Safety-Critical Java Virtual Machine Services

James Baxter
University of York, UK
jeb531@york.ac.uk

Ana Cavalcanti
University of York, UK
ana.cavalcanti@york.ac.uk

Andy Wellings
University of York, UK
andy.wellings@york.ac.uk

Leo Freitas
Newcastle University, UK
leo.freitas@ncl.ac.uk

ABSTRACT

To ensure that Safety-Critical Java (SCJ) programs run correctly and safely, the virtual machine they run on must be shown to operate correctly. To the best of our knowledge, however, currently we do not even have a clear specification of the requirements for such a virtual machine. In this paper, we present an identification of these requirements for the SCJ API and infrastructure, based on the requirements of the SCJ standard and on consideration of existing virtual machines for SCJ. Formal methods provide a powerful tool in modelling and eliciting requirements, and establishing correctness of implementations. We also present here a formal model of the requirements written in the *Circus* specification language, which has already been used in a technique for verification of SCJ programs. Our work is a contribution to establishing a framework for the development of fully verified systems using SCJ.

Categories and Subject Descriptors

C.3 [**Computer Systems Organization**]: Special-purpose and Application-based Systems—*real-time and embedded systems*; D.3.4 [**Programming Languages**]: Processors—*runtime environments, Safety-Critical Java*

General Terms

Design, Verification

1. INTRODUCTION

The development of the Safety-Critical Java (SCJ) specification [17] has been motivated by the need to create certifiable real-time applications in Java. While the SCJ specification covers all aspects of the SCJ API and infrastructure, there is no explicit account of requirements for an SCJ virtual machine (SCJVM). Yet, the underlying virtual machine must also be correct in order to fulfil properly the certifiability requirements of safety-critical systems. Even in systems that do not have a full virtual machine there must still be some runtime environment to support the running of the SCJ bytecode that is required to be correct.

There is already an accepted specification of the standard Java Virtual Machine (JVM) [15] with widely used implementations. The JVM provides an environment for the execution of multithreaded Java programs that have been compiled into Java bytecode. The environment consists of three main components: a bytecode interpreter, a memory manager that supports a garbage-collected heap (including the method area), and a scheduler that shares the underlying processing resources amongst the threads.

Work has already been done on the semantics of Java bytecode and verification of standard JVMs [4, 12, 25]. However, SCJ has a number of differences from standard Java. Firstly, the SCJ memory model, abandons the garbage collector in favour of a scoped memory model. Garbage collection is less predictable and often quite complex, and so unsuitable for some of the strictest certifiability requirements of safety-critical systems. By contrast, the scoped memory model provides greater predictability on when memory is freed as it can be determined when a given scope is left and triggers deallocation. The SCJ approach to scheduling differs from that of standard Java, using a preemptive priority scheduling approach rather than the loosely specified priority scheduling system used for Java threads. These differences mean that the standard JVM is unsuitable for running SCJ programs. A specialised virtual machine is required.

Various virtual machines have been created that can run SCJ programs, including Fiji VM [21], icecap HVM [24], OVM [1], HVM_{TP} [18] and PERC Pico [2, 22]. The Fiji VM and icecap HVM both take the approach of precompiling Java bytecode to C in order to allow for faster running programs that use fewer memory resources. Fiji VM is, however, not specifically concerned with SCJ; it is rather designed to run general real-time Java programs.

By contrast, icecap has been designed specifically to run SCJ programs. It includes an implementation of the SCJ libraries, although they cannot be easily decoupled from the virtual machine itself. In addition, icecap also provides a lightweight Java bytecode interpreter and allows for interpreted code to be mixed with compiled code. HVM_{TP} is a modification of the icecap HVM's bytecode interpreter to improve time predictability and ensure that bytecode instructions are executed in constant time.

OVM follows the approach of precompiling code for performance reasons, similar to Fiji VM and icecap HVM, but translates Java to C++ instead of bytecode to C. OVM predates SCJ; it is written to implement the Real-Time Specifi-

cation for Java (RTSJ) [10], though it can still support SCJ programs as SCJ is based on a subset of RTSJ.

PERC Pico is a product of Atego based on early ideas for SCJ, but uses its own system of Java metadata annotations to ensure the safety of scoped memory. PERC Pico does not support the current SCJ standard.

In summary, as far as we know, there is one implementation of an SCJVM publicly available: icecap.

It is and, typically, virtual machines for SCJ will be, designed to be very small and fast so as to be able to run on embedded systems. While these are important goals in an embedded virtual machine implementation, the resultant virtual machines tend to be complex and so hard to verify. There does not yet appear to be any work on verification of an SCJVM. Moreover, not even the requirements of an SCJVM, which must be clearly identified before verification can commence, are currently agreed on.

In addition to providing a reference to verify against, a specification of the requirements for an SCJVM also allows for greater portability of SCJ infrastructure implementations. With an accepted specification of the services to be provided by an SCJVM, the SCJ infrastructure can be developed without specific concerns about target operating systems and platforms. Separating the requirements of an SCJVM from the requirements of the SCJ infrastructure also potentially allows for greater reusability, reducing the burden of proving the correctness of different implementations of SCJ. There are also benefits in that the requirements of the underlying operating system can be identified. In this way, the minimal necessary features of the operating system can be implemented as part of a hardware abstraction layer, rather than a fully fledged operating system. This may be important given the resource constraints of embedded systems.

In this paper, we present a detailed specification for an SCJVM. We identify the main components and describe the API that they should provide to support the implementation of the SCJ infrastructure and of other components.

We have designed and specified our API by inspecting the SCJ standard and extracting the implicit requirements imposed on an SCJVM by the requirements on the API and infrastructure. The requirements have also been validated and completed by considering the icecap implementation. There has already been much research into the semantics of Java bytecode [4, 12, 25] and bytecode for SCJ does not greatly differ from that of standard Java [15]. So we have focused on the services required by an SCJVM.

The SCJ specification defines three compliance levels to which programs and implementations may conform. Level 0 is the simplest compliance level. It is intended for programs following a cyclic executive approach. Level 1 lifts several of the restrictions of level 0, allowing handlers that may trigger in response to external events and preempt one another. Level 2 is the most complex compliance level, allowing access to real-time threads and suspension via `wait()` and `notify()`. There is almost no difference in the virtual machine requirements for each level, so the virtual machine presented here supports level 2 programs, although we do not consider the multiprocessor facilities available at level 2.

