A Technology Compatibility Kit for Safety Critical Java

Lei Zhao Daniel Tang Jan Vitek
Purdue University

ABSTRACT

Safety Critical Java is a specification being built on top a subset of interfaces from the Real-Time Specification for Java. It is designed to ease development and analysis of safety critical programs that have to be certified. Though Real-Time Java was developed to add hard real-time support to Java, it permits too many freedoms that are not desirable for safety critical programs. The Safety Critical Specification for Java aims to deliver a more restricted programming model and provide a predictable structure and data flow that will ease analysis of applications written against it. Since the specification is being developed with the Java Community Process, a Technology Compatibility Kit is required so that all aspiring implementations can check for conformance to the specification. In this paper, we present the first steps towards an open source compatibility kit. We have developed it to test other implementations, including the reference implementation being developed concurrently with the specification.

1. INTRODUCTION

The Real-Time Specification for Java (RTSJ) [5] is an extension to Java that provides real-time capabilities, enabling development of programs with hard real-time constraints in Java. RTSJ uses region-based memory management to curtail unbounded delays caused by the garbage collector and provide predictable timing, which is necessary for hard real-time programs. However, there are a number of characteristics about RTSJ that make it unsuitable for safety critical systems. Safety Critical Java (SCJ) [11] is being developed to provide a real-time framework based on a subset of RTSJ that is also suitable for the development and analysis of safety critical programs for safety critical certification. For example, it removes the ability to allocate from the heap, which is still present in RTSJ. The specification also aims to leverage static checking techniques to remove runtime exceptions caused by bad memory accesses.

Three pieces are required in the Java Community Process (JCP) for a Java Specification Request (JSR) ratification: the specification, the reference implementation (RI), and the Technology Compatibility Kit (TCK). As the specification and RI are evolving alongside each other, the need for a SCJ TCK is becoming more urgent.

A TCK is a suite of tests that determine whether or not an implementation conforms to its corresponding specification. Although every approved JSR release contains a TCK, little literature has been published on the subject of TCKs. An example TCK can be found in the Java Compatibility Kit (JCK) [7], a TCK for Java Standard Edition (J2SE) 6 [6], which was initiated and is supported by Sun Microsystems. It consists of tests on the compiler, runtime, and development tools. The JCK source is released under a read-only license [8]. For TCK development, the JCP provides the Java Compatibility Test Tools (Java CTT) [9] for assistance in many aspects. The package contains a test harness, signature test tool, and several instructional documents, as well as a Spec Trac tool which can associate tests with the specification assertions. However, Java CTT is only free to particular JCP members. JUnit [12] is a known open-source testing framework. However, JUnit is designed to test standard Java programs in a per unit way, it is not suitable for direct use in the development of SCJ TCK, which can get limited runtime supports and requires entire program tests.

Generally, performance is not taken into account in judging the compatibility of an implementation to a specification. However, due to the nature of real-time systems, timeliness is also an important factor in judgment. Real-time benchmarks, such as [2] [3] [1], should also be integrated into TCKs if possible.

In this paper, we present our work on an open source TCK for Safety Critical Java. Our TCK focuses on functionality and behavior tests. We are also developing a benchmarking suite for testing timeliness and performance. The TCK will eventually also include signature tests, but this is not particularly interesting, since open source signature testing tools already exist [14].

In our work, we try to measure how well the TCK is designed. Unfortunately, due to the lack of conventional criteria, measurement is difficult. We rate the TCK by considering the following three quantities, where higher values are desired.

- RI code coverage rate;
- the amount of tested assertions in the specification;
- the amount of failure messages the TCK can report.

Clearly, the code coverage rate reflects how comprehensive the TCK can is in testing code paths. A higher coverage rate means less untested code, which are the potential places where specification violations can avoid detection. Although not the whole story, the amount of assertions and failure messages show the test's completeness in one dimension.

The main contributions of this paper are:

1. development of an open source SCJ TCK;
2. results of using it against the current SCJ RI;

3. a report and analysis of code coverage and failure messages.

The remainder of the paper is organized as follows: Section 2 gives a more detailed overview of Safety Critical Java; section 3 describes what assertions the TCK tests and explains testing logic; section 4 gives implementation details of the TCK; section 5 presents testing results on the current reference implementation; section 6 gives conclusive statements. The appendix sections provide information about where to find the source code, how to run the TCK and get code coverage report, as well as a summary of all failure messages.

2. SAFETY CRITICAL JAVA

Safety Critical Java was born from the desire to design programs that were not only more predictable, but also highly reliable. When these programs concern the well-being of something or someone, they are said to be *safety critical*. Safety critical programs, due to the risk they incur, must undergo certification to be used in real systems; for example, in avionics, DO-178B/ED-12B is required in the United States and Europe, respectively [13]. While SCJ does not and cannot ensure that applications built against it will be certifiable, SCJ provides a very strict model which makes analysis easier, compared to potentially more freeform RTSJ or standard Java programs.

