UNIVERSITY *of York*

This is a repository copy of *Algebraic Compilation of Safety-Critical Java Bytecode*.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/121650/

Version: Accepted Version

Proceedings Paper:
Baxter, J. and Cavalcanti, A. L. C. orcid.org/0000-0002-0831-1976 (2017) Algebraic Compilation of Safety-Critical Java Bytecode. In: Polikarpova, N. and Schneider, S., (eds.) Integrated Formal Methods. Springer , pp. 161-176.

https://doi.org/10.1007/978-3-319-66845-1_11

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

White Rose
university consortium
Universities of Leeds, Sheffield & York

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk/

Algebraic Compilation of Safety-Critical Java Bytecode

James Baxter and Ana Cavalcanti

Department of Computer Science, University of York, UK

Abstract. Safety-Critical Java (SCJ) is a version of Java that facilitates the development of certifiable programs, and requires a specialised virtual machine (SCJVM). In spite of the nature of the applications for which SCJ is designed, none of the SCJVMs are verified. In this paper, we contribute a formal specification of a bytecode interpreter for SCJ and an algebraic compilation strategy from Java bytecode to C. For the target C code, we adopt the compilation approach for icecap, the only SCJVM that is open source and up-to-date with the SCJ standard. Our work enables either prototyping of a verified compiler, or full verification of icecap or any other SCJVM.

1 Introduction

Java is widely used and there is interest in using it for programming safety-critical real-time systems. This has led to the creation of a variant of Java called Safety-Critical Java (SCJ) [16]. It is being developed by the Open Group under the Java Community Process as Java Specification Request 302. SCJ replaces Java's garbage collector with a system of scoped memory areas to allow determination of when objects are deallocated. It also introduces preemptive priority scheduling of event handlers to ensure predictable scheduling.

Due to these new mechanisms, SCJ requires a specialised virtual machine, although, since the syntax of Java is not modified, a standard Java compiler can be used to generate bytecode. There exist some SCJ virtual machines (SCJVMs) [1, 22, 27]; they all allow for code to be compiled ahead-of-time to a native language, usually C, since SCJ targets embedded systems with low resources. As far as we know, the icecap HVM [27] is the only publicly-available SCJVM that is up-to-date with the SCJ specification; it outputs production-quality code.

Neither icecap nor any of the other SCJVMs has been formally verified. In [3], we present a formal account of the services of an SCJVM. Here, we focus on the execution of Java bytecode and its compilation to native C code. We formalise the requirements for an SCJVM bytecode interpreter and a compilation strategy, using the algebraic approach [25] to verify compilation from bytecode to C, with icecap as a source of requirements for our specification. We use C as our target, following the scheme used by icecap that aims for portable native code that can be easily integrated into existing systems. The decision to use bytecode rather than SCJ itself as our source ensures that we can rely on existing Java compilers and work ensuring their correctness. Our focus here is not the development of

an SCJVM or compiler, but a technique that can be used to develop and verify an ahead-of-time compiling SCJVM implementation.

We use algebraic compilation, in which the semantics of the source and target languages are defined using the same specification language, and a compilation strategy is a procedure to apply compilation rules: refinement laws that address the program constructs independently. Implementing the rules using a rewrite engine can produce a prototype verified compiler. Algebraic compilation has been studied for imperative [25] and object-oriented languages [9], and for hardware compilation [21]. Here we use it, for the first time, to compile a low-level language, Java bytecode, to a high-level language, C. While Java bytecode has some high-level features, particularly its notion of objects, we view it as low-level since it is unstructured, with control flow managed using a program counter.

In summary, our main contributions here are

- a formal model of an SCJVM interpreter,
- a set of provably correct compilation rules for transforming this model, and
- a specification of a strategy for applying these rules to transform Java bytecode in the interpreter to a shallow embedding of C code.

All *Circus* models are mechanically checked by the CZT infrastructure, and some domain checks using Z/Eves are available. In doing this we also provide insights into the application of algebraic compilation to compile low-level source languages, and SCJ programs in particular, to high-level targets. While there is existing work on compiling Java to C [23, 27, 30], none of these works include verification of such a compilation.