In characterising the SCJVM, it is useful to have the specification in some formal notation. The development of a formal specification is challenging as the formal notation is more precise than the informal text that is found in the SCJ specification. On the other hand, it provides several benefits. Firstly, it gives precise guidance for those implementing an SCJVM and the SCJ API. Moreover, the effort to extract a formal specification from the implicit requirements of the SCJ specification has had the benefit of uncovering areas where it is insufficiently specified. Here, we briefly describe a formal specification of the virtual machine services in the *Circus* specification language [20].

In summary, our contributions are a specification of an API for an SCJVM, together with a formal model. This specification identifies the lower-level services that should be provided by an SCJVM to permit portable SCJ implementations to be written. The provision of the formal model states the requirements more precisely and facilitates proof of correctness of an SCJVM.

The structure of this paper is as follows. Section 2 presents an overview of Safety-Critical Java and its differences from standard Java, giving a more detailed explanation of why a specialised virtual machine is needed for SCJ. Section 3 contains the informal specification of the SCJVM services that we have identified. An overview of *Circus* and the formal model is then presented in Section 4. Finally, Section 5 discusses our conclusions, the relation of this work to similar work, and directions of future work.

2. SAFETY-CRITICAL JAVA

Safety-Critical Java is a variant of Java designed for writing programs for which certifiability is an important concern. SCJ is based on the Real-Time Specification for Java, which augments the standard Java scheduler and garbage-collected heap with more predictable priority-scheduled event handlers and scoped memory areas. The modification that SCJ makes to the RTSJ is to remove the aspects of RTSJ that make certification difficult, including standard Java threads and the garbage collector. This leads to scheduling and memory management models that are very different to the models for standard Java and that, therefore, require specialised virtual machines to support them.

An SCJ program consists of one or more missions, which are collections of schedulable objects that are scheduled by SCJ's priority scheduler. Missions are run in an order determined by the mission sequencer supplied by an SCJ program. Running a mission consists of starting each of the schedulable objects in the mission, waiting for a request to terminate the mission, then terminating each of the schedulable objects in the mission. The schedulable objects within a mission are event handlers that are released either periodically, at set intervals of time, aperiodically, in response to a software release request, or once at a specific point in time (though handlers that are released once can have a new release time set, allowing them to be released again).

Each schedulable object has a priority and the highest priority object that is eligible to run at each point in time is the object that runs. This allows for simpler reasoning about order of execution of schedulable objects and allows for more urgent tasks to preempt less urgent tasks.

SCJ allows for assigning schedulable objects to "scheduling allocation domains", consisting of one or more processors. At level 1, each domain is restricted to a single processor. Hence, in scheduling terms, the system is fully partitioned. This allows for mature single processor schedulability analysis to be applied to each domain (although the calculation of the blocking times when accessing global syn-

chronised methods is different than on a single processor system, due to the potential for remote blocking [7]).

SCJ deals with memory in terms of memory areas, which are Java objects that provide an interface to blocks of physical memory called backing stores. Memory allocations in SCJ are performed in the backing store of the memory area designated as the current allocation context. Each schedulable object has a memory area associated with it that is used as the default allocation context during a release of that object, and is cleared after each release. Each mission also has a mission memory area that can be used as an allocation context by the schedulable objects of that mission, to provide space for objects that persist for the duration of the mission or that need to be shared between schedulable objects. There is also an immortal memory area where objects can be allocated if they are needed for the entire running of the program (they are never freed).

This system of memory areas makes it easy to predict when memory is freed, which cannot be easily done with a garbage-collected heap. Furthermore, this memory model cannot be achieved with a standard JVM as it does not provide memory outside of the heap for allocation and lacks a notion of allocation context. The memory manager also needs to provide a means of accessing raw memory for the purposes of device access, but that section of the SCJ standard is not yet finalised so we will not cover it here. It can, however, be seen that any system of raw memory access is not supported by most standard JVMs.

A further requirement of SCJ is that dynamic class loading is not allowed; all classes used by the program must be loaded when the program starts. This is because dynamic class loading may introduce time overheads that are hard to predict and additional code paths that complicate certification. SCJ also disallows object finalisers as it is not always easy to predict when they are run.

Because of these features of SCJ, a specialised virtual machine that provides support for allocation in memory areas and preemptive scheduling is required for SCJ.

3. SERVICES OF AN SCJVM

Figure 1 shows the components that are required of an SCJVM. The core execution environment handles the running of Java bytecode. The SCJVM services support, for example, scheduling and memory management, that is, services that are required to support the SCJ infrastructure and the core execution environment.

The core execution environment is not required to use any particular means to run the bytecode. It may interpret bytecode instructions, just-in-time compile the bytecode, or execute native code precompiled from Java bytecode (in which case the compiler is regarded as part of the core execution environment). As mentioned previously, we do not go into detail about the core execution environment here. Its specification is basically the semantics of Java bytecode, which is mostly the same for SCJ. It should also be noted that class loading is treated as part of the core execution environment, and so it is bound by the requirement in the SCJ standard to load all classes at virtual machine startup.

As can be seen from Figure 1, we group the SCJVM services to characterise three main components:

- the memory manager, which manages backing stores for memory areas and allocation within them;
- the scheduler, which manages threads and interrupts that allow for implementation of SCJ event handlers; and
- the real-time clock, which provides an interface to the system real-time clock.

Each of these services is used either by the core execution environment or by the SCJ infrastructure; some of the services also rely on each other. For example, the scheduler must update the allocation context in the memory manager when performing a thread switch.

The SCJVM services make use of low-level operating system services to access hardware. These operating system services may be supplied by a full-featured operating system on which the SCJVM runs or may simply be a minimal set of services required to run the virtual machine. The latter case is desirable for low-end embedded systems that lack the resources to run a general operating system.

