Modelling and Verifying a Priority Scheduler for an SCJ Runtime Environment

Leo Freitas\textsuperscript{1}, James Baxter\textsuperscript{2}, Ana Cavalcanti\textsuperscript{2}, and Andy Wellings\textsuperscript{2}

\textsuperscript{1} Newcastle University, Newcastle upon Tyne, UK
\texttt{leo.freitas@ncl.ac.uk}

\textsuperscript{2} University of York, Heslington, UK
\texttt{jeb531@york.ac.uk}

\textbf{Abstract.} Safety-Critical Java (SCJ) is a version of Java suitable for programming real-time safety-critical systems; it is the result of an international standardisation effort to define a subset of the Real-Time Specification for Java (RTSJ). SCJ programs require the use of specialised virtual machines. We present here the result of our verification of the scheduler of the only SCJ virtual machine up to date with the standard and publicly available, the icecap HVM. We describe our approach for analysis of (SCJ) virtual machines, and illustrate it using the icecap HVM scheduler. Our work is based on a state-rich process algebra that combines Z and CSP, and we take advantage of well established tools.

\textbf{Keywords:} Java · SCJ · Circus · Process algebra · FDR

\section*{1 Introduction}

There has been an international effort to make Java and its Runtime Environment (RTE) suitable for safety-critical systems. All proposed extensions to Java have an associated Java Specification Request (JSR), a Reference Implementation (RI) and a Technology Compatibility Kit (TCK). The Safety-Critical Java (SCJ) specification (JSR 302) is an Open Group Standard \cite{20}, based on a subset of the Real-Time Specification for Java (RTSJ) \cite{35}. It defines Java services designed for applications requiring certification. It replaces Java’s memory model with support for memory regions \cite{34}, and its execution model is based on missions and event handlers with a predictable scheduler.

The goal of an RI is to demonstrate the feasibility of implementing a proposed JSR and to illustrate its impact on the standard Java RTE. The RI for JSR 302 consists of the addition of the \texttt{javax.safetycritical} package and a modified JVM. Together these make up the SCJ RTE.

The TCK is a suite of test programs that check that an implementation conforms to a JSR. The SCJ TCK, when available, will provide a degree of confidence in the correctness of an SCJ RTE. It is unlikely, however, that this will be adequate for systems with the highest certification level. For SCJ to become a viable technology, certified runtime environments must become available.
The constraints embedded in the SCJ design makes programs amenable to formal analysis. Schedulability analysis techniques [3] can be used to provide evidence that programs meet their deadlines. Ongoing effort to support development, validation, and verification of SCJ programs has already produced results [6,7,16,24,33]. The work presented here uses formal methods to increase confidence and provide evidence that an SCJ RTE satisfies its requirements.

SCJ programs cannot run on a standard JVM; they require specialised support for memory regions and preemptive priority-based scheduling. The SCJ RI is under development and will be based on JamaicaVM [18]. To our knowledge, there are currently five SCJVM (virtual machines that support SCJ): Fiji VM [28], icecap HVM (Hardware near Virtual Machine) [32], Ovm (Open Virtual Machine) [1], HVMTP [21] and PERC Pico [2,29]. Of these, Fiji VM and Ovm are not specific for SCJ, PERC Pico does not conform to the current version of SCJ, and HVMTP is based on the icecap HVM.

As far as we know, the only SCJVM that is up to date with the SCJ standard and publicly available is the icecap HVM. Here, we consider the verification of its single-processor scheduler, a core component of an SCJVM. We present a formal model, and establish some of its properties by model checking and theorem proving. This is part of a larger effort to produce a completely verified SCJ RTE. We also present the general approach to construct and analyse formal models of an SCJVM that we use in the verification of the icecap HVM scheduler.

An identification of requirements for an SCJVM and an associated formal model are presented in [4]. The modelling uses Circus [26], a state-rich process algebra that combines Z and CSP. Circus is a notation for refinement and can be used to compare the requirements in [4] with models of implementation.