In Section 2 we present SCJ and the *Circus* language that we use for refinement. Section 3 is an overview of our approach, whose main components are detailed in the subsequent sections: Section 4 describes our SCJVM model; Section 5 discusses the shallow embedding of C in *Circus*; and Section 6 describes our compilation strategy. Section 7 discusses some of our design decisions. We conclude in Section 8, where we discuss related and future work.

2 Preliminaries

We present SCJ in Section 2.1 and *Circus* in Section 2.2.

2.1 Safety-Critical Java

An SCJ program is structured as a sequence of missions. An instance of a class implementing an interface called `Safelet` defines the starting point of an SCJ program, via an initialisation method, and the definition of a mission sequencer that determines the sequence of missions of the program.

Each mission consists of a collection of schedulable objects, which include asynchronous event handlers that can be released aperiodically, in response to a release request, or periodically, at set intervals of time. Each of these schedulable objects is executed on a separate thread. These threads continue executing until the mission is signalled to terminate by one of its own schedulable objects.

Scheduling follows a preemptive priority policy. The threads eligible to run are placed into queues, with one queue for each priority. The thread at the front of the highest priority non-empty queue runs. A priority ceiling emulation system, whereby a thread's priority is raised when it takes a lock, avoids deadlock when it is interrupted by a thread of higher priority.

SCJ replaces the Java garbage collector with a system of memory areas. Different kinds of area are cleared at different times: the immortal memory is never cleared; the mission memory is cleared between missions; a per-release memory is local to an event handler and is cleared after each of its releases; and a private memory can be created and entered as needed.

SCJ uses an API that includes components that provide real-time clocks, support for raw memory accesses, and a lightweight input/output system. Some of the classes from the standard Java API are removed or restricted to ensure the classes required by SCJ are small enough for embedded systems.

2.2 *Circus*

We formalise the bytecode interpreter and our compilation strategy in *Circus* [20]. It is a refinement notation that combines the process-based style of CSP [24] with the data-based style of the Z notation [31]. It also includes programming constructs from Dijkstra's guarded command language [8]. *Circus* is appropriate for our work because it is a notation for refinement, which is a key part of the algebraic approach to verifying compilation, and it permits reasoning about parallelism. It also combines data and reactive behaviour, which enables us to pass from a bytecode program represented as data in an interpreter to a C program with the same control flow as the bytecode program.

Circus specifications define processes: basic or composed from other processes using CSP operators, such as parallel composition, sequence, and internal and external choice. Each process may have an internal state defined using Z and communicates with its environment via channels like a CSP process.

To illustrate the structure of a *Circus* model, we present in Figure 1 a sketch of a simplified model for an SCJVM interpreter. Typically, a specification begins with type and channel declarations. The types are declared as in Z. Channels carry data of the type specified in their declaration. In our example, there are declarations for channels *getInstruction* and *getInstructionRet*, used to obtain the instruction for each address, provided externally in this simplified model.

The state of a process is defined by a Z schema; in the case of the *Interpreter* process, by the schema *InterpreterState*. The components of the state initially have arbitrary values; specific initial values can be defined through Z schemas, such as *InterpreterInit*, which specifies that the *frameStack* component is empty.

The *Loop* action is defined using a CSP guarded choice that offers different actions depending on whether *frameStack* is empty. *Loop* then repeats via a sequential composition with a recursive call. The main action of a process specifies its behaviour at the end, after a spot. The main action of *Interpreter* initialises the state using *InterpreterInit* and then calls *Loop*.