3.1 Memory Manager API

The SCJVM memory manager deals with the raw blocks of memory used as backing stores for the memory areas of SCJ. The memory areas themselves are Java objects, and so are dealt with by the core execution environment and accessed through the SCJ API, instead of directly via the virtual machine. This is in line with what is specified in the SCJ standard and also done for RTSJ. Backing stores are assumed to have unique identifiers that can be used to refer to them; these identifiers can be simply pointers to the physical blocks of memory used for backing stores.

There is initially one backing store, called the root backing store, which has its size set when the SCJVM starts up to cover all the memory available for allocation in backing stores. The root backing store is not allowed to be resized or destroyed, so that there is always a fixed base for the layout of memory. The root backing store is intended to be used as the backing store for the immortal memory area. A backing store may have other backing stores nested within it, so that a possible memory layout is as shown in Figure 2. In this example, the backing store of the mission memory is nested within the root backing store, and the backing stores for the per-release memory of each schedulable object in a mission is nested within the mission memory's backing store.

The operations of the memory manager API are summarised in Table 1. In addition to the inputs and outputs described there, there should also be some system of reporting erroneous inputs, whether that be exceptions, global error flags or particular return values signalling errors. The conditions that cause an error to be reported are listed.

The root backing store described above is always available to the SCJ infrastructure through the `getRootBackingStore` operation. An SCJ program, on the other hand, does not have direct access to the root backing store except through memory areas provided by the infrastructure.

In addition to managing the layout of backing stores in general, the memory manager must track the current allocation context. The operations `getCurrentAllocationContext` and `setCurrentAllocationContext` provide a means to get and set the current allocation context. The functionality of changing the allocation context is used by the methods of the SCJ API that allow code to be run with a different memory area as allocation context, such as `executeInAreaOf()` and `enterPrivateMemory()`. The memory man-

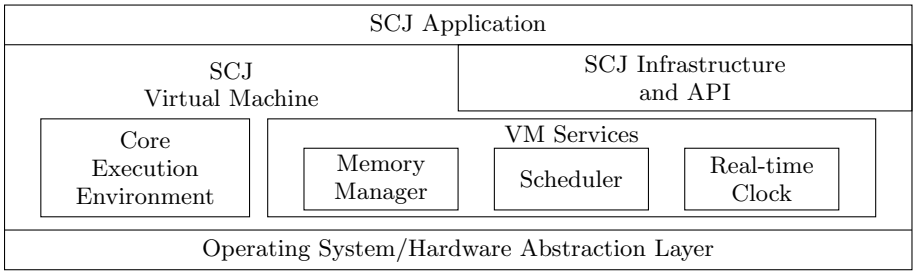

Figure 1: Structure of the SCJVM and its relation to the SCJ infrastructure and the operating system/hardware abstraction layer.

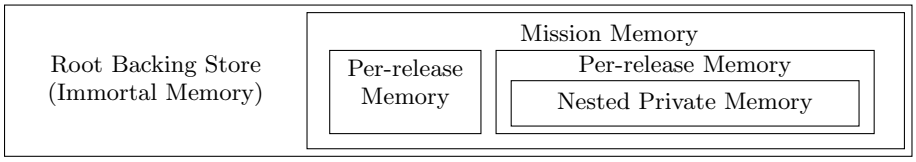

Figure 2: An example memory layout

ager stores each thread's allocation context and queries the scheduler as to which thread is current when it performs operations affecting the current allocation context.

It is possible to obtain information about the used and available space in a given backing store using the operations `getTotalSize`, `getUsedSize`, and `getFreeSize`. This information is made available to SCJ programs through the interface provided by memory areas defined in the infrastructure.

Another query that can be made concerning backing stores is that of which backing store a particular memory address lies in. This information can be obtained by the `findBackingStore` operation and is required by the infrastructure for obtaining the memory area of a given object.

Allocation within backing stores is possible through the `allocateMemory` operation, which allocates blocks of memory within the current allocation context. This operation is provided in order for the core execution environment to implement the `new` bytecode instruction and should not be directly available to the program or infrastructure. Allocations within backing stores must not cause fragmentation, so as to fulfil real-time predictability requirements. The operation `allocateMemory` must also zero the memory it allocates, in order to match the semantics of `new`.

Allocation of backing stores is provided by `makeBackingStore`, which is available to the infrastructure for use when creating new memory areas. A new backing store is created nested within the specified backing store. The infrastructure is responsible for storing the backing store identifier returned by `makeBackingStore`. Backing store allocation must be done in constant time without fragmentation.

Deallocation of memory in backing stores cannot be done directly as that could introduce fragmentation and would defeat the scoped memory model of SCJ. Instead, the SCJVM provides for clearing a backing store when the memory area it serves is no longer in use. This functionality is provided by the operation `clearBackingStore`, which clears the specified backing store, deallocating all objects and nested backing stores within it. It is not necessary to track exactly which objects are deallocated by this operation as SCJ does not have object finalisers. The clearing of a backing store includes the clearing of all backing stores nested within it. Since a backing store is not necessarily aware of what backing stores are nested within it, its nested backing stores' memory is freed with the rest of the backing store. This would create a problem if the parent backing store is cleared while another thread is using a backing store within it as an allocation context. However, such a situation should not occur as the backing stores of mission memory and immortal memory are the only backing stores that contain backing stores in use by different threads. The mission memory is only cleared when all the event handler threads within the mission have finished and the immortal memory should never be cleared. This is handled as an error case, since it cannot be guaranteed in general and relies on the API implementation functioning correctly.

The last operation on backing stores is their resizing. This is provided for by `resizeBackingStore` but, as resizing a backing store presents a lot of difficulties in terms of fragmentation, there are several restrictions. In addition to being a valid backing store and there being enough space in the parent backing store for the resizing to take place, a backing store to be resized must not be the root backing store, and must be empty and the only backing store within its parent. However, this operation should only be needed for resizing of the mission memory in between missions and resizing of a nested private memory when it is reentered. In both these cases all the needed restrictions hold.