In our work, we use Circus processes to specify components of the icecap tools and their integration. Circus has an extension to deal with object-orientation [5]; it is helpful here, since the icecap tools are implemented in Java. We pursue a close match to the implementation structure to provide accurate low-level Circus models. In our approach, the Circus processes define the boundaries of each component of an SCJVM implementation, and their dependencies. Due to the compositionality of refinement, we can analyse these components in isolation. We tackle here a central component of an SCJVM, the scheduler, and identify the assumptions it makes about management of SCJ processes.

In summary, our contributions in this paper are a modelling and analysis technique tailored for an SCJVM, and its application to the scheduler of the only up to date SCJVM publicly available. The discussion of the technique summarizes the lessons we have learned in carrying out this case study. In addition, the Circus model itself is of interest for documentation of the icecap tools and for fostering reuse of the icecap scheduler in the implementation of other SCJVMs.

Next, we present background material for our work: SCJ and the icecap tools, and Circus. Section 3 describes our modelling approach. Sections 4 and 5 present our scheduler model and its analysis. We consider related work in Sect. 6, and conclude in Sect. 7 considering also future work.
2 Preliminaries

We present SCJ and the icecap tools in Sect. 2.1, and Circus in Sect. 2.2.

2.1 SCJ and the Icecap Tools

SCJ places restrictions on the features of Java that can be used, and defines different scheduling and memory models. SCJ has three compliance levels of increasing complexity; the icecap tools support all levels.

An SCJ program is structured as a series of missions, executed sequentially in an order determined by a program-supplied mission sequencer. Each mission manages various schedulable objects: asynchronous event handlers (at levels 0 and 1), and real-time threads and nested mission sequencers (at level 2). A mission execution goes through several phases shown in Fig. 1. First, each of the schedulable objects and any data that may be required for the duration of the mission are initialised. Afterwards, the mission runs until requested to terminate, and then each of the schedulable objects are terminated, any required cleanup is performed, and the mission sequencer runs the next mission.

![Mission Execution Phases](image)

Asynchronous event handlers can be released periodically at set intervals or aperiodically in response to software requests. Schedulable objects are managed by a priority-based preemptive scheduler. Priority ceiling emulation, whereby a thread has its priority elevated upon taking a lock, prevents problems arising from priority inversion in locking [35]. Support for multiprocessor systems allows schedulable objects to be associated with allocation domains that define the processors in which they are allowed to run.

The icecap tool target embedded systems and precompiles Java bytecode to C, in addition to supplying a lightweight bytecode interpreter. The running of SCJ programs is supported by an implementation of the SCJ API, tightly coupled to the SCJVM. The API implementation and the code that supports it are written in Java, with only the most low-level components written in C and assembly. In the scheduler, only the task of process switching is written in C and assembly, with the code to determine which process should run written in Java.

The structure of the scheduler implementation is shown in Fig. 2. The scheduler is triggered by the clock interrupt handler, a singleton instance of Clock-InterruptHandler, which implements the interfaces InterruptHandler and...
Runnable. A process executing the `run()` method of the `ClockInterruptHandler` instance is created when the clock interrupt handler is initialised. Upon receiving a clock interrupt, the icecap HVM calls the clock interrupt handler’s `handle()` method, which switches to the clock interrupt handler’s process. The clock interrupt handler then calls the scheduler’s `getNextProcess()` method.

The scheduler itself is an instance of `PrioritySchedulerImpl`, which calls the `move()` method of `PriorityScheduler` to choose the next process. The `move()` method wakes any sleeping processes that have passed their wake-up time and pops the next process from a priority queue of processes that are ready to run. The work of switching to the new process is then performed via a native method call to the low-level virtual machine (written in C, rather than Java).

In Sect. 4, we present a model of this scheduler: a network of `Circus` processes interacting with the SCJ API implementation and the operating system.

### 2.2 Circus

`Circus` [26] is a formal notation that combines the style for data modelling of the Z [36] notation with that for process specification of CSP [17,30]. Like a CSP model, a `Circus` specification defines processes that communicate over channels, but that, unlike CSP processes, may contain internal state defined in Z. The internal state is encapsulated so that it can only be updated and accessed via communication on the channels of the process.