$$
\begin{array}{l}
\mathbf{channel}\ getInstruction : ProgramAddress \\
\mathbf{channel}\ getInstructionRet : Bytecode \\
\quad \vdots \\
\mathbf{process}\ Interpreter \mathrel{\widehat=} \mathbf{begin}
\end{array}
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{InterpreterState} \\
\hline
frameStack : \operatorname{seq} StackFrame \\
pc : ProgramAddress \\
currentClass : Class \\
\hline
frameStack \neq \varnothing \Rightarrow \\
\quad currentClass = \\
\quad\quad (last\ frameStack).frameClass \\
\hline
\end{array}
$$

$$
\mathbf{state}\ InterpreterState
$$

$$
\begin{array}{|l}
\multicolumn{1}{l}{InterpreterInit} \\
\hline
InterpreterState\,' \\
\hline
frameStack' = \varnothing \\
\hline
\end{array}
$$

$$
\begin{array}{l}
\quad \vdots \\
Loop \mathrel{\widehat=} \\
\quad (frameStack \neq \varnothing) \mathbin{\&} HandleInstruction \\
\quad \Box\, (frameStack = \varnothing) \mathbin{\&} StartInterpreter \,; \\
\quad Loop \\
\bullet\ InterpreterInit \,;\ Loop \\
\mathbf{end}
\end{array}
$$

Fig. 1. A sketch of a simple interpreter process

For a full description of *Circus*, we refer to [20]. For a substantial example, we refer to [2], where we present our specification of the SCJVM services.

3 Our Approach to Algebraic Compilation

In the algebraic approach to compilation the source and target language semantics are embedded in the same specification language and compilation is proved correct by establishing a refinement. A series of compilation rules are applied according to a strategy to refine a source program into a representation of the target machine containing the instructions of the target code.

Here, we adapt the approach to deal with a low-level source language. Our approach can be viewed as the usual approach applied in reverse, starting with an interpreter containing the bytecode source program, and proving that it is refined by an embedding of the C code, as shown in Figure 2. The core services of an SCJVM must be available for both the source and target codes.

For a low-level language, a deep embedding is the natural method for representing its semantics, since it is defined in terms of how it is processed by a (virtual) machine. For the C code we must choose whether to use a shallow embedding, representing C constructs by corresponding *Circus* constructs, or a deep embedding, creating a *Circus* model that interprets the C code.

We use a shallow embedding, since it allows existing algebraic laws for *Circus* to be used directly for manipulation of the C code and proof of the compilation rules. A deep embedding would require representing the syntax of C separately in *Circus* and rules for transforming the C code would have to be proved.

The shallow embedding approach is much easier to extend or adapt. If a larger subset of bytecodes needs to be considered or the target C code needs to be modified, in the worst case, we need more or different *Circus* compilation rules. There will be no need to extend the *Circus* model defining the C semantics.

In the next few sections, we describe Figure 2 in more detail. A complete formal account of the components in Figure 2 can be found in [2].

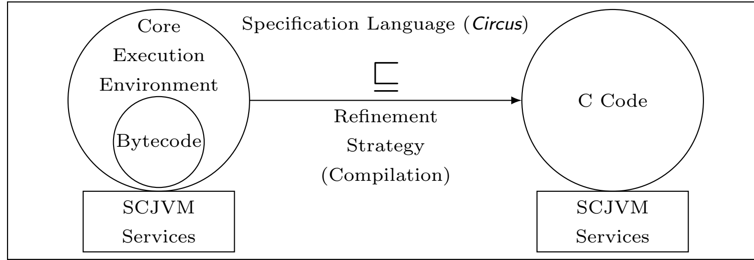

Fig. 2. Our algebraic approach

4 SCJVM and Interpreter Model

Our *Circus* model of the SCJVM has six components, each defined by a single *Circus* process. The first three components are the SCJVM services: the memory manager, the scheduler and the real-time clock. They support the execution of an SCJ program by the core execution environment and are unaffected by the compilation strategy, ensuring the memory management and scheduling models of SCJ are preserved by the compilation strategy.

The remaining components form the core execution environment (CEE), which manages the execution of an SCJ program. It is defined by a parallel composition of three *Circus* processes as shown below. Note that the parallelism here represents composition of requirements, not a requirement for a parallel implementation. In an implementation, such as icecap, these processes would be different parts of the program, which may be made up of C files or Java classes.

$$\mathbf{process}\ CEE(bc, cs, sid, initOrder) \mathrel{\widehat=} \\ \quad ObjMan(cs) \parallel Interpreter(bc, cs) \parallel Launcher(sid, initOrder)$$

CEE uses global constants that characterise a particular program: bc, recording the bytecode instructions, cs, recording information about the classes in the program, sid, recording the identifier of the `Safelet` class, and $initOrder$, a sequence of class identifiers indicating in which order the classes should be initialised. (For simplicity here, and in what follows, we write $\parallel$ to indicate a parallel composition, but omit the definition of the synchronisation sets.)