SCJ programs are composed of one or more *missions*. Each mission is composed of a bounded number of schedulable objects, the types of which are restricted by SCJ compliance levels. Level 2 is the most liberal compliance level, and allows `NoHeapRealtimeThread` objects to run, while level 0 is the most restrictive, which uses a frame-based model of execution. These levels of compliance refer to the expected cost and difficulty of certification, rather than any actual relation to the certification process itself.

Throughout execution of an SCJ program, allocation to heap memory is not allowed. This is important, because heap memory implies garbage collection and thus unbounded delay. Each mission is given its own dedicated memory region for allocation during an initialization phase, and each schedulable object is given its own private memory to allocate to.

These are arguably the two outstanding characteristics of SCJ. More information will be presented while also presenting the TCK in the following section.

3. TCK DESCRIPTION

The SCJ APIs consist of a subset of plain Java and RTSJ APIs as well as a newly introduced package `javax.safetycritical`. Given that TCKs exist for standard Java and RTSJ, we concentrate our TCK on the new SCJ package. Requirements are extracted from the SCJ specification, and the enforcement of which are tested. In our following discussion, some of the requirements from the SCJ specification will be quoted. We will go through most of them; a complete failure message list can be found in Appendix C. Some kinds of requirements are not and will not be included in check. This is for the following three reasons.

1. They are not testable from user-code, e.g. for the assertion of "the thread that executes the initialization phase is a bound asynchronous event handler", the thread itself has no way to determine whether it is a schedulable object (which will be discussed in Section 3.2.1) or what kind of schedulable object it is. The enforcement of such as assertions relies on a careful runtime system implementation.

2. They are requirements that require static analysis, e.g. "no use of the synchronized statement is allowed". Applications with prohibited syntax should not compile or verify. The TCK works based on the assumption that there are well functioning compilers or checkers able to rule out all syntax errors, which we think is reasonable.

3. They need negative tests, e.g. "deadline miss detection is not supported". Clearly, there is no need to label those with the feature of deadline miss detection as failed implementations.

Following we discuss the testing approaches taken in every aspect of SCJ. The discussion is organized with the same section structure as in SCJ specification.

3.1 Mission Life Cycle

An SCJ application must be organized as a series of *missions*, whose order is specified by the *mission sequencer* and issued by the infrastructure. The mission life cycle consists of three phases: *initialization*, *execution*, and *cleanup*. A mission can be restarted as necessary.

We have tested that the infrastructure can issue the series of missions completely and without lifetime overlaps. To test for completeness, a global mission counter is maintained to show how many missions have been actually issued. The difference between the final value of the counter and the real mission numbers implies a failure of the infrastructure. To test the serializability of missions, the *initialization* and *cleanup* time of each mission are recorded and checked for linearity as soon as the whole application terminates. Aside from the above requirements, which apply to all levels, some are level specific. On level 0, the workload in a mission is further divided into *frames*, which are also issued by the infrastructure. We test that "the frames will be issued according to their pre-defined order". On level 2, "applications are allowed to launch nesting missions". In the TCK, a nesting mission which contains all kinds of schedulable objects are launched. The schedulable objects check the indicating variables if they get executed successfully.

3.2 Concurrency and Scheduling Models

3.2.1 Schedulable Objects

In SCJ, work is carried out through *schedulable objects*. Three kinds of `Schedulable` objects are supported: `PeriodicEventHandler` (PEH), `AperiodicEventHandler` (AEH), and `NoHeapRealtimeThread` (NHRT). Regular Java threads and RTSJ real-time threads that use heap are forbidden. A PEH is a periodically released schedulable object whose release is ensured by the SCJ runtime. The functions of AEH and NHRT are analogous with their counterparts in RTSJ. For level considerations, PEH objects are allowed on all levels; AEH is allowed on level 1 and 2; NHRT is only allowed on level 2. In the TCK, all testing logic is delivered via schedulable objects.

SCJ requires that "the schedulable objects must be noheap and non-daemon". This is tested on level 2, where all kinds of schedulable objects are allowed to run. While the check of *non-daemon* is straightforward by consulting `Thread.isDaemon()` method on current threads, that of *no-heap* is to some extent guaranteed automatically. Because heap memory is not part of the SCJ specification, programmers would have no chance to allocate in the heap explicitly (which known can be done in RTSJ by using `HeapMemory.instance().newInstance()`). Because of this, we want to make sure that the active memory areas during mission life cycle would always be either immortal memory or scoped memory, so that an implicit allocation in heap is not possible. However, though the TCK can guarantee no use of heap by user logic, it cannot determine things on the infrastructure side. For a virtual machine which has heap on it but also supports SCJ, the absence of heap interface actually prevents the TCK from monitoring the heap usage. As a result, improperly creating auxiliary objects in the heap by the infrastructure for some implementing purposes during the mission life cycle cannot be reflected by the TCK. The infrastructure designers should always make sure no heap is used in SCJ sphere.