The memory manager must also manage stacks, which are placed in a separate area of memory to the backing stores. The operations `createStack` and `destroyStack` allow for stacks to be created and destroyed. The stack space must not be fragmented, which is a requirement that can be met since stacks for threads are allocated together when a mission is initialised and destroyed together when the mission ends. That remains true at level 2 where nested missions are permitted, since the nested mission's stacks are allocated after the stacks of its parent mission and are destroyed before the parent mission ends. Like backing stores, stacks are referred to by unique identifiers that may simply be pointers

Operation	Inputs	Outputs	Error Conditions
`getRootBackingStore`	(none)	backing store identifier	(none)
`getCurrentAllocationContext`	(none)	backing store identifier	no current thread allocation context
`setCurrentAllocationContext`	backing store identifier	(none)	invalid identifier no current thread allocation context
`getTotalSize`	backing store identifier	size in bytes	invalid identifier
`getUsedSize`	backing store identifier	size in bytes	invalid identifier
`getFreeSize`	backing store identifier	size in bytes	invalid identifier
`findBackingStore`	memory pointer	backing store identifier	no backing store found
`allocateMemory`	size in bytes	memory pointer	insufficient free memory no current thread allocation context
`makeBackingStore`	backing store identifier size in bytes	backing store identifier	invalid identifier insufficient free memory no current thread allocation context
`clearBackingStore`	backing store identifier	(none)	nested backing store in use no current thread allocation context
`resizeBackingStore`	backing store identifier size in bytes	(none)	invalid identifier backing store in use backing store is root backing store not empty backing store not only child insufficient free space no space for memory overhead
`createStack`	size in bytes	stack identifier	insufficient free space
`destroyStack`	stack identifier	(none)	invalid identifier stack space fragmentation

Table 1: The operations of the SCJVM memory manager

to the space allocated for the stack.

3.2 Scheduler API

The SCJVM scheduler manages the scheduling of threads, which are abstract lines of execution, each with its own stack and current allocation context. These threads should be used to implement the event handlers of SCJ, with each event handler being bound to a single thread. The operations of the scheduler are summarised in Table 2.

Each thread is scheduled according to a priority level. The SCJ standard requires that there be at least 28 priorities and separates them into hardware and software priorities, with hardware priorities being higher than software priorities. The range of priorities that an SCJVM actually supports may vary between different implementations within these restrictions. To allow the range of supported priorities to be determined and support corresponding methods in the SCJ API, the minimum and maximum hardware and software priority levels can be obtained with `getMaxSoftwarePriority`, `getMinSoftwarePriority`, `getMaxHardwarePriority`, and `getMinHardwarePriority`. The SCJVM chooses a default normal software priority for threads, that can be queried through the `getNormSoftwarePriority` operation.

Initially there is one thread running, which is called the main thread. The main thread is created when the SCJVM starts and has an implementation-defined priority. The main thread can be suspended by the infrastructure when it is not needed and resumed when it is needed again (using operations described in the sequel). This allows it to be used for setting up the SCJ application and missions, then suspended during mission execution. The main thread's identifier can be retrieved using the `getMainThread` operation.

Threads other than the main thread can be created by the `makeThread` operation, which takes the entry point and priority level of the thread to be created, as well as a backing store as the allocation context and a stack. This operation returns the identifier of the newly created thread, which must be stored by the infrastructure. The SCJVM does not distinguish between the different thread-release conditions, so for periodic and one-shot threads the infrastructure must set a timer separately using the real-time clock API when a thread is created. The only priorities allowed for threads are the software priorities, as hardware priorities are reserved for interrupts. The backing store supplied is only used to set the backing store in the memory manager when the thread starts and is not stored by the scheduler.

The SCJVM threads that are eligible to run must be scheduled as if they are placed in queues with one queue for each priority. At each moment in time, the thread at the front of the highest priority non-empty queue is running. A thread becomes eligible to run after it is started, and stops being eligible to run when it is blocked. A thread is started using the `startThread` operation and must be started by the infrastructure when its enclosing mission starts. The reason for the separation between thread creation and thread starting is to ensure that threads all start together after mission initialisation has been completely finished.

The identifier of the currently running thread can be obtained through `getCurrentThread`. This operation may be used by the infrastructure as part of obtaining the current schedulable object, but is mainly intended for use by the memory manager to discern the current allocation context.

A thread can suspend itself, causing it to become blocked, and be resumed on command from another thread, causing it to become eligible to run again, by the operations `suspendThread` and `resumeThread`. A thread must not be holding any locks when it suspends. These operations are only visible to the program through `wait()` and `notify()` at level 2. These operations are also used in hardware communication, when a thread must wait for the hardware to complete a request, and to implement thread release, whereby a thread remains suspended until released.

A thread that has been created can then be destroyed with the `destroyThread` operation, which removes the thread

Operation	Inputs	Outputs	Error Conditions
`getMaxSoftwarePriority`	(none)	priority level	(none)
`getMinSoftwarePriority`	(none)	priority level	(none)
`getNormSoftwarePriority`	(none)	priority level	(none)
`getMaxHardwarePriority`	(none)	priority level	(none)
`getMinHardwarePriority`	(none)	priority level	(none)
`getMainThread`	(none)	thread identifier	(none)
`makeThread`	entry point priority level backing store identifier stack identifier	thread identifier	(none)
`startThread`	thread identifier	(none)	invalid identifier thread already started
`getCurrentThread`	(none)	thread identifier	(none)
`destroyThread`	thread identifier	(none)	invalid identifier thread not destroyable
`suspendThread`	(none)	(none)	thread cannot be blocked thread holds locks
`resumeThread`	thread identifier	(none)	invalid identifier thread not blocked
`setPriorityCeiling`	pointer to object priority level	(none)	invalid priority
`takeLock`	pointer to object	(none)	lock in use
`releaseLock`	pointer to object	(none)	lock not held
`attachInterruptHandler`	interrupt identifier entry point	(none)	(none)
`detachInterruptHandler`	interrupt identifier	(none)	(none)
`getInterruptPriority`	interrupt identifier	priority level	(none)
`disableInterrupts`	(none)	(none)	(none)
`enableInterrupts`	(none)	(none)	(none)

Table 2: The operations of the SCJVM scheduler

from the scheduler. Destroying a thread does not automatically destroy its stack or the backing store being used as its allocation context. The SCJ infrastructure should not destroy a thread while it is running as a thread should only be destroyed when the mission it is part of is ending. The infrastructure should instead ensure that all threads in a mission are suspended before destroying them.

