✔	✔
Fragmented Memory	✘	✘	✔	✔
Array Range	✘	✘	o	✔
Race Conditions	✘	✘	✘	✔
Uninitialised Variables	✘	✘	✔	✔
Nullpointer Deref.	✘	✘	o	✔
Division by Zero	✘	✘	o	✔
Exceptions	✘	o	o	✔

✘ danger , o runtime error, ✔ detected during development

Tabular 2: Common programming errors and the time of their location

7. Conclusion

Object Oriented Technology in general and Java in particular improve the efficiency of developers by reducing the complexity of applications. Automated memory management, a strong type system and the prohibition of pointer arithmetics and multiple inheritance prevent hard-to-locate errors, which are common in C and C++ applications.

RTSJ is a safe extension for realtime applications. It is a reasonable, but insufficient standard for certified Java applications, e.g., to DO-178B.

Combined with a realtime garbage collection, certifications should be possible up to level C. The successor DO-178C, which is currently being defined, provides clear rules and guidelines for the certification of OOT applications, with or without automated garbage collection. This is an important mile stone in the development of future safety critical applications, which will have to solve more complex tasks than their currently deployed counterparts, and thus require better development tools.

8. References

[1] Software considerations in airborne systems and equipment certification. Advisory Circular DO-178B, Radio Technical Commission for Aeronautics, 1992. Errata Issued 3-26-99.
[2] Greg Bollela. Real-Time Specification for Java. Addison-Wesley, 2001.
[3] J. Chelini, P. Heller, and SG-5. Technical Supplement Template (Draft OOT supplement to support DO-178C, Revision C of 25th of June 2009). Technical Report RTCA/DO-OOT, SC-205/WG-71 Plenary, 2009.
[4] Peter Dibble. JSR 1, final release 3: RTSJ version 1.02. http://jcp.org/en/jsr/detail?id=1.
[5] Peter Dibble. JSR 282: RTSJ version 1.1. http://jcp.org/en/jsr/detail?id=282.
[6] Andy Walter, "Java in Safety Critical Systems", Embedded World Conference 2009
[7] James J. Hunt, Isabel Tonin, and Fridtjof Siebert, "Using global data flow analysis on bytecode to aid worst case execution time analysis for real-time Java programs". Volume 343 of ACM International Conference Proceeding Series, 2008.
[8] C. Douglass Locke. JSR 302:Safety Critical Java Technology. http://jcp.org/en/jsr/detail?id=302.
[9] Fridtjof Siebert. Hard Realtime Garbage Collection in Modern Object Oriented Programming Languages. aicas Books, 2002.
[10] Fridtjof Siebert. Realtime Garbage Collection in the JamaicaVM 3.0. 5th International Workshop on Java Technologies for Real-time and Embedded Systems (JTRES 2007), Vienna, Austria, September 2007. ACM Press.