A `Circus` process is defined as a series of Z paragraphs and `Circus` actions, which are written using a combination of CSP constructs and Z operations. The process definition ends with a main action that defines the behaviour of the process using the actions defined previously in the process. Most CSP operators can be used in `Circus` actions, including external and internal choice, parallel
Table 1. A summary of the Circus notation used in this paper

<table>
<thead>
<tr>
<th>Operator</th>
<th>Circus Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prefixing of signal on channel ( c ) to action ( A )</td>
<td>( c \rightarrow A )</td>
</tr>
<tr>
<td>Prefixing of input on ( c ) of value ( x ) to ( A )</td>
<td>( c?x \rightarrow A )</td>
</tr>
<tr>
<td>Prefixing of output on ( c ) of value of expression ( e ) to ( A )</td>
<td>( c!e \rightarrow A )</td>
</tr>
<tr>
<td>Guarding of ( A ) with predicate ( g )</td>
<td>( g &amp; A )</td>
</tr>
<tr>
<td>Termination</td>
<td>Skip</td>
</tr>
<tr>
<td>External choice of actions ( A ) and ( B )</td>
<td>( A \triangleright B )</td>
</tr>
<tr>
<td>Sequential composition of ( A ) and ( B )</td>
<td>( A ; B )</td>
</tr>
<tr>
<td>Interrupt of ( A ) by ( B )</td>
<td>( A \triangle B )</td>
</tr>
<tr>
<td>Parallel composition ( A ) and ( B ) synchronising on the intersection of channel sets ( cs1 ) and ( cs2 )</td>
<td>( A_{cs1}</td>
</tr>
<tr>
<td>Parallel interleaving of ( A ) and ( B )</td>
<td>( A \parallel B )</td>
</tr>
<tr>
<td>Hiding of channel set ( cs ) in ( A )</td>
<td>( A \setminus cs )</td>
</tr>
</tbody>
</table>

composition, sequential composition, and the interrupt operator. Additionally, Circus includes assignment, if statements, loops, and variable declarations, as well as permitting the use of Z schema data operations in Circus actions. Processes can also be combined using CSP operators: parallelism, hiding, and so on.

Circus has several extensions to cater for time, mobility, synchronicity and so on. Here, we use classes, including the object-oriented extension OhCircus [5].

A detailed account of Circus can be found in [26]. Examples are presented in Sect. 4. Table 1 summarises the action notation that we use here.

3 Verification Approach

In this section, we present our approach to verification of an SCJVM. This arises from previous experiences on modelling and verification of large existing systems [9–11]. It is, however, tailored for the needs of a VM and of an SCJVM, in particular. In this respect, this is our distillation of the lessons learned in applying Circus to reason about the icecap HVM and its scheduler. The application of the approach presented here to the scheduler is the subject of the next section.

Our technique, first of all, creates a model for a piece of code implementing a VM or a component of a VM, like a scheduler, written in any imperative or object-oriented language. Having identified the modules or classes that implement the component, our approach to modelling is in three phases: (a) data, (b) control-flow, and (c) integration modelling. These are described below.

a. Define a Z Data Model. In this phase, we formalise the data types used in the program, via the four steps below. The data types may be in the program in one of three forms: types available in the programming language, types available via a library, like the collection API of Java, or just as pieces of data not necessarily
identified as a data type in the program. For the latter, identification of data
types in the model is a matter of convenience for verification.

1. Define the state to capture the variables used in the program, creating appro-
priate datatypes where necessary.

2. Capture invariants that are expected. These are properties of the data model
we expect the program to satisfy, even if not explicitly checked (but see b.4).

3. Define the procedures (methods, functions, and so on) of the program that
manipulate the data types defined above.

4. Identify their error cases and totalise the data operations. We use theorem
proving to reveal the preconditions of the operations, and extend the model
to totalise those whose precondition are not just true.

By modelling errors to totalise the operations, we achieve a better under-
standing of the data types of the program. Its precise behaviour, which may
not include checking error conditions, is captured in the next phase.

The model may be seen as a suggestion for improving the code structure (like
error checking). Information from the environment of the component may also
need to be identified as a type whose values are communicated in channels.

b. Capture the Control Flows Through the VM. Using the Z model, we construct
a Circus model following the steps below. Roughly, each module (that is, a class,
in the case of a Java program) is modelled by a Circus process or class, and its
procedures by actions and methods in Circus.