The SCJVM must support priority ceiling emulation. This is handled by the `setPriorityCeiling` operation that associates a priority ceiling value to an object. An object that does not have its priority ceiling explicitly set has a priority ceiling equal to the default ceiling. This should be the highest software priority, but it is possible for an SCJVM to have an option to change the default priority ceiling. The SCJVM scheduler does not require a notion of object in order to associate priority ceilings to objects since an object's pointer can be used as an opaque identifier.

The operations for taking and releasing locks are `takeLock` and `releaseLock`. A thread can only take a lock if its active priority and the ceiling priorities of any other objects it holds the locks for are less than or equal to the ceiling priority of the object the lock is being taken on. Only one thread can take a given object's lock at a time. When a lock is taken, the thread's active priority is raised to the object's priority ceiling. When a thread releases a lock, the thread's active priority is lowered to its previous active priority. The thread may hold nested locks on multiple objects.

The SCJVM scheduler must also manage interrupts, as interrupt handlers have priorities (though these should be in the hardware priority range) and so must be dealt with within the priority scheduling model. An interrupt handler can be attached to a given interrupt using the `attachInterruptHandler` operation, and an interrupt's handler can be removed with the `detachInterruptHandler` operation. An interrupt with no handler attached to it is ignored. The clock interrupt coming from the hardware is handled by the SCJVM clock (see Section 3.3) and converted into a clock interrupt that is passed to the scheduler for handling by the attached interrupt handler (which should simply call the `triggerAlarm()` method of `Clock`).

Each interrupt has a priority associated with it, which is set by the SCJVM on startup and cannot be changed by the application. These interrupt priorities must be hardware priorities. An interrupt handler interrupts any lower-priority interrupt handler and any running threads, and blocks lower-priority interrupt handlers from running until it has finished. The priority associated with each interrupt can be obtained by the `getInterruptPriority` operation.

Interrupts can be disabled using the `disableInterrupts` operation and re-enabled using the `enableInterrupts` operation. While interrupts are disabled no interrupt handlers can run, but it is implementation-defined as to whether or not interrupts fired while interrupts are disabled are lost.

3.3 Real-time Clock API

The SCJVM must manage the system real-time clock, providing an interface that allows for the time to be read and alarms to be set to trigger time-based events. The operations of the SCJVM real-time clock are summarised in Table 3.

The main function of the real-time clock API is to provide access to the system time through the `getSystemTime` operation. The SCJ API deals with time values in terms of milliseconds-nanoseconds pairs. That should also be the format for time values passed to and from the SCJVM though another format could be used. The system time may be measured from January 1, 1970 or from the system start time

Operation	Inputs	Outputs	Error Conditions
`getSystemTime`	(none)	time	(none)
`getSystemTimePrecision`	(none)	time precision	(none)
`setAlarm`	time	(none)	time in past
`clearAlarm`	(none)	(none)	(none)

Table 3: The operations of the SCJVM real-time clock

(in case there is no reliable means of determining the date and time), and so may not correspond to wall-clock time.

The time between ticks of the system clock (its precision) must be made available through the `getSystemTimePrecision` operation. The clock's precision must not change.

The SCJVM must also provide a facility to set an alarm that sends a clock interrupt to the scheduler when a specific time is reached. This facility is provided by the `setAlarm` operation, which accepts an absolute time value at which the alarm should trigger. The time passed to `setAlarm` is required to not be in the past. Running code at a specified relative time offset should be handled by the infrastructure. Once an alarm has triggered, it is removed and a new alarm must be set in order to perform events periodically.

The current alarm (if any) can be cleared using the `clearAlarm` operation. Attempting to clear the alarm when there is no alarm set does nothing.

4. FORMAL MODEL

The formal model of the SCJVM is written in the *Circus* specification language [20]. *Circus* is based on CSP [23], which is used to specify processes that communicate over channels, and the Z notation [27], which is used to specify state and data operations. In this section, we present a brief explanation of *Circus* and an overview of our model. We then present part of the memory manager model. The complete model can be found in [3]. It is type checked with Community Z Tools and we have started to prove some basic properties of the memory manager using Z/Eves.

A *Circus* specification is made up of processes that communicate over channels. These channels may carry values of a particular type, or may be used as flags for synchronisation or signalling between processes. Each process may have state, and is made up of actions that operate on that state and communicate over channels.

In our model, each of the components of the SCJVM shown in Figure 1 is specified by a single process whose channels represent the services they provide. The whole collection of VM services are then specified by a parallel composition of the memory manager, scheduler and clock processes. In this case, parallelism is used to define a conjunction of requirements: those for each of the components.

The processes synchronise on the channels they share, specified in the sets *MMSInterface* and *RTCSInterface*. The set *MMSInterface* is the interface between the memory manager and the scheduler, which contains a channel to get the current thread from the scheduler, and channels to inform the memory manager of the creation and destruction of threads. The interface between the real-time clock and the scheduler, *RTCSInterface*, contains a channel to pass the clock interrupt to the scheduler for handling. The channels in the set *VMServicesInternals* are used for communication between the SCJVM components and are hidden. So, the only channels that can be used to communicate with the SCJVM services are those representing the services in Tables 1, 2 and 3, and those used for communication with the core execution environment.

$$
\begin{aligned}
&VMServices \mathrel{\widehat=} \\
&\quad ((MemoryManager \llbracket MMSInterface \rrbracket Scheduler) \\
&\qquad \llbracket RTCSInterface \rrbracket RealtimeClock) \\
&\qquad\quad \setminus VMServicesInternals
\end{aligned}
$$

The parallel composition of *VMServices* with a process representing the core execution environment (which, as mentioned, we do not specify here) specifies the full SCJVM.

Our model of the scheduler is similar to other formal models of priority schedulers [9, 11, 13, 14], and the real-time clock specification is fairly simple and just manages the current time and any alarm that may be set. We, therefore, do not detail the scheduler and clock models. Instead, we present part of the memory manager model. We show the memory manager state and some of the operations.