3.2.2 Synchronization

Like plain Java, synchronization in SCJ is performed by using the `synchronized` keyword or `wait/notify` features, but with some restrictions applied:

- "No use of the synchronized statement is allowed"
- "The receiver of `Object.wait()` should only be `this`"
- "Synchronized methods are supported and nested calls from one synchronized method to another are allowed"

For the reason stated at the beginning of this section, we do not test the first two assertions. The test for the last one is done by simply defining a number of synchronized methods nested together and invoking the outermost during mission. It is expected that the deepest nesting method could be eventually invoked.

Beside the above restrictions, which are applicable to all levels, "synchronized code is not allowed to self-suspend" applies to level 0 and 1 applications. Violation to this will cause an `IllegalMonitorStateException` to be thrown. Self-suspension can be the result of `sleep` method calls or blocking I/O operations; requesting a lock (via the synchronized method) is not considered self-suspension. To test this, we define a class with two methods: one self-suspends by calling `Thread.sleep()` and the other just invokes some synchronized method, which not considered self-suspension. An `IllegalMonitorStateException` should be caught outside the self-suspending method on level 0 and 1, while no exception outside the other. On level 2, no exception should be caught when either is invoked.

3.2.3 Scheduling Model

SCJ uses a preemptive, priority-based scheduling model with priority ceiling emulation (PCE) for priority inversion management. Unlike RTSJ, priority inheritance is not supported. SCJ requires that "the default ceiling for locks should be equal to the maximum priority that current priority scheduler can provide". This can be tested by comparing the results of `System.getDefaultCeiling()` against that of `PriorityScheduler.instance().getMaxPriority()`.

For the requirement "on level 0, only one thread of control shall be provided by the real-time virtual machine to execute handlers", we do the necessary, but insufficient test – make sure all PEHs are executed by the same thread. The insufficiency of the test lies in that they can happen to be executed by the same thread. At all events, an enforcement to this requirement implies the assertion that "the handlers shall be executed non preemptively".

As the specification states, "full preemptive scheduling should be supported on level 1 and 2". The test for this is carried out on three threads with different priorities: L (low), M (median), H (high), which are launched in order of L, H, M. Each records its actual starting and ending time. The thread life time is adjusted to be long enough so that preemption has a definite chance of happening. Under a correctly implemented scheduler, we should get a monotonically increasing time record with the form of $t^L_{start} < t^H_{start} < t^H_{end} < t^M_{start} < t^M_{end} < t^L_{end}$, where the subscript indicates whether it is starting or ending time, while the superscript indicates the thread.

The test for priority ceiling emulation is mostly similar to the preemption test, but with following changes: the L and H threads serialize their execution by synchronizing on a lock, while the M thread synchronizes on a lock for which no other thread competes. With L correctly lifting its priority to the maximum as entering its critical section, we expect to get the time record of $t^L_{start} < t^L_{end} < t^H_{start} < t^H_{end} < t^M_{start} < t^M_{end}$.

Testing that "a preempted schedulable object must be placed at the front of the run queue for its active priority level" is somewhat complex. This requirement is enforced if and only if following assertion is true: the preempted thread is the first one, among all those on the same priority level, getting back to run upon the preempter finishes. The major issues here are how a thread knows it is the first one being scheduled after a preemption happens and how it knows who is the last one that has just been preempted. Our solution is to maintain two shared variables: `preempted` indicates whether or not a thread has been preempted and `curRunner` stores the ID of the just-preempted thread. When the mission starts, several low priority threads run in a loop to compete to assign `curRunner` with their own ID in order to indicate who the current running thread is. Clearly, once the preemption occurs, the thread being preempted will have its ID left in `curRunner`. A high-priority thread is defined to kick in periodically to preempt low threads and set `preempted` to true to declare a preemption. After it is done, the first low thread being scheduled will notice the preemption by checking `preempted`. Then, the low thread will check `curRunner` for the preemptee's ID. If it gets its own ID, it knows it was correctly placed at the front of the run queue for its priority level. Otherwise, the failure will be reported.

3.2.4 Interrupt Handling

Compared to the RTSJ, SCJ aims to enhance support for interrupt handling; however, the discussion of which features are to be supported is still in progress. Basically, SCJ defines its own external event interface and will support handling both software and hardware interrupts. We have an elementary testcase for this, but since we lack meaningful definition of the *happening* string and interrupt ID, we leave this for future work.

3.3 Memory

RTSJ extends standard Java with immortal and scoped memory regions, which enable programmers to get more control of memory allocation and deallocation and their associated delays. However, as mentioned above, SCJ removes heap memory access to completely eliminate the option, which makes time analysis much simpler. It also restricts the way scoped memory regions are used.

3.3.1 Memory Model

All objects needed by the mission should be allocated in the mission initialization phase, where "the default allocation area is *mission memory*". All schedulable objects share the mission memory and immortal memory, but will allocate in their own *private memory* by default.