1. Define channels corresponding to the services of the component. For each
provided or internal service, we define a pair of channels to model calls and
returns of invocations. For each required external service, we have a single
channel, because we do not model its behaviour.

2. Use processes and classes to capture the modular structure of the program. A
Java class should be modelled by an OhCircus class, if it includes only passive
methods, that is, methods that can be modelled using only data operations
without the use of channel communications, and by a process, otherwise.

3. Define the actions for the services corresponding to the channels above. In this
step, we use the data operations defined in the previous phase, and capture
the control flows in the definition of each action.

4. Eliminate the error cases in the Z data operations that are not handled in the
code, transforming the remaining cases to guards in order to enable identifi-
cation of mistakes in use of data operations via deadlock checks. In this way,
we ensure that the model is not more robust than the code, and any invalid
assumptions about the use of the data can be revealed by analysis.

5. Define in the main action how the services are to be provided. In principle,
all actions could be combined in an external choice, so that their services are
available for use one at a time. A call graph, however, may identify services
that are needed in parallel because they are part of different lines of execution.
c. Generate a Circus model of the component. We use the processes defined in phase (b) to produce a model of the component as a network of processes.

1. Introduce processes to reflect the parallel design (if any) of the code.
2. Combine the various processes to define the component.

In this phase, we need to make the case that the model is closely related to the program. The argument should explain how the program modules are reflected in the structure of the Z data model and of processes in the network of phase (b). We need to explain how the model can be refined back to the code, and that needs to be relatively simple. For Java programs, a strong argument includes a class diagram and a mapping from the Circus processes to that diagram.

A truly faithful model defines the modelled procedures of the program as actions in Circus using programming constructs (assignments, loops, and so on) just like in the code. For reasoning, however, it is convenient to use a predicative specification of the procedures in Z. This is the reason for the abstraction in phase (a), followed by the argument constructed here. As a consequence, we catch integration problems via formal analysis, but not necessarily programming errors. The modelling effort, however, may well reveal programming errors in phases (a) and (b.4–5). In our case study, we found missing error checks.

With a model produced as described above, we open the possibility of the use of a multitude of analysis techniques. We distinguish the following possibilities as particularly useful in the case of SCJVM analysis.

1. Prove that the Circus model is deadlock free
2. Use refinement to prove more general properties.

For the icecap case study, we carry out proof of properties of the Z data model using Z/Eves [25], and translate the Circus model to CSP to use the FDR3 model checker [14]. In the translation, we lose the expressiveness of Z, but gain the ability to use automatic analysis of the process network.

More details about our approach to modelling and analysis are in [12].

4 Formal Model Overview

We next give an overview of our model of the icecap HVM scheduler. The complete model can be found in [12, Ch. 5]; its components are shown in Fig. 3. There is a Circus process for each Java class in Fig. 2. The environment includes the low-level virtual machine written in C, the operating system, and other components of the SCJVM, including the SCJ API. These components communicate with the scheduler to initialise it and to obtain information. PrioScheduler, corresponding to the PriorityScheduler class, receives requests to move and stop SCJVM processes from PrioSchedulerImpl. It also communicates with the ClockInterruptHandler to enable and disable interrupts, and to register and start the clock handler. ClockInterruptHandler communicates with PrioSchedulerImpl to obtain the next SCJVM process to run. PrioSchedulerImpl is a bridge between ClockInterruptHandler and PrioScheduler, using services of PrioScheduler to
determine the next process. \texttt{ClockInterruptHandler} also sends requests to transfer between processes to the low-level virtual machine.

Next, we describe how our model is obtained using the approach in the previous section. In phase (a), we define a Z data model. The most interesting types come from the class \texttt{PriorityScheduler}. Its state (a.1) has four parts: (i) the current time, the identifier of a \texttt{ClockInterruptHandler}, and a reference to an unique instance of \texttt{PriorityScheduler} itself; (ii) the references to the processes managed by the scheduler; (iii) the scheduling queues containing the processes that are ready, sleeping/blocked, locked/waiting, and so on; and (iv) the SCJ event handlers managed by the scheduler. The managed processes are identified by unique elements of a set \texttt{PID} of identifiers used by the operating system.

