

Analysis Tools for the HIJA Safety-Critical Java Model

Dr. Fridtjof Siebert
siebert@aicas.com
Director of Development
aicas GmbH
Haid-und-Neu-Straße 18
76131 Karlsruhe, Germany
www.aicas.com

March 31, 2005

Abstract

The European project HIJA (High-Integrity Java) [6] started its work on defining and implementing a new High-Integrity Java for future networked real-time embedded systems in June 2004. Based on the features of the Realtime Specification for Java (RTSJ) [3], a safety-critical profile is defined. This profile provides a restricted subset with the aim to permit certification up the DO178B level A.

The aspects covered by the profile address the supported thread model, synchronization mechanism, memory model and annotations that permit tools to perform correctness verification. In parallel, formal verification tools are developed to proof the functional and non-functional (resource use, etc.) correctness of an application.

1 Introduction

The enormous success of Java technology is due to the many advantages, such as higher productivity and safety, the language brings to the developer. Even critical applications, as in automotive or aerospace control, can profit from these advantages [5, 2].

Language extensions like the Realtime Specification for Java [3] have made it possible to use Java implementations even though a garbage collector may interrupt the execution of normal Java code in an unpredictable way.

New specific features such as realtime threads that cannot access the garbage collected heap make it possible to develop code that has predictable timing behavior and that can be used for hard realtime tasks.

Safety-Critical systems as defined in DO178B [1] levels A and B are software systems that are vital in a way such that anomalous behavior would cause or contribute to a failure of system function resulting in a catastrophic (level A) or hazardous/severe-major failure condition for the aircraft. A catastrophic failure is one which would prevent continued safe flight and landing, while a hazardous/severe-major failure is one which would reduce the capability of the aircraft or the ability of the crew to cope with adverse operating conditions to the extent that there would be, for example, a large reduction in safety margins or potentially fatal injuries to a small number of the aircrafts' occupants.

For Ada, the Ravenscar profile [4] provides a language for safety-critical applications that is powerful enough while remaining analysable. A corresponding profile is required for the use of Java in this area.

To ensure the correctness of a safety-critical software system, stringent verification techniques need to be applied that proof the absence of error conditions. Dynamic features such as dynamic memory management or even garbage collection or dynamic code loading are not techniques that currently can be verified in a way that would be accepted by certification authorities. For realtime Java to be applicable in such a system, a very stringent subset

of the features available in Java, the standard libraries and the RTSJ extensions is required.

Since the early versions of the RTSJ are available, work on defining subsets for the use in safety-critical systems has been performed. Of important influence to the HIJA project is the profile defined by Puschner and Wellings [10] and the Assessment of Java for High Integrity Systems by Kwon et al [8] and the Ravenscar-Java profile [9].

The HIJA project [6] started in June 2004 to define Java profiles for critical systems and to implement tools for the required verification of Java applications developed for these profiles. There are 12 partners in the HIJA consortium: aicas, Aonix, Bellstream, Fiat Research Centre, FZI Karlsruhe, Thales-Avionics, Telecom Italia, The Open Group, Trialog, the University of Karlsruhe, the University Polytechnic of Madrid, and the University of York,

2 Guiding Principles for SC-Java Profile

The safety-critical Java profile has the aim of providing a profile that maintains Java's advantages such as portability, inter-operability, object-orientation and the large tool and library base available for Java, while it provides a language subset that is simple enough that it can be analyzed by static tools and certified for safety-critical systems. The following main guiding principles were followed:

- **Conservative Approach:** No concepts that go too far for the safety-critical community should be introduced. Concepts that would go too far are, e.g., realtime garbage collection, or the asynchronous transfer of control provided by the RTSJ.
- **Do not fragment the market.** All specification should be based on standard Java and RTSJ. Ideally, safety-critical Java programs should run on a standard RTSJ JVM. Any extensions to the RTSJ that may be required for safety-critical Java shall be made through the RTSJ Technical Interpretation Committee.
- **New classes on top of RTSJ** will be added for the safety-critical profile. However, it must be possible to implement these

classes on top of a standard RTSJ implementation such that a safety-critical Java application may run on top of RTSJ.