It is required that "the mission memory and private memory are scoped memory, and the private memory should also be `LTMemory`". We test this by acquiring the memory area instance during mission initialization and execution phases, and checking their classes. Mission memory should be an instance of `ScopedMemory` and private memory should be that of `LTMemory`.

The immortal and mission memories should hold mission global objects, so we test that "the global objects can be accessed by all schedulable objects during a mission". According to the SCJ specification, "object creation in mission scoped memory or immortal memory during the mission phase is not encouraged but allowed" and "mission memory is resizable". The TCK covers both of them.

3.3.2 Nesting Scopes

Except level 0, "new private memory instances are allowed to be created and nested during mission". We test this by simply creating several nesting private memory instances in a PEH and checking the deepest nesting one can be entered.

In order to simplify the nesting structure, SCJ requires that "a given scoped memory can only be entered by a single thread at any given time", and "a scope may only be entered from the memory area in which it is created." We test the first assertion by arranging several PEHs to try to enter a scoped memory simultaneously. According to the specification, after the private memory is occupied by the first PEH, a `MemoryInUseException` should be thrown upon the second PEH is about to enter. If more than one PEH successfully enters the private memory, a `ScopedCycleException` should be caught due to the violation of single parent rule – a memory region can only have one parent region, and multiple PEHs entering the region would result in several. In this case, a failure report is generated. For the second assertion, an intentional violation is made by creating two private memories in the same place and entering one from another. If this can be done without any exception being thrown, an implementation failure is detected.

3.4 Clocks, Timers, and Time

SCJ inherits the RTSJ classes `AbsoluteTime`, `RelativeTime`, and `HighResolutionTime` for representing time. It supports only a single system real-time clock. No timer is supported. What has been tested in this section includes:

- "The real-time clock should be monotonic and non-decreasing."
- "`HighResolutionTime.waitForObject()` method should be allowed on level 2."

3.5 JNI

SCJ provides very restricted JNI support. The supported JNI services are listed below, all of which have been tested in straightforward ways.

Version Information
`GetVersion`
General Object Analysis
`GetObjectClass`
`IsInstanceOf`
`IsSameObject`
`GetSuperclass`
`IsAssignableFrom`
String Functions
`GetStringLength`
`GetStringChars`
`ReleaseStringChars`
`GetStringUTFLength`
`GetStringUTFChars`
`ReleaseStringUTFChars`
`GetStringRegion`
`GetStringUTFRegion`
Array Operations
`GetArrayLength`
`GetObjectArrayElement`
`SetObjectArrayElement`
`New<PrimitiveType>Array` Routines
`Get<PrimitiveType>ArrayElements` Routines
`Release<PrimitiveType>ArrayElements` Routines
`Get<PrimitiveType>ArrayRegion` Routines
`Set<PrimitiveType>ArrayRegion` Routines
NIO Support
`NewDirectByteBuffer`
`GetDirectBufferAddress`
`GetDirectBufferCapacity`

3.6 Exceptions

SCJ's exception feature is similar to the RTSJ's when dealing with exceptions thrown across memory boundaries. The only differences lie in two aspects of `ThrowBoundaryError`. First is the exception allocation place. In RTSJ, with an occurrence of a boundary-crossing exception, a `ThrowBoundaryError` exception will be allocated in the enclosing scoped memory. In SCJ, exceptions "behave as if there is a pre-allocated exception instance on a per-schedulable object basis." As the specification does not explicitly specify the allocation location, we just test that the `ThrowBoundaryError` exception is not allocated in the scoped memory which is created by user logic. Such kind of memory is created on the fly while the schedulable object runs, therefore logically impossible to be pre-allocated in before the schedulable object actually starts. Obviously, the test is necessary but insufficient. SCJ also defines its own `ThrowBoundaryError`, which extends the RTSJ version with following functions:

- `getPropagatedExceptionClass()`
- `getPropagatedMessage()`
- `getPropagatedStackTraceDepth()`

- `getPropagatedStackTrace()`

Information of the original exception can be retrieved with these new features. In our testcase, a scenario of cross-scope boundary exception propagation is constructed. The class and message of the original exception are examined. In order to check the stack trace, the same exception is made to be thrown without crossing boundary. The stack trace (the elements and depth) between the two scenarios should be the same. However, this seemingly reasonable assumption is not necessarily true. Since boundary exceptions are preallocated, the stack trace can only hold so many items, which may result in a truncated stack trace. To try to avoid false failure reports, we choose to just compare the top several elements. The compared element amount can either be set to maximum length of stack trace or kept in a reasonable small range if no such information is available so that no truncation is likely to happen.

4. IMPLEMENTATION

In this section, we describe the implementation details of the TCK, and how to create new testcases within the current framework.