$ObjMan$ manages the cooperation between the SCJ program and the SCJVM memory manager, including the representation of objects. The SCJVM memory manager is agnostic as to the structure of objects.

$Interpreter$ and $Launcher$ define the control flow and semantics of the SCJ program. The interpreter is for a representative subset of Java bytecode that covers stack manipulation, arithmetic, local variable manipulation, field manipulation, object creation, method invocation and return, and branching. This covers the main concepts of Java bytecode. A full list of the instructions can be found in [2]. We do not include instructions for different types as that would add duplication to the model while yielding no additional verification power. We also do not include exception handling as SCJ programs can be statically verified to prove that exceptions are not thrown [11, 17]. Furthermore, reliance on

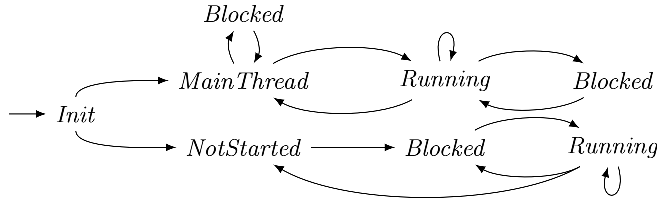

Fig. 3. The overall control flow of *Thr*

exceptions to handle errors has been discouraged by an empirical study due to the potential for errors in exception handling [26]. Errors caused in the SCJVM by an incorrect input program are represented by abortion.

The *Interpreter* interacts closely with the *Launcher*, which defines the flow of mission execution. The *Launcher* begins by creating an instance of the `Safelet` class and then executes programmer supplied methods using *Interpreter*.

We describe *Interpreter* in more detail, since it is a central target of the compilation. A simplified version of it is sketched in Section 2.2. In the full model the *Interpreter* is defined as the parallel composition of *Thr* processes.

$$\textbf{process}\ Interpreter(bc, cs) \mathrel{\widehat=} \mathbin{\|} t : TID \setminus \{idle\} \bullet Thr(bc, cs, t)$$

There is one process $Thr(bc, cs, t)$ for each thread identifier t in the set TID, of thread identifiers, except the identifier of the *idle* thread. Each *Thr* process represents an interpreter for a separate thread, with thread switches coordinated by communication between threads. The state of each *Thr* process is defined by the *InterpreterState* schema in Figure 1. The control flow of the main action of *Thr* is shown in Figure 3. It consists of state initialisation as described by *InterpreterInit*, followed by a choice of two actions, *MainThread* and *NotStarted*, with *MainThread* representing the control flow for the main thread, and *NotStarted* for all other threads. The different behaviours are not described as separate processes because they are similar.

MainThread offers a choice of executing a method in response to a signal from the Launcher or switching to another thread. When it executes a method, *MainThread* creates a new Java stack frame (on *frameStack*) for the method and behaves as *Running*. It repeatedly handles bytecode instructions and polls the scheduler until *frameStack* is empty. Polling occurs inbetween bytecode instructions. (This assumption does not necessarily rule out compiler implementations that do not preserve atomicity, as discussed in Section 7.)

When a mission is initialised, the *Launcher* communicates with *ObjMan* to set up the memory areas for the mission's schedulable objects, and with the scheduler to start their associated threads. This causes the scheduler to signal that the threads are starting. With that, the instances of *Thr* associated with the started threads, which behave as the *NotStarted* action, create a new stack frame and behave as *Blocked*, each waiting for a request to switch to its thread.

After all the threads are started, the *Launcher* signals the scheduler to suspend the main thread, following which the scheduler signals the *Interpreter* to

switch to a new thread. This causes the *Thr* process for the main thread to behave as *Blocked* and the *Thr* process for the new thread to behave as *Running*.