The memory manager specification begins by declaring a type, *MemoryAddress*, of memory addresses to be the set of natural numbers. This then allows for specification of a type, *ContiguousMemory*, that contains contiguous ranges of memory addresses and is used to specify that backing stores must not be fragmented.

$$
\begin{aligned}
&MemoryAddress == \mathbb{N} \\
&ContiguousMemory == \\
&\quad \{ m : \mathbb{P}\, MemoryAddress \mid \\
&\qquad \exists a, b : MemoryAddress \bullet m = a \mathinner{..} b \}
\end{aligned}
$$

Backing stores are identified by the elements of the type *BackingStoreID*. These are opaque identifiers and there are no constraints on *BackingStoreID*.

To specify backing stores, we first define a notion of memory block. The schema *MemoryBlock* represents a memory block via a record that stores the used, free and total memory. This aspect of backing stores is separated out because it is also used in specifying the stack space. The variables *used* and *free* correspond to the areas of used and free memory, while *total* represents the whole memory area covered by the *MemoryBlock*. We note that *free* and *total* are required to be contiguous to enforce the requirement that there must be no fragmentation whereas *used* is simply specified to be a set of memory addresses. There are two invariants on *MemoryBlock*, identified by the predicates in the schema below. The first invariant simply specifies that *used* and *free* must be contained in *total*, but it does not require them to cover *total* as there may be additional memory overhead. The second invariant requires *used* and *free* to be disjoint.

MemoryBlock

$$
\begin{aligned}
&free, total : ContiguousMemory \\
&used : \mathbb{P}\, MemoryAddress \\
&\hline
&used \cup free \subseteq total \\
&used \cap free = \varnothing
\end{aligned}
$$

A *MemoryBlock* is initialised by the *MemoryBlockInit* operation, omitted here. It accepts as an input a contiguous area of memory for the memory block and set the values such that the memory block is initally empty.

As an example of an operation on a *MemoryBlock*, we present *MBAllocate*, the operation that allocates memory within a *MemoryBlock*. This operation takes as input *size*?, the requested size of the allocated memory, and outputs an area of contiguous memory called *allocated*!. There is a precondition on this operation: *size*? must be smaller than the size of *free*, to ensure that there is enough free space to fulfil the allocation request. The output, *allocated*!, is then specified to be of the requested size and contained within *free*. The final state $used'$ is obtained by adding *allocated*! to *used* and $free'$ is obtained by removing *allocated*! from *free*. Finally, it is specified that *total* does not change.

MBAllocate

$$\Delta MemoryBlock$$
$$size? : \mathbb{N}$$
$$allocated! : ContiguousMemory$$

$$size? \leq \# free$$
$$\# allocated! = size? \land allocated! \subseteq free$$
$$used' = used \cup allocated!$$
$$free' = free \setminus allocated!$$
$$total' = total$$

With *MemoryBlock* specified, the remaining parts of the backing store state are added in the schema *BackingStore*, which is a *MemoryBlock* with a variable, *self*, to store its own identifier and a finite set, *children*, of the identifiers of its immediate children. The invariants specify that it cannot be a child of itself and that the overhead, left loosely specified in *MemoryBlock*, must have a size equal to some constant *backingStoreOverhead*. The invariants inherited from *MemoryBlock* are also required to hold.

BackingStore

$$MemoryBlock$$
$$self : BackingStoreID$$
$$children : \mathbb{F}\, BackingStoreID$$

$$self \notin children$$
$$\# total = \# used + \# free + backingStoreOverhead$$

A *BackingStore* is initialised by the *BackingStoreInit* operation, which behaves similarly to *MemoryBlockInit* with the additional precondition that the size of the provided memory area must be larger than *backingStoreOverhead*.

An example of a *BackingStore* operation is the operation of allocating space for a new child backing store within its parent, given by the the schema *BSAllocateChild* below.

BSAllocateChild

$$\Delta BackingStore$$
$$MBAllocate$$
$$childID! : BackingStoreID$$

$$childID! \notin children \land childID! \neq self$$
$$children' = children \cup \{childID!\}$$
$$self' = self$$

This operation is defined using *MBAllocate*, but has an additional output, *childID*!, which is the identifier of the newly allocated child, and specifies how the extra state components in *BackingStore* are updated. The new child's identifier is specified to not be the identifier of an existing child or of the backing store itself. The final value of $children'$ is specified to include *childID*! as well as the identifiers in *children*, and *self* does not change. This operation is used to define a robust operation, *RBSAllocateChild*, that handles the case where the precondition defined in *MBAllocate* does not hold by outputting a value *report*!, which is either a reported error or the value *okay*.

Backing stores are managed by the global memory manager, the state of which is given by the schema below. The global memory manager state contains a partial function, *stores*, that relates backing store identifiers to the backing stores, along with the identifier *rootBackingStore* of the root backing store, and a relation, *childRelation*, between backing store identifiers and the identifiers of their direct children. The global state also contains a map, *threadACs*, from thread identifiers to their allocation context, which is used, together with information obtained from the scheduler, to perform operations on the current allocation context.

The relationships between the backing stores are specified by the invariants of this state as defined by the predicates in the schema below. The first invariant specifies that the root backing store must be a valid identifier, that is, in the domain of *stores* (this is implied by the sixth invariant, but is written separately for clarity). The second invariant requires that the value of *self* for each backing store is its own identifier. The third invariant requires that a backing store's children are all disjoint and contained in that backing store's used memory. The fourth invariant requires that the thread allocation contexts be valid backing stores. The fifth invariant defines *childRelation*, using the set of children in the backing store record to form a relation between backing store identifiers. The sixth invariant uses the reflexive transitive closure $childRelation^*$ of *childRelation* to specify that every known backing store must be a (direct or indirect) child of the root backing store or the root backing store itself. Lastly, the seventh invariant specifies that no backing store can be a (direct or indirect) child of itself; $childRelation^+$ is the transitive closure of *childRelation*.