4.1 Framework

The TCK is organized as one testcase per SCJ application manner instead of squeezing all tests into one application. This is partly forced by the characteristics of SCJ applications that the compliance level is fixed throughout the application lifetime. Starting a new application enables level adjustment and testing of level-specific features. The TCK framework is shown in Figure 1. All SCJ applications must implement the `Safelet` interface. All testcases in the TCK subclass a `MainSafelet` abstract class, which implements `Safelet` and helper methods that are useful for testcases and benchmarks. For example, `MainSafelet` includes a `Properties` object which is loaded at runtime from a configuration file and contains parameters such as period length, mission memory size, running level, etc.
The availability of the file system is not guaranteed by the SCJ specification. Since our target SCJ RI is implemented based on J2SE and RTSJ, we can make use of file system

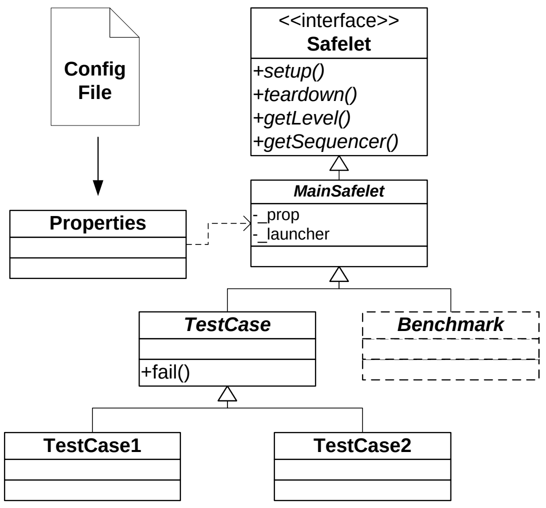

Figure 1: TCK Framework

to ease the property configuration. For those running in an environment without file systems, all parameters have to be hardcoded for each run, which introduces no difficulties but takes more effort in regenerating TCK binaries.

Failure Report. Our `TestCase` class has a `String` set for storing the failure messages; `fail()` can be used to report such failure wherever an error is detected.

4.2 Start and Termination

An SCJ application launched by the infrastructure may require termination if necessary. Ideally, the infrastructure should be able to notice the termination of the application and shift to the cleanup phase. As no complete SCJ infrastructure is available thus far, we have written a plain Java launcher to emulate the SCJ infrastructure on a RTSJ-compliant virtual machine. Some synchronization between the launcher thread and testcase thread is needed – the testcase notifies the launcher of its termination in order for an external script to be able to run all of following the testcases without overlap. To achieve this, a `_launcher` field in `MainSafelet` is maintained and assigned with reference to the launcher thread while `setup()` is invoked. The launcher will sleep until it is interrupted by the testcase after all work is done.

4.3 Extending the TCK

The SCJ specification is still in development. The appearance of new features will require extensions to the current TCK. Extensions are also expected while the `java` and `javax.realtime` packages in SCJ are taken into consideration. We claim that adding more testcases to current testcase set is very easy. To extend the TCK, following steps may be taken:

1. Define a testcase class extending `TestCase`.
2. Organize the test logic in a mission sequence.
3. Invoke `TestCase.fail()` to report failure.
4. Interrupt the launcher thread at the cleanup phase of the last mission.
5. Define more properties in `Properties` class as need and providing proper configuration files.

Several pre-configured SCJ application components are provided in `MainSafelet` as inner classes, which further facilitates testcase creation and simplifies code as well. For example, a `GeneralMission` class exists that automatically calls its parent `Mission` constructor based on the parameters in the `Properties` object. Other classes do the same with their parent constructors, but of course, the parent classes can be used directly if necessary.

5. TESTING RESULTS

Currently, there is no full-fledged SCJ RI released; the SCJ RI developed by the JSR 302[10] expert group is still incomplete. It implements most of the classes in `javax. safetycritical`, but is still missing a few components, which will be mentioned. However, we tested it against our TCK anyway, as it is the only implementation of SCJ at all at the present state. Our results are presented below.

5.1 Code Coverage

In this section, we report code coverage rates, which are one indication of how thoroughly the TCK tests SCJ. We measure the code coverage rate on `javax.safetycritical` package by using the open-source tool EMMA [4], which supports offline Java bytecode instrumentation. The result is achieved by running the instrumented TCK and SCJ classes; EMMA can automatically merge the coverage of each single testcase into a comprehensive report. Tables 2 and 3 show the merged results of `javax.safetycritical` package. Coverage rate is calculated with respect to class, method, block, and line respectively. Table 2 is the summary for the whole package, and table 3 breaks it down into files. Files are sorted by block coverage rate. From examination of the blocks without coverage, it turns out that most of them are the result of exception and recovery mechanisms which are just for implementation purposes. For those documented exceptions, for example the `IllegalArgumentException` upon feeding a negative memory size to an event handler, we test that they can actually be thrown with illegal parameters. A second important reason for non-covered code comes from non-invoked small methods, such as setter/getter, `toString()`, `getName()`, and so on. They have not been taken into consideration by current stage of specification and can be added into test easily once they have been. Detailed classification is shown below:

Exception & Recovery
`MissionMemory, MissionManager`
`LevelZeroManager, LevelOneManager`
`Mission, MissionSequencer, ManagedEventHandler`
Small Methods
`Mission, PrivateMemory`

Currently, `ThrowBoundaryError` has no coverage at all. This is simply because the RI currently has no implementation for the SCJ version of the class, so the RTSJ version is thrown instead. Once the RI implements the class, the coverage rate will naturally rise above 0. The next least covered classes are `ExternalEvent` and `InterruptHandler`. The reason, as previously stated in section 3, is that interrupt handling is virtual machine dependent (which may even not supported); it is impossible to give a general test. As the happening string and interruption ID are given arbitrary values in the TCK, these two classes do not fully function during tests. `LevelOneManager`, which manages interrupt handling, has its related code that is also not covered as consequence.

5.2 Detected Failures

In this section, we report detected RI failures. As stated above, the RI we tested against is incomplete. A few of its methods have dummy implementations. We show the detected specification violations without the intention to report RI bugs, but instead to give a feeling of how well the TCK can work in practice.

Ten RI failures are detected by the TCK. The corresponding messages are listed in Table1. Note that the interrupt handler test is excluded due to the lack of a meaningful way to test. A proper test requires a happening string or interruption ID, which are VM-dependent. Without documentation on such, we are unable to implement a meaningful test.

The failures can be classified into three groups according to their causes. The first two failures occur because the current RI is built based on existing RTSJ implementations. The maximum priority of the scheduler has not been adjusted to be equal to the default ceiling of monitors. As the RTSJ runtime would not be aware of any SCJ requirements, the old `javax.realtime.ThrowBoundaryError` instance will still be thrown upon the happening of cross scope boundary propagations.

The second group, failures 3, 4, and 5, results from the broken `PrivateMemory` class. It does not perform the runtime checks for nesting and parent scope at all. While it does attempt to check that only one thread can be in a private memory at a time, the checking logic is incorrect: each private memory instance relies on a field indicating whether it is occupied or not. Unfortunately, while the first thread entering it will set the field to true, the second will incur a `MemoryInUseException` and reset the field to false, which will wrongly allow the third thread to enter the occupied private memory without incurring any exception.

The last four failures are simply due to the dummy implementations of a series of `sleep` methods. These methods do not function at all.

6. CONCLUSIONS

In this paper, we presented an open source SCJ TCK. A TCK is an integral part of a JCP specification, which makes it a high-priority task to complete, alongside the specification itself. The aspects that we covered span the mission life cycle, scheduling, memory, clock, JNI, and exceptions. Many of the assertions in the specification were presented to demonstrate how the TCK tests them. We also showed some results of running the TCK against the current RI; the results show high code coverage, which we believe indicates that the TCK is thorough in testing. If more implementations show otherwise, the TCK is easily extendable with new testcases, with little to worry about aside from the testing logic. The detected failures were also stated and the causes were discussed.

Our current TCK emphasizes functional tests of the `javax.safetycritical` package. In future work, the completeness of APIs including those in `java` and `javax.realtime` packages would be a concern as well. The SCJ specification is still in development; pending issues will eventually be decided. Accordingly, we need to make the corresponding tests, such as the interrupt handling test, work. As the specification grows, more testcases may be added into the TCK as well.

7. ACKNOWLEDGMENTS

The authors thank Martin Schoeberl from Vienna University of Technology, Anders P. Ravn from Aalborg University for their comments and advice in paper revision. The authors thank Mike Fulton and Shirish Kuncolienkar from IBM and Wolfgang Puffitsch from Vienna University of Technology for the feedback of testing the TCK on related virtual machines. The authors also thank James Hunt, Andy Wellings, and Tomas Kalibera for providing helpful information in the TCK development.

8. REFERENCES

[1] A. Corsaro and D. C. Schmidt. Evaluating real-time java features and performance for real-time embedded systems. In *RTAS '02: Proceedings of the Eighth IEEE Real-Time and Embedded Technology and Applications*

Symposium (RTAS'02), page 90, Washington, DC, USA, 2002. IEEE Computer Society.

[2] B. P. Doherty. A real-time benchmark for java^{TM}. In *JTRES '07: Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, pages 35–46, New York, NY, USA, 2007. ACM.

[3] D. Dvorak, G. Bollella, T. Canham, V. Carson, V. Champlin, B. Giovannoni, M. Indictor, K. Meyer, A. Murray, and K. Reinholtz. Project golden gate: towards real-time java in space missions. In *Object-Oriented Real-Time Distributed Computing, 2004. Proceedings. Seventh IEEE International Symposium on*, pages 15–22, May 2004.

[4] http://emma.sourceforge.net/.

[5] J. Gosling and G. Bollella. *The Real-Time Specification for Java*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000.

[6] http://java.sun.com/javase/6/.

[7] https://jck.dev.java.net/.

[8] https://jck.dev.java.net/jck-read-only-license.txt.

[9] http://jcp.org/en/resources/tdk.