The compilation strategy refines the *CEE* process. Basically it transforms the *Thr* processes, with little change to the other processes. A complete model of the CEE, including the definition of *Thr*, can be found in [2].

5 Shallow Embedding of C in *Circus*

In our approach, compilation generates a C program represented by a *Circus* process. The particular definition of this process depends on the Java bytecode program, as defined by our constants bc and cs, that it implements. So, we refer to the *Circus* process as $CProg_{bc,cs}$, but note that it does not include any reference to these constants, since this is the process that represents the compiled program. For all values of bc and cs, $CProg_{bc,cs}$ has the structure defined below.

$$\mathbf{process}\ CProg_{bc,cs} \mathrel{\widehat=} \mathop{|||}\ t : TID \setminus \{idle\} \bullet CThr_{bc,cs}(t)$$

The parallelism of C threads is represented by a *Circus* parallelism, like the parallelism of Java threads in the *Interpreter*. In $CProg_{bc,cs}$ there is a process $CThr_{bc,cs}$ for each identifier t in the set TID, except for the $idle$ thread identifier.

The $CThr_{bc,cs}$ process has a similar structure to the *Thr* process presented in the previous section, except that the *Running* action is replaced with an *ExecuteMethod* action that executes the C function corresponding to a given method identifier. Within the body of $CThr_{bc,cs}$, each function of the generated C code is represented by a *Circus* action of the same name. The constructs within the C function are represented using *Circus* constructs.

The constructs we allow within a C function are conditionals, while loops, assignment statements, and function calls. These are comparable with those allowed in MISRA-C [18] and present in the code generated by icecap. These constructs can be represented by the corresponding constructs in *Circus*.

As each function in the C code is a *Circus* action, function calls are represented as references to those actions. Function arguments in C are passed by value, although those values may be pointers to other values. Accordingly, since our SCJVM model represents pointers explicitly, we represent function arguments using value parameters of the *Circus* action. Local variables of the function are represented using *Circus* variable blocks.

If a function has a return value, it is represented with a result parameter of the *Circus* action, with an assignment to that parameter at the end of the action representing return statements. It is not necessary to cater for return statements in the middle of a function as we have control over the structure of the functions. We follow guidelines for safety-critical uses of C variants, such as MISRA-C [18], and use a single return statement at the end of a function. A function with both a return value and arguments has its value parameters (representing the arguments) followed by the result parameter (representing the return value).

6 Compilation Strategy

Our compilation strategy refines the $CEE(bc, cs, sid)$ process defined in Section 4 to obtain a process that includes a representation of C code as described in Section 5. The overall theorem for the strategy is as follows.

Theorem 1 (Compilation Strategy). *Given* bc, cs, sid *and* $initOrder$, *there are processes* $StructMan_{cs}$ *and* $CProg_{bc,cs}$ *such that,*

$$CEE(bc, cs, sid, initOrder) \\ \sqsubseteq StructMan_{cs} \parallel CProg_{bc,cs} \parallel Launcher(sid, initOrder).$$

$StructMan_{cs}$ manages objects represented by C structs that incorporate the class information from cs, refining the process *ObjMan*, which handles abstract objects. $StructMan_{cs}$ has Z schemas representing struct types for objects of each class. These schemas contain the identifier *classid* of the object's class, so that polymorphic method calls can be made by choice over the object's class. There are also components for each of the fields of the object.

The schema types for each type of object are combined into a single free type *ObjStruct*. $StructMan_{cs}$ contains a map from memory addresses managed by the SCJVM to the *ObjStruct* type, representing the C structs in memory, and provides access to the individual values in that map.

$CProg_{bc,cs}$ refines the *Interpreter*, with the *Thr* processes refined into the $CThr_{bc,cs}$ processes described in the previous section. This means that the threads from SCJ are mapped onto threads in C, since we do not dictate a particular thread switch mechanism in either the source or target models.

In order to apply the compilation strategy, the *bc* and *cs* inputs must conform to a few restrictions. The most important of these are the restrictions on the structure of control flow: each loop must have only a single exit and conditionals must have a single common end point for all branches. Recursion is also not allowed, directly or indirectly, since method calls cannot be handled unless control flow constructs and method calls in the called method have been introduced first. Finally, the program must be a valid program that could run in the interpreter described in Section 4.