The managed event handlers are modelled by the schema \texttt{HandlerSet}. It contains \texttt{PID} sets representing the different categories of handlers: periodic (\texttt{peh}), aperiodic (\texttt{aeh}), and one shot (\texttt{oseh}), as well as sets of allocated (\texttt{meh}) and free (\texttt{freeHS}) handlers. It also includes a dummy identifier \texttt{idle}, used to avoid management of empty queues: it is queued, when the ready queue is emptied.

\begin{verbatim}
HandlerSet
  peh, aeh, oseh, meh, freeHS : \mathbb{P} \texttt{PID}
  idle : \texttt{PID}

idle \in peh \wedge \langle peh, aeh, oseh \rangle \text{ partition meh}
\langle meh, freeHS \rangle \text{ partition PID}
\end{verbatim}

The state invariant (a.2) establishes that the \texttt{idle} process is a periodic event handler, and that the allocated managed event handlers partition the different categories. Similarly, all values in \texttt{PID} correspond to managed processes: the allocated (\texttt{meh}) and free (\texttt{freeHS}) identifiers partition \texttt{PID}.

The methods corresponding to \texttt{HandlerSet} operations are captured as Z operations (a.3). They are simple and involve adding and removing various handlers.
from the corresponding sets, for example. Proving that the state invariant is preserved by these operations shows whether or not they are feasible. This is part of the totalisation process (a.4) that identifies error conditions to be dealt with. For example, there are operations in the scheduler that take a handler as input. There is, however, no check in the code that the input is a handler managed by the scheduler. This is an example of where we have uncovered possible issues, or hidden assumptions, of the code through modelling and proof.

In phase (b), the SCJVM-specific control-flow is captured using Circus processes. We cater for the scheduling-specific features. For illustration, we describe below the Circus basic process representing the PriorityScheduler class.

To identify the services (b.1), we perform a call-graph analysis of the non-private methods and create corresponding channels. As shown in Fig. 3, the scheduler interacts with the SCJ API and the operating system. A path in the call graph involving the public method transfer is as follows. Although transfer is a method of Process, because we model Process as a data type, conceptually, we regard transfer as a method of ClockInterruptHandler.

ClockInterruptHandler.run, PrioritySchedulerImpl.getNextProcess, ClockInterruptHandler.disable, PriorityScheduler.move, PriorityScheduler.stop, ClockInterruptHandler.transfer

The path above is part of a line of execution (and is public). So, we have a channel transfer corresponding to uses of this service. The types of the channels depend on the associated method’s parameter and return types.

We follow a naming convention to identify what method call and return we are capturing. For instance, the channel KPSreleaseCall represents the package (K) method of the PriorityScheduler class (PS) named release that is being called. Public methods follow a similar naming, and private methods are represented with subsidiary actions, so there are no channels associated with them.

As already mentioned, for each class of the scheduler, we define a process (b.2). In defining actions (b.3), we also take advantage of the call graph. As an example, we present below the action for the release method.

Release \cong\ KPSreleaseCall?apeh \rightarrow\ PCIHdisableCall \rightarrow\ PCIHdisableRet \rightarrow\ (pre ReleaseHandler & ReleaseHandler);\ PCIHenableCall \rightarrow\ PCIHenableRet \rightarrow\ KPSreleaseRet \rightarrow\ Skip

It is triggered by a call via KPSreleaseCall, and concludes with a synchronisation on KPSreleaseRet. Its body contains a call to disable followed by a data operation ReleaseHandler of HandlerSet and by a call to enable. The precondition pre ReleaseHandler of ReleaseHandler is used as a guard (b.4); the input apeh is used in ReleaseHandler and its guard pre ReleaseHandler.