- **Annotations** will be used to permit or facilitate off-line analysis of the code. It must, however, be possible to execute a safety-critical Java application without regarding these annotations.

3 HIJA SC-Java Profile

The HIJA safety-critical Java profile poses severe restrictions on the Java programs developed according to the profile. There is no provision for garbage collection in the profile. The execution is split into two distinct phases: an initialization phase and a mission phase. During the mission phase dynamic loading, thread creation and object allocation in non-local memory areas is not permitted. Instead, local memory areas for each task are used for allocation of temporary objects in the mission phase. Annotations are used to document which methods or classes are safe to be used in the mission phase.

3.1 Concurrency Model

The safety-critical profile uses the concurrency model of the RTSJ, i.e., fixed-priority preemptive scheduling using 28 distinct priority levels and FIFO execution for tasks with equal priorities.

Only two kinds of schedulable objects are permitted by the profile: Periodic threads and bound asynchronous event handlers. All threads and event handlers are created in the initialization phase and are of the no-heap kind, i.e, they cannot access the normal Java heap that is under the control of the garbage collector. Periodic threads have fixed periodic release parameters and may be used for any periodic activity required during the mission phase, while asynchronous event handlers have sporadic release parameters and are triggered by aperiodic external or internal events.

3.2 Synchronization

Java provides a powerful synchronization mechanism with monitors that are part of all Java objects. The RTSJ has extended these monitors for the use in realtime systems by

providing priority inheritance and the priority ceiling emulation protocol which avoid priority inversion.

For the safety-critical profile, it has to be possible to prove that no synchronization related errors are present in an application. Possible errors are potential dead-locks, priority inversion and race conditions. Both priority inheritance and priority ceiling avoid priority inversion. However, priority inheritance cannot prevent deadlocks when nested monitors are used, while priority ceiling is inherently deadlock-free as long as no thread blocks while holding a lock. For this reason, the safety-critical profile uses only the priority ceiling emulation protocol.

The default ceiling priority is set to the maximum priority, but the application can select different ceiling priorities for some monitors during the initialization phase. The verification of the correctness of priority ceiling settings is a remaining, non-trivial task that has to be performed by offline analysis tools.

To simplify the analysis for these tools, synchronization in the safety-critical profile is restricted to synchronized methods. The use of synchronized statements is not permitted since these statements may synchronize on arbitrary objects which can make static analysis more difficult.

3.3 Memory Management

No garbage collection is used in the safety-critical profile. In RTSJ terms, this means that no allocation in heap memory will be performed, since the heap is under the control of the garbage collector. Instead, all memory allocation will use either immortal memory, which is never reclaimed, or scoped memory, which provides a safe region-based memory management.

Since memory allocated in immortal memory will never be reclaimed, repeated dynamic allocation in this memory area will eventually cause the system to run out of available memory. Therefore, allocation in immortal memory is restricted to the initialization phase. All objects that need to be shared by different schedulable objects need to be allocated statically in immortal memory in the initialization phase. Once the mission phase has started, the amount of immortal memory that is used will

remain constant, there is no possibility to obtain an out of memory error due to allocation in immortal memory during the mission phase.

However, the allocation of temporary objects during the mission phase cannot be prohibited without sacrificing the object-oriented programming style that is encouraged by Java. To provide a safe means of allocation of temporary objects by periodic threads and event handlers during the mission phase, each of these schedulable objects has its own scoped memory region. No other scoped memory regions may be used, i.e., nesting of scoped memory regions is not permitted. The assignment rules for scoped memory regions in the RTSJ then automatically ensure that these temporary objects may not be shared between different schedulable objects.

The scopes will be entered automatically on each release of a schedulable object, and exited after the execution of schedulable object returns for this release. This ensures that all temporary objects allocated during one release will be reclaimed before the next release. Since the scopes are local to each schedulable object, the allocation of temporary objects in one task is not affected by the allocation in any other task. Any allocation-related errors in one schedulable object, such as uncontrolled allocation or too large array sizes, may not affect any other schedulable object.