[10] http://jcp.org/en/jsr/detail?id=302.

[11] JSR302-Expert-Group. *Safety Critical Specification for Java (Version 0.66 Draft)*. 1 May 2009.

[12] http://www.junit.org/.

[13] RTCA. *DO178B / ED12B, Software Considerations in Airborne Systems and Equipment Certification*. RTCA, 1992.

[14] https://sigtest.dev.java.net/.

APPENDIX

A. GETTING THE TCK

The source code of the TCK can be downloaded from `http://ovmj.org/`.

B. RUNNING THE TCK

Our TCK is arranged into several different directories:

- `bin/` Java class files
- `doc/` Documentation files (readme, etc.)
- `lib/` JNI C library sources
- `properties/` Property files for testcases
- `RICompl/` Complementary classes to the RI
- `src/` Contains `s3scj` Java package (TCK)

For testing, we used the Sun Real-Time Virtual Machine. Compilation and execution of the TCK may vary between different virtual machines, due to different command-line parameters. For example, IBM's J9 compiler requires the `-Xrealtime` flag for compiling and running RTSJ programs. A Makefile is included in the root directory that compiles all of the TCK classes. It compiles the RI and subsequently the TCK against the RI. It assumes that the `javac` in the executable path is that of a real-time virtual machine with no necessary command-line parameters. If this is the case, simply typing `make` should successfully build the TCK, placing the class files in the `lib` directory.

To run the TCK, a `tckrun.sh` script is provided in the root directory. Once again, it assumes that `java` is a real-time virtual machine. Manually running a testcase simply involves using `javax.safetycritical.S3Launcher` as the main class and passing the class name and property file as arguments. For example, to run the `TestSchedule400` testcase:

```
$ java -cp bin javax.safetycritical.S3Launcher \
    s3scj.tck.TestSchedule400 \
    properties/tck/TestSchedule400.prop
```

EMMA, the coverage tool we used, is not included in the download, but is open source and easily accessible. The steps to using EMMA are:

1. Instrument the TCK and RI class files with EMMA. The following instruments the class files in `bin` and outputs them in `instrbin`.

   ```
   $ java -cp emma.jar emma instr -d instrbin -ip bin
   ```

2. Run the TCK normally, except add `emma.jar` to the classpath.

   ```
   $ java -cp instrbin:emma.jar \
       javax.safetycritical.S3Launcher \
       s3scj.tck.TestSchedule400 \
       properties/tck/TestSchedule400.prop
   ```

3. Generate a text and HTML report with EMMA. The `-sp` option gives EMMA a source path for highlighting coverage of blocks.

   ```
   $ java -cp emma.jar emma report -r txt,html \
       -in coverage.em,coverage.ec -sp <RISRCPATH>
   ```

C. FAILURE MESSAGES

All failure messages that the TCK can report are summarized in Table 4. Some messages contain execution dependent information, which represented by <...>.

Table 1: Detected Failure

Default ceiling should equal to the max priority
javax.realtime.ThrowBoundaryError is thrown (javax.safetycritical.ThrowBoundaryError expected)
Private memory should not nest on Level 0
Private memory not entered from its parent scope
Multiple handlers entered a private memory simultaneously (scope cycle)
Self-suspension illegally allowed in synchronized code on Level 0 and 1
Error occurred in javax.safetycritical.System.sleep(RelativeTime)
Error occurred in javax.safetycritical.System.sleep(RelativeTime, boolean true)
Error occurred in javax.safetycritical.System.sleep(RelativeTime, boolean false)
Error occurred in javax.safetycritical.System.sleep family (threads not sleep)

Table 2: Coverage Summary for Package

name	class, %	method, %	block, %	line, %
javax.safetycritical	97% (28/29)	88% (118/134)	87% % (1413/1621)	89% (378.6/426)