The compilation strategy is split into three stages, each with a theorem describing it, for which the strategy acts as a proof. The proof of Theorem 1 is obtained by an application of the theorems for each stage. All the theorems and their proofs, with a full description of the stages, can be found in [2].

Each stage of the compilation strategy handles a different part of the state of the *Interpreter*: the *pc*, the *frameStack*, and objects. They operate over each of the *Thr* processes, managed by the SCJVM services.

The first stage introduces the control constructs of the C code. This removes the use of *pc* to determine the control flow of the program. The choice over *pc* values is replaced with a choice over method identifiers pointing to sequences of operations representing method bodies.

In the second stage, the information contained on the *frameStack*, which is the local variable array and operand stack for each method, is introduced in

the C code. This is done by introducing variables and parameters to represent each method's local variables and operand stack slots. A data refinement is then used to transform each operation over the *frameStack* to operate on the new variables. The *frameStack* is then eliminated from the state.

In the final stage, the class information from *cs* is used to create a representation of C structs. This means that *ObjMan*, which has a very abstract representation of objects, is transformed into *StructMan*. The process for each thread is then made to access the structs for the objects in a more concrete way that represents the way struct fields are accessed in C code.

This yields final method actions of a form similar to that of the example shown below, which is taken from the `handleAsyncEvent()` method of a simple SCJ event handler class named `InputHandler`.

$$
\begin{array}{l}
InputHandler_HandleAsyncEvent \mathrel{\widehat=} \\
\quad \mathbf{val}\ var0 \bullet \mathbf{var}\ var1, stack0, stack1 : Word \bullet \\
\quad stack0 := var0 \,;\ Poll \,;\ getObject!stack0 \longrightarrow getObjectRet?struct \\
\quad \longrightarrow stack0 := (castInputHandler\ struct).input \,;\ \ldots
\end{array}
$$

The method is compiled to the action *InputHandler_HandleAsyncEvent*, with the implicit `this` parameter represented as a value parameter $var0$. The local variable ($var1$) and stack slots ($stack0$ and $stack1$) are represented as *Circus* variables. The operations of the C code are composed in sequence, with an action named *Poll* that polls for thread switches between each operation. Stack operations are represented as assignments. For instance, $stack0 := var0$ arises from the compilation to load a local variable onto the stack. Access to objects is performed by communicating with $StructMan_{cs}$ to obtain the struct for the object, casting it to the correct type, and accessing the required value. Above, we obtain the value of the *input* field from an *InputHandler* object. The communication with $StructMan_{cs}$ is performed via the *getObject* channel and the function *castInputHandler* is used to map the *ObjectStruct* returned from the communication to a type representing an `InputHandler` object.

We illustrate our approach by giving further details about the first and most challenging stage of the strategy, elimination of program counter. The theorem describing this stage is shown below, where *ThrCF* is the result of transforming *Thr* to eliminate *pc* as indicated above and detailed in the sequel.

Theorem 2. $Thr(bc, cs, t) \sqsubseteq ThrCF_{bc,cs}(cs, t)$

The strategy for this stage is defined by Algorithm 1. Each of its steps is defined by its own algorithm, which details how the compilation rules are applied. The correctness of Algorithm 1 (and the other algorithms in the strategy) relies on the correctness proofs for the compilation rules, which are *Circus* laws. The algorithm forms the basis of a proof for Theorem 2 since it provides a strategy to apply the rules to refine $Thr(bc, cs, t)$ into $ThrCF_{bc,cs}(cs, t)$. All that then need be proved is that the algorithm does indeed yield *Circus* code of the correct form.