Another example is the action SCJStop corresponding to the method Stop. It takes an input curr, with the guard curr \neq nullpid, which corresponds to the
precondition of the method.

\[ SCJStop \triangleq KPSstopCall?curr : (curr \neq nullpid) \rightarrow PVMtransfer!curr!mainProcess \rightarrow KPSstopRet \rightarrow \text{Skip} \]

The input \( curr \) is passed on, along with the state component \( mainProcess \) corresponding to an inherited field of the class \( \text{PriorityScheduler} \), to the lower-level virtual machine using a channel \( PVMtransfer \).

To conclude phase (b), we identify the services of the API that are provided in choice and in interleaving (b.5). Following the structured indicated in Fig. 3, we define that the \( \text{PriorityScheduler} \) API has three separate groups, which we name \( \text{SCJApi} \), \( \text{SCJRTE} \), and \( \text{CIHApi} \), containing services provided to the SCJ infrastructure, the runtime environment and the \( \text{ClockInterruptHandler} \). The three groups of services are combined in interleaving, with each of its constituent services in external choice. The choice is external, since it provides to the environment of the \( \text{PriorityScheduler} \) (see Fig. 3) the choice of which service to execute. The interleaving defines an action \( \text{Run} \).

\[
\begin{align*}
\text{CIHApi} & \triangleq \text{Move} \Box \text{SCJStop} \\
\text{SCJRTE} & \triangleq \text{Start} \Box \text{Release} \Box \text{AddOuterMostSeq} \cdots \\
\text{SCJApi} & \triangleq \text{GetHWPrio} \Box \text{GetPrio} \\
\text{Run} & \triangleq \text{SCJApi} || \text{SCJRTE} || \text{CIHApi}
\end{align*}
\]

In the main action of \( \text{PriorityScheduler} \), which is distinguished below by a preceding \( \bullet \) symbol, after an initialisation using an action \( \text{Init} \), another action \( \text{Execute} \) uses \( \text{Run} \) to provide the services of the scheduler.

\[
\begin{align*}
\text{Catch} & \triangleq \text{PCIHcatchError}?e \rightarrow \text{Skip} \\
\text{Execute} & \triangleq \text{Run} \triangledown \text{Catch} \\
\bullet \text{Init} ; \text{Execute}
\end{align*}
\]

Low-level (VM) exceptions might interrupt the control flow. These exceptions may occur as a result of user-code runtime exceptions, VM-generated exceptions from environmental assumption violations (like out of memory), or residual design errors. They are indicated via a channel \( \text{PCIHcatchError} \) as defined in the action \( \text{Catch} \), used in \( \text{Execute} \) to define the possible interruption of \( \text{Run} \).

Finally, in phase (c) we define the \( \text{Circus} \) processes network linking together all processes representing classes from Fig. 2; it is as follows.

\[
\begin{align*}
\text{process IcecapVM} & \triangleq \left( \begin{array}{c}
\text{ClockInterruptHandler} \ CihPsInterface \\
\text{PrioScheduler} \ PSIInterface \\
\text{PrioSchedulerImpl} \ ScjPInterface \\
\text{ScjProcess} \ ScjInterface
\end{array} \right) \setminus \text{csSCJRTE}
\end{align*}
\]
Channel sets \textit{CihPsInterface}, \textit{PSIInterface}, \textit{ScjPInterface}, and \textit{ScjInterface} are defined to include all channels used in each process. A final set containing the internal channels identified in (b.1) is used in a hiding: \textit{csSCJRTE} above.

5 Evaluation

The icecap classes \texttt{PrioScheduler} and \texttt{ClockInterruptHandler} are modelled as processes. For other infrastructure classes, like \texttt{PriorityFrame}, for example, only a data model is provided, because their provided services are support operations over such data, rather than active lines of execution or SCJ provided services.

The icecap Java code associated with the component in Figs. 2 and 3 amounts to about 1600 lines of code. Following the modelling technique in Sect. 3, and illustrated in Sect. 4, we obtain a \texttt{Circus} model presented in its entirety in [12]. There, we also find Z/Eves proofs of the totalisation of various Z schema operations, as well as a CSP version of the \texttt{Circus} model used for refinement and deadlock freedom checks. Table 2 provides a summary of numbers of definitions and proofs, to provide an overall total of 755. The nature of the actual proofs using Z/Eves and FDR3 is discussed in the sequel.