Table 3: Coverage Breakdown by Source File

name	class, %	method, %	block, %	line, %
ThrowBoundaryError.java	0% (0/1)	0% (0/5)	0% (0/11)	0% (0/6)
ExternalEvent.java	100% (1/1)	75% (3/4)	74% (35/47)	76% (13/17)
InterruptHandler.java	100% (1/1)	60% (3/5)	75% (6/8)	67% (4/6)
System.java	100% (1/1)	89% (8/9)	77% (10/13)	89% (8/9)
LevelOneManager.java	100% (1/1)	100% (6/6)	77% (139/180)	69% (31.8/46)
MissionMemory.java	100% (2/2)	88% (21/24)	81% (192/236)	87% (54.6/63)
Mission.java	100% (2/2)	83% (10/12)	83% (158/190)	96% (42/44)
MissionSequencer.java	100% (1/1)	100% (4/4)	86% (93/108)	89% (25/28)
MissionManager.java	100% (1/1)	100% (10/10)	88% (85/97)	96% (26/27)
PrivateMemory.java	100% (1/1)	83% (5/6)	88% (124/141)	95% (26.6/28)
ManagedEventHandler.java	100% (2/2)	83% (5/6)	91% (86/94)	92% (22/24)
LevelZeroManager.java	100% (1/1)	100% (5/5)	93% (136/147)	92% (29.6/32)
AperiodicEvent.java	100% (1/1)	100% (3/3)	100% (31/31)	100% (9/9)
AperiodicEventHandler.java	100% (1/1)	100% (2/2)	100% (16/16)	100% (4/4)
AperiodicReleaseParameters.java	100% (1/1)	100% (1/1)	100% (7/7)	100% (2/2)
CyclicExecutive.java	100% (1/1)	100% (2/2)	100% (19/19)	100% (5/5)
CyclicSchedule.java	100% (2/2)	100% (7/7)	100% (58/58)	100% (16/16)
LevelTwoManager.java	100% (1/1)	100% (2/2)	100% (12/12)	100% (5/5)
MissionMemoryStore.java	100% (1/1)	100% (4/4)	100% (20/20)	100% (8/8)
NoHeapRealtimeThread.java	100% (1/1)	100% (3/3)	100% (24/24)	100% (6/6)
PeriodicEventHandler.java	100% (1/1)	100% (2/2)	100% (44/44)	100% (10/10)
PortalExtender.java	100% (1/1)	100% (1/1)	100% (6/6)	100% (2/2)
SingleMissionSequencer.java	100% (1/1)	100% (3/3)	100% (13/13)	100% (5/5)
Terminal.java	100% (1/1)	100% (7/7)	100% (96/96)	100% (23/23)
ThreadConfigurationParameters.java	100% (1/1)	100% (1/1)	100% (3/3)	100% (1/1)

Table 4: Failure Message List

Mission Life Cycle
Failed to launch all missions
Mission timelines overlap
Frames not executed sequentially
Error occurred in nested NHTR
Error occurred in nested PEH
Error occurred in nested AEH
Memory
PrivateMemory illegally created in <memory area name>
Mission memory is not instance of ScopedMemory
Schedulable objects not run in private memory
Private memory is not instance of ScopedMemory
Private memory is not instance of LTMemory
Error occurred in PrivateMemory.getManager()
Unable to enter nested private memory
Error occurred during object creation in mission memory during mission phase
Error occurred during object creation in immortal memory during mission phase
Private memory not entered from its parent scope
Mission global object inaccessible or incorrect
Multiple handlers entered a private memory simultaneously (scope cycle)
Failure in resizing mission memory size - Msg: <message>
Clock, Timer, Time
The real-time clock should be monotonic and non-decreasing
HighResolutionTime.waitForObject() should be allowed on level 2
Schedule
AEH should be non-daemon
AEH should be non-heap
PEH should be non-daemon
PEH should be non-heap
NHRT should be non-daemon
NHRT should be non-heap
Error occurred in AEH or PEH
Error occurred in nested synchronization
More than one realtime server thread running
Private memory should not nest on Level 0
Default ceiling should equal to the max priority
Scheduler should provide at least 28 priorities
Requesting a lock (via the synchronized method) should not be considered self-suspension
Level 0 execution illegally preempted
Error in priority ceiling emulation
Error in preemptive scheduling
Thread being preempted should be placed in front of the run queue
Error occurred in calling Thread.join() - Msg: <message>
Self-suspension illegally allowed in synchronized code on Level 0 and 1
Self-suspension should be allowed on Level 2 - Msg: <message>
Error occurred in wait/notify/notifyAll
“null” illegally accepted as PriorityParameters
“null” illegally accepted as ReleaseParameters
Negative memory size illegally accepted
Error occurred in ExternalEvent (single handler)
Error occurred in ExternalEvent (multiple handlers)
Error occurred in InterruptHandler

Table 4: Failure Message List (continued)

Exception
ThrowBoundaryError should be pre-allocated
Error in ThrowBoundaryError.getPropagatedExceptionClass()
Error in ThrowBoundaryError.getPropagatedMessage()
Error in ThrowBoundaryError.getPropagatedStackTraceDepth()
Error in ThrowBoundaryError.getPropagatedStackTrace()
javax.realtime.ThrowBoundaryError is thrown (javax.safetycritical.ThrowBoundaryError expected)
JNI
Failure in JNI test: Object information
Failure in JNI test: String
Failure in JNI test: Array
Failure in JNI test: NIO
Misc
Error occurred in javax.safetycritical.System.sleep(RelativeTime)
Error occurred in javax.safetycritical.System.sleep(RelativeTime, boolean true)
Error occurred in javax.safetycritical.System.sleep(RelativeTime, boolean false)
Error occurred in javax.safetycritical.System.sleepNonInterruptable(RelativeTime)
Error occurred in javax.safetycritical.System.sleepNonInterruptable(RelativeTime, boolean true)
Error occurred in javax.safetycritical.System.sleepNonInterruptable(RelativeTime, boolean false)
Error occurred in javax.safetycritical.System.sleep() family (threads not sleep)
Time out in TestMisc