GlobalMemoryManager

$$stores : BackingStoreID \pfun BackingStore$$
$$childRelation : BackingStoreID \rel BackingStoreID$$
$$rootBackingStore : BackingStoreID$$
$$threadACs : ThreadID \pfun BackingStoreID$$

$$rootBackingStore \in \mathrm{dom}\ stores$$
$$\forall b : \mathrm{dom}\ stores \bullet (stores\ b).self = b$$
$$\forall b : \mathrm{ran}\ stores \bullet \exists m : \mathbb{P}\, b.used \bullet (\lambda x : b.children \bullet (stores\ x).total)\ \mathrm{partition}\ m$$
$$\mathrm{ran}\ threadACs \subseteq \mathrm{dom}\ stores$$
$$childRelation = \bigcup \{i : \mathrm{dom}\ stores \bullet \{j : (stores\ i).children \bullet (i, j)\}\}$$
$$\mathrm{dom}\ stores = (childRelation^*)\ (\!|\ \{rootBackingStore\}\ |\!)$$
$$\forall s : \mathrm{dom}\ stores \bullet s \notin childRelation^+\ (\!|\ \{s\}\ |\!)$$

The operations on the memory manager are specified using the Z idiom of promotion, which allows operations on a local state to be lifted to operations on a global state. This

allows simple operations on the backing store records to be used to update the backing stores in the *stores* function. As an example we present the *GlobalMakeBS* schema that specifies the creation of a backing store within another, which is ultimately used to specify the `makeBackingStore` operation.

Much of this operation's complexity comes from the fact that it is promoting two operations: *RBSAllocateChild* and *BackingStoreInit*. The allocation occurs in the current allocation context, identified by the input *allocationContext*?. It is determined by obtaining the current thread from the scheduler, looking up its allocation context and passing it as an input to the operation specified by the Z schema.

The first condition on the operation requires that the allocation context be a valid backing store. After that, the existential quantifiers introduce variables local to the schema, specifying the size of the allocated backing store, *actualSize*, to include some implementation-defined overhead.

The variables *childID*! and *childAddresses*! are identified with outputs of the promoted operation *RBSAllocateChild* that have the same names. The initial state for the promoted operation is taken from the backing store in *stores*, and the final state is placed in the variable *parent*. The output *childID*! is required to not be the identifier of another backing store. It is required that no error be reported in the error reporting variable *report*!.

The second operation promoted is *BackingStoreInit*. The variables *actualSize*, *childAddresses*! and *childID*! are passed to it as inputs via renaming. The final state of the promoted operation is stored in *child*. The final states of both backing stores, *parent* and *child* are stored in the *stores* function, with *childID*! being used as the identifier of *child*. The specification of the operation ends by stating that all other state variables remain the same.

$$
\begin{array}{|l}
\hline
\mathit{GlobalMakeBS} \\
\Delta \mathit{GlobalMemoryManager} \\
\mathit{size?} : \mathbb{N} \\
\mathit{allocationContext?} : \mathit{BackingStoreID} \\
\hline
\mathit{allocationContext?} \in \operatorname{dom} \mathit{stores} \\
\exists\, \mathit{actualSize} : \mathbb{N} \mid \\
\quad \mathit{actualSize} = \mathit{size?} + \mathit{backingStoreOverhead} \bullet \\
\exists\, \mathit{childID}! : \mathit{BackingStoreID} \bullet \\
\exists\, \mathit{allocated}! : \mathit{ContiguousMemory} \bullet \\
\exists\, \mathit{parent}, \mathit{child} : \mathit{BackingStore} \bullet \\
\quad (\exists\, \Delta \mathit{BackingStore};\ \mathit{report}! : \mathit{Report} \mid \\
\qquad \mathit{RBSAllocateChild}[\mathit{actualSize}/\mathit{size?}] \bullet \\
\qquad \theta\, \mathit{BackingStore} = \\
\qquad\quad \mathit{stores}\ \mathit{allocationContext?} \land \\
\qquad \mathit{parent} = \theta\, \mathit{BackingStore}' \land \\
\qquad \mathit{childID}! \notin \operatorname{dom} \mathit{stores} \land \\
\qquad \mathit{report}! = \mathit{okay}) \land \\
\quad (\exists\, \mathit{BackingStore}';\ \mathit{report}! : \mathit{Report} \mid \\
\qquad \mathit{BackingStoreInit}[\mathit{allocated}!/\mathit{addresses?}] \bullet \\
\qquad \mathit{child} = \theta\, \mathit{BackingStore}') \land \\
\quad \mathit{stores}' = \mathit{stores} \oplus \\
\qquad \{\mathit{allocationContext?} \mapsto \mathit{parent}, \\
\qquad \mathit{childID}! \mapsto \mathit{child}\} \\
\mathit{rootBackingStore}' = \mathit{rootBackingStore} \\
\mathit{childRelation}' = \mathit{childRelation} \\
\mathit{threadACs}' = \mathit{threadACs} \\
\hline
\end{array}
$$

Other global memory manager operations are similarly specified using promotion. Operations on stacks are specified separately, using a specification based on *MemoryBlock*. The Z schemas that specify the operations are used to define *Circus* actions that are used in the memory manager process, with the inputs and outputs communicated over channels. For the operation described above, the *Circus* action is *MakeBackingStore*, which accepts input on the channel *MMmakeBackingStore* and returns its output on the channel *MMmakeBackingStoreRet*.

5. CONCLUSIONS AND RELATED WORK

In this paper we have presented the requirements of an SCJVM, identified by examining the SCJ standard and considering existing implementations. One of the authors has been involved in the SCJ standard specification, and this has allowed us to obtain clarifications on the occasions where the requirements on the SCJVM interface were unclear. Of course, we cannot guarantee that these requirements are complete and, hence we welcome comments indicating omissions from the community. Furthermore, the interface we define is not the only interface that can be designed to meet the overall requirements. However, we contend that any SCJVM must provide the functionality specified here in some way.

We have also presented a formal model of an SCJVM, written in *Circus*. Our future efforts in using our formal model of the requirements can also lead to their further elaboration. For example, in our overview of our *Circus* formal model of the requirements, we have focused on the memory manager. One of the main design goals of the SCJ specification in this area has been to "enable the development of SCJ applications that are not vulnerable to reliability failures due to memory fragmentation" [17]. A correct SCJVM that follows the requirements specified here and that is used in conjunction with a correct implementation of the SCJ infrastructure is guaranteed to deliver fragmentation free memory management. A proof of properties of our model and a verification of an SCJVM implementation can well lead to identification of additional requirements.