With this memory model, what remains to be checked by static analysis is that no assignment errors may occur at runtime. An assignment error occurs whenever a reference that is allocated in scoped memory is assigned to a static variable or to an object that is allocated in a memory area that may have a longer lifespan, such as heap, immortal or a surrounding scoped memory. Since heap memory is not used in the profile and nesting of scoped memory is not possible, all the static analysis has to show is that there are no assignments of temporary objects into static variables or objects in immortal memory.

To avoid running out of memory within the scoped memory and the runtime stack that is local to one schedulable object, it is furthermore required to perform a worst-case heap and stack use analysis for each schedulable object. This analysis can now be performed locally to one schedulable object, it is simplified significantly.

4 Tools for correctness proof of application

A whole set of tools for different level of verifications are developed within the HIJA project. This chapter gives a short overview over these tools.

The functional correctness verification using the *KeY* tool will use very detailed annotations using the Java Modeling Language JML and provide functional correctness verification results to the developer.

A program-wide data flow analysis (*DFA*) based on the intermediate representation used by the Jamaica [7] compiler will provide control and data flow information that will be needed by tools for non-functional correctness analysis.

The worst-case execution time analysis (*WCET*) uses the output from *DFA*, the binary executable code of the application and additional bounds information to determine the worst case execution time of tasks in the system.

Similarly to the worst-case execution time analysis, the *Memory Usage* analysis tool determines the worst-case heap and stack usage for all tasks in the system. This analysis requires input from *DFA* and additional bounds information.

The *Code Consistency* analysis performs correctness verifications that do not require any additional input apart from the *DFA*. This analysis proves the absence of certain error conditions such as runtime errors (assignments, casts, etc.) and deadlock conditions.

Annotations made to Java code by the developer can be verified by the *Annotation Check* tool that is based on the *DFA* and, of course, requires access to the annotations made by the user.

Finally, the *Model Generator* uses input from the class files, bounds information provided via annotations and the results from *WCET* and *Memory Usage* analysis to generate a model that can be verified by the *UPPAAL* model checker.

All these non-functional correctness verification tools provide output to the developer to permit modification to the source or annotation to reach a level of proven correctness required for the application.

5 Conclusion

The extensions defined by the Realtime Specification for Java enable the use of Java in new application domains that have strict requirements on the timing behavior of the application. However, the use in safety-critical applications requires a severe restriction of the provided functionality to permit static analysis and certification.

The HIJA project provides such a safety-critical Java profile and develops tools for the correctness verification of applications developed in this profile. This is a large step forward towards the certification of Java for safety-critical applications. The advantages in productivity and safety that the use of Java technology has brought to a broad range of different applications hence become available to the development of safety-critical applications.

References

- [1] Rtca/do-178b software considerations in airborne systems and equipment certification, 1992.
- [2] Aero-vm, the hard realtime virtual machine for onboard space systems. www.aero-vm.com, 2003.
- [3] Greg Bollela. *Real-Time Specification for Java*. Addison-Wesley, 2001.
- [4] A. Burns, B. Dobbing, and G. Romanski. The ravenscar tasking profile for high integrity real-time programs. *Proceedings of Ada-Europe 98*, 1411:263–275.
- [5] Hidoors, high integrity object-oriented realtime systems. www.hidoors.org, 2002-2004.
- [6] Hija, high-integrity java. www.hija.info, 2004-2006.
- [7] Jamaica virtual machine. www.aicas.com/jamaica, 1999-2005.
- [8] J. Kwon, A. Wellings, and S. King. Assessment of the java programming language for use in high integrity systems. Technical Report YCS 341 (2002), University of York, 2002.

- [9] Jagun Kwon, Andy Wellings, and Steve King. Ravenscar-java: a high-integrity profile for real-time java. *Concurrency and Computation: Practice and Experience*, 17(5):681–713, February 2005.
- [10] P. Puscher and A. J. Wellings. A profile for high integrity real-time java programs. *Proceedings of the 4th IEEE International Symposium on Object-oriented Real-time distributed Computing (ISORC)*, 2001.