<table>
<thead>
<tr>
<th>Z Declarations</th>
<th>Total</th>
<th>\textit{Circus} Declarations</th>
<th>Total</th>
</tr>
</thead>
<tbody>
<tr>
<td>Unboxed items</td>
<td>84</td>
<td>Channel declarations</td>
<td>51</td>
</tr>
<tr>
<td>Axiomatic definitions</td>
<td>28</td>
<td>Channel set declarations</td>
<td>13</td>
</tr>
<tr>
<td>Generic axiomatic definitions</td>
<td>2</td>
<td>Process declarations</td>
<td>12</td>
</tr>
<tr>
<td>Schemas</td>
<td>77</td>
<td>Actions</td>
<td>83</td>
</tr>
<tr>
<td>Generic schemas</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Theorems</td>
<td>202</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Proofs</td>
<td>202</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

5.1 Z/Eves Proofs

We have used the CZT tools \cite{23}\footnote{See also \url{http://czt.sourceforge.net}.} to develop the \texttt{Circus} model. These tools include \texttt{Circus} as an extension of Z within its Eclipse interface. CZT also integrates the Z/Eves theorem prover \cite{31} and its proof language as an extension.

Within CZT, we have typeset (in \LaTeX{}), typechecked and proved well-formedness conditions of the whole model. This involves theorems about functions being applied within their domain, axiomatic definitions soundness, type non-emptiness, and so on. These proofs ensure that the model is consistent.
The CZT tools also have a verification-condition generator for Z and Circus. These include well-formedness checks (for instance, functions are called within their domains), and other consistency checks like feasibility of Z schema operations and race-freedom of Circus parallel actions. We have performed mechanised proofs in Z/Eves of each of these generated verification conditions.

Of greater interest are the (21) precondition proofs: they are directly related to the totalisation of operations as described by our approach (a.4). It is useful to discover the various conditions to feature in the Circus model as guards for communications. As explained, this introduces deadlocks whenever they are not satisfied. Other proofs are for well-formedness (12) and various lemmas (169) about involved types to make the precondition proofs viable.

5.2 FDR Refinement Checks

The Circus model is translated to CSP for automatic analysis using FDR3 [14]: a powerful refinement checker for CSP enabling automatic checking for deadlock and livelock freedom, as well as other properties of interest.

The translation strategy from Circus to CSP is beyond the scope of this paper; details about it can be found in [27]. It involves representing the Z data model within FDR’s rich functional language, whereas the Circus CSP constructs are almost in one-to-one correspondence with those of CSP-M. Access to process state is done via channel communication and appropriate parallelism with its corresponding process main action representation in CSP.

Details about this translation for the icecap HVM scheduler model can be found in [12]. Key decisions about data abstraction and simplification of type domains are necessary to avoid state explosion. Even so, FDR can handle quite complex processes and enabled us to perform important consistency checks.

We have checked for deadlock and livelock freedom the processes related to components in Figs. 2 and 3. As expected, deadlock counterexamples occur on either events external to the components, for example, required services from the operating system, or failed precondition proofs modelled as CSP guards. The required services are for handling exceptions thrown by design or at runtime. The guards highlight hidden assumptions the icecap HVM scheduler code makes. For instance, the priority scheduler implicitly expects all processes to be known to the scheduler, and yet we can call the scheduler with “rogue” processes.

The CSP model has 540 lines excluding comments, and contains 4 top-level processes with a total of around 100 implicitly declared processes through let expressions. We use such expressions to encode Circus actions as well as state. We are still working on the process network to deal with complex state invariants, and prove more specific properties of a scheduler and of SCJ.

6 Related Work

There are other works on verification of real-time schedulers. Ferreira et al. [8] have worked on formal verification of the FreeRTOS scheduler using Hip/SLEEK.
They use separation logic to verify memory safety as the FreeRTOS scheduler uses a lot of pointers, which make the use of more traditional formal verification techniques difficult. In our work, memory safety is partially guaranteed by Java's memory model and our challenges arise instead from the complex control flow of the icecap HVM scheduler. On the other hand, it has to be shown that icecap tools generate C code that is memory safe. This is a separate problem of compiler correctness, which is part of our agenda for future work.