This work is done in the context of a wider effort to apply formal methods to Safety-Critical Java. There has already been work done on generating correct SCJ programs from *Circus* models [6], as well as work on formalising the SCJ memory model [5]. Those works aim at ensuring that SCJ programs can be verified.

There has also been work on modelling virtual machines for Java, and on the formal correctness of compilers targeting those virtual machines. Some of the most complete work in that area was by Stärk, Schmid and Börger [25], who present a model of the full Java language and virtual machine, along with a formally verified compiler, although for an old version of Java. Other work has also been done on modelling the JVM and Java compilation using refinement techniques [8], similar to those used in the works mentioned above on generating SCJ programs from *Circus* models. Additionally there has been work considering machine-checked models of Java virtual machines and compilers [16, 19, 26].

This work can be viewed as the next stage of the effort to provide complete formal verification for SCJ, as a correct program that has been compiled correctly must run on a correct virtual machine for the whole system to be correct.

Future work includes further consideration of the core execution environment and the application of formal methods to the approach of compilation to native code that many virtual machines for SCJ take. We will also tackle the veri-

fication of icecap against the model presented here and the possible construction of a correct SCJVM from this model.

Acknowledgements This work is supported by EPSRC Grant EP/H017461/1. No new primary data were created during this study.

6. REFERENCES

[1] A. Armbruster, J. Baker, A. Cunei, C. Flack, D. Holmes, F. Pizlo, E. Pla, M. Prochazka, and J. Vitek. A real-time Java virtual machine with applications in avionics. *ACM Trans. Embed. Comput. Syst.*, 7(1):5:1–5:49, Dec. 2007.

[2] Atego. Atego Perc Pico - Products - Atego. `http://www.atego.com/products/atego-perc-pico/`, 2015.

[3] J. Baxter. Requirements for Safety-Critical Java Virtual Machines. Technical report, University of York, 2015. `http://www.cs.york.ac.uk/circus/publications/techreports/reports/scjvm-requirements.pdf`.

[4] P. Bertelsen. Dynamic semantics of Java bytecode. *Future Gener. Comp. Sy.*, 16(7):841–850, 2000.

[5] A. Cavalcanti, A. Wellings, and J. Woodcock. The Safety-Critical Java memory model: A formal account. In M. Butler and W. Schulte, editors, *FM 2011: Formal Methods*, volume 6664 of *Lect. Notes Comput. Sc.*, pages 246–261. Springer Berlin Heidelberg, 2011.

[6] A. Cavalcanti, F. Zeyda, A. Wellings, J. Woodcock, and K. Wei. Safety-Critical Java programs from Circus models. *Real-Time Syst.*, 49(5):614–667, Sept. 2013.

[7] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, 2011.

[8] A. Duran, A. Cavalcanti, and A. Sampaio. An algebraic approach to the design of compilers for object-oriented languages. *Form. Asp. Comput.*, 22(5):489–535, 2010.

[9] J. F. Ferreira, C. Gherghina, G. He, S. Qin, and W.-N. Chin. Automated verification of the FreeRTOS scheduler in Hiip/Sleek. *Int. J. Software Tools Technol. Trans.*, 16(4):381–397, 2014.

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

[11] A. Gotsman and H. Yang. Modular verification of preemptive os kernels. *J. Funct. Program.*, 23:452–514, 2013.

[12] M. Jones. The functions of Java bytecode. In *Workshop on the Formal Underpinnings of the Java Paradigm*, 1998.

[13] G. Klein, J. Andronick, K. Elphinstone, T. Murray, T. Sewell, R. Kolanski, and G. Heiser. Comprehensive formal verification of an OS microkernel. *ACM Trans. Comput. Syst.*, 32(1):2:1–2:70, Feb. 2014.

[14] D. Lime and O. Roux. Formal verification of real-time systems with preemptive scheduling. *Real-Time Syst.*, 41(2):118–151, 2009.

[15] T. Lindholm, F. Yellin, G. Bracha, and A. Buckley. *The Java virtual machine specification*. Pearson Education, 2014.

[16] A. Lochbihler. *A Machine-Checked, Type-Safe Model of Java Concurrency: Language, Virtual Machine, Memory Model, and Verified Compiler*. KIT Scientific Publishing, 2012.

[17] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, A. Wellings, et al. *Safety-Critical Java Technology Specification*. The Open Group, Jun 2013.

[18] K. S. Luckow, B. Thomsen, and S. E. Korsholm. HVMTP: A time predictable and portable Java virtual machine for hard real-time embedded systems. In *Proceedings of the 12th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '14, pages 107:107–107:116, New York, NY, USA, 2014. ACM.

[19] T. Nipkow, D. von Oheimb, and C. Pusch. μjava: Embedding a programming language in a theorem prover. In F. L. Bauer and R. Steinbrüggen, editors, *Foundations of Secure Computation, volume 175 of NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[20] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3–32, 2009.

[21] F. Pizlo, L. Ziarek, and J. Vitek. Real time Java on resource-constrained platforms with Fiji VM. In *Proceedings of the 7th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '09, pages 110–119, New York, NY, USA, 2009. ACM.

[22] M. Richard-Foy, T. Schoofs, E. Jenn, L. Gauthier, and K. Nilsen. Use of PERC Pico for safety critical Java. In *Conference Proceedings: Embedded Real-Time Software and Systems, Toulouse, France*, 2010.

[23] A. W. Roscoe. *Understanding Concurrent Systems*. Texts in Computer Science. Springer, 2011.

[24] H. Søndergaard, S. E. Korsholm, and A. P. Ravn. Safety-critical Java for low-end embedded platforms. In M. Schoeberl and A. Wellings, editors, *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 44–53. ACM, 2012.

[25] R. Stärk, J. Schmid, and E. Börger. *Java and the Java Virtual Machine*. Springer-Verlag, 2001.

[26] M. Strecker. Formal verification of a Java compiler in Isabelle. In A. Voronkov, editor, *Automated Deduction — CADE-18*, pages 63–77. Springer, 2002.

[27] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., 1996.