A comprehensive verification of the seL4 microkernel is reported in [19]. This includes verification of the scheduler and other areas of the kernel, and a proof that the binary code of the kernel correctly implements the C source code. The verification of the functional properties of the system is machine-checked using a C semantics in Isabelle/HOL. While we focus on the icecap HVM scheduler, we expect that the icecap tools can be completely verified in the future.

For larger kernels, a major challenge in verification is the complex interdependence between the scheduler and the rest of the kernel. Gotsman and Yang [15] have developed an approach for verifying such kernels modularly using separation logic. It is also relevant to embedded systems where size and speed constraints necessitate tight coupling between operating system components. Indeed, Klein et al. [19] note that the call graph of seL4 shows high levels of interdependency between components. Gotsman and Yang demonstrate their approach by verifying a scheduler based on the Linux 2.6.11 scheduler.

We face similar challenges as the icecap tools also target embedded systems, leading to tight coupling between components, but, as said before, our challenges concern the communication between components rather than sharing of pointers. We tackle our challenges by identifying the components and specifying their interfaces. We define them as Circus processes and specify their interaction via parallel networks. Compositional reasoning and refinement can then be used.

The work of Ludwich and Fröhlich [22] verifies a system-level model of a scheduler by annotating its functions with preconditions and postconditions. These annotations constitute a formal specification that the scheduler must fulfil, which is checked using the C/C++ model checker CMBC. This work is perhaps most similar to ours due to its use of preconditions and postconditions, but our approach involves constructing a formal model from the code rather than presenting the requirements as annotations to the code.

Finally, the great value of applying formal methods in the area of scheduling is shown in [13], which reports a verification of the GCC scheduler using a model in Isabelle/HOL. This effort has uncovered a bug in the GCC Itanium scheduler that caused programs to be compiled incorrectly.

7 Conclusion

For SCJ to become a viable technology for use in safety-critical systems, certified runtime environments are essential. The most advanced implementation of an SCJ RTE is provided by the icecap tools. The implementation is complex. A formal model is a major step in the development of a verified RTE for SCJ.
Developing a formal model of existing software is a major challenge. This is made more difficult with the SCJ RTE as we have to model both high and low-level abstractions. We have presented an approach that produces a Circus model for Java and C source code. Automatic construction of models is not possible, but support can be made available for encoding Java types in Z, calculation of preconditions of data operations, and extraction of call graphs, for instance.

Our experience with the icecap tools implementation has been largely positive. Although the code can be hard to fathom in places, we have found just a few bugs, mainly as the result of studying the code in sufficient depth to produce the model. There are also places where there appears to be unreachable code and where more defensive programming techniques can be employed to catch errors that can be introduced during development and maintenance.

Our experience with Circus has also been largely positive. The lack of process inheritance in Circus, however, has hindered some of our efforts. For example, for the ScjProcess class representing the abstraction for a low-level vm.Process within the SCJ paradigm, we need the gotoNextState method, which is redefined in subclasses of ScjProcess that represent periodic and aperiodic handlers and so on. Since we do not have process inheritance in Circus, in our model we have a single process ScjProcess, in which the action corresponding to gotoNextState uses a conditional to model the dynamic binding.

Our future work includes: (1) more analysis of the scheduler, for example, to show it always dispatches the highest priority SCJ event handler; (2) improvement to the code to take into account our results; (3) the analysis of other components of the icecap tools RTE, in particular the memory management module; and (4) extensions of Circus with process inheritance.

Acknowledgements. The authors gratefully acknowledge useful feedback from anonymous referees, and Stephan Erbs Korsholm and Shuai Zhao for their help in understanding the icecap HVM and its rationale. This work is supported by EPSRC Grant EP/H017461/1. No new primary data were created in this study.

Open Access. This chapter is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, duplication, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, a link is provided to the Creative Commons license and any changes made are indicated.

The images or other third party material in this chapter are included in the work’s Creative Commons license, unless indicated otherwise in the credit line; if such material is not included in the work’s Creative Commons license and the respective action is not permitted by statutory regulation, users will need to obtain permission from the license holder to duplicate, adapt or reproduce the material.
References