Algorithm 1 begins, on line 1, by expanding the semantics of each bytecode instruction (using a copy rule). Afterwards, sequential composition is introduced

Algorithm 1 Elimination of Program Counter

1: EXPANDBYTECODE
2: INTRODUCESEQUENTIALCOMPOSITION
3: **while** ¬ ALLMETHODSSEPARATED **do**
4: INTRODUCELOOPSANDCONDITIONALS
5: SEPARATECOMPLETEMETHODS
6: RESOLVEMETHODCALLS
7: **end while**
8: REFINEMAINACTIONS
9: REMOVEPCFROMSTATE

between bytecode instructions on line 2. Dependencies between methods must be considered in order to introduce the remaining control constructs, since method calls are handled by placing the method invocation bytecode in sequence with a call to a *Circus* action containing the body of the method being invoked. We say a method call for which this transformation has been done is *resolved*. Resolution is necessary to introduce a reference to the method action representing the C function for the method at the appropriate place in the control flow, after the value of *pc* has been set to the method's entry point by the invocation instruction.

The action containing the body of the method can only be created after loops and conditionals have been introduced and the method's body has been sequenced together into a single block of instructions. However, loops and conditionals can only be introduced when all the method calls in their bodies have been resolved (since method calls break up the body of a loop or conditional). For this reason, we perform loop and conditional introduction and method resolution iteratively, until all methods have had their control flow constructs introduced and their bodies copied into separate *Circus* actions. This occurs in the loop beginning at line 3 of Algorithm 1.

Within the loop (lines 4 to 6), loops and conditionals are first introduced to methods that have already had method calls resolved, on line 4. Methods that are in a form in which their control flow is described using C constructs are then copied into separate actions, on line 5. Calls to the separated methods are then resolved, introducing references to the newly created method actions, on line 6.

After all the methods have been copied into separate actions, the *MainThread* and *NotStarted* actions are refined to replace the choice over *pc* with a choice over method identifiers, on line 8. Finally, a data refinement is used to eliminate the *pc* from the state, on line 9.

In our example, the *InputHandler_HandleAsyncEvent* action is created in this stage as shown below. The control flow, mainly sequential composition, has been introduced, but the instructions are in the form of data operations over the *frameStack*. A call to the *InputStream_Read* method action can be seen here.

$$\begin{aligned}&\mathit{InputHandler_HandleAsyncEvent} \mathrel{\widehat=} \mathit{HandleAloadEPC}(0) \mathbin{;} \mathit{Poll};\\ &\quad \mathit{HandleGetfieldEPC}(15) \mathbin{;} \mathit{Poll} \mathbin{;} \mathit{HandleInvokevirtualEPC}(33);\\ &\quad \mathit{Poll} \mathbin{;} \mathit{InputStream_Read} \mathbin{;} \mathit{Poll} \mathbin{;} \mathit{HandleAstoreEPC}(1) \mathbin{;} \cdots\end{aligned}$$

The algorithms for all stages of the strategy can be found in [2]. For illustration, we describe the IntroduceSequentialComposition procedure, referenced on line 2 of Algorithm 1. It begins with construction of a control flow graph for the program, which is then examined for nodes with a single outgoing edge leading to a node with a single incoming edge. Such nodes represent points at which simple sequential composition occurs, rather than more complex control flows such as loops and conditionals that are introduced later in the strategy. At these nodes, the compilation rule given by Rule 1 is applied.

Rule 1 (Sequence introduction) *If* $i \neq j$ *and*

$$\{frameStack \neq \varnothing\} \,;\; A \quad = \quad \{frameStack \neq \varnothing\} \,;\; A \,;\; \{frameStack \neq \varnothing\}$$

then,

$$\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\ frameStack = \varnothing \longrightarrow \mathbf{Skip} \\
\quad [\!]\ frameStack \neq \varnothing \longrightarrow \\
\qquad \mathbf{if}\ \cdots \\
\qquad [\!]\ pc = i \longrightarrow \\
\qquad\quad A \,;\; pc := j \\
\qquad [\!]\ pc = j \longrightarrow B \\
\qquad \cdots \\
\qquad \mathbf{fi} \,;\; Poll \,;\; X \\
\quad \mathbf{fi}
\end{array}
\quad \sqsubseteq_A \quad
\begin{array}{l}
\mu X \bullet \\
\quad \mathbf{if}\ frameStack = \varnothing \longrightarrow \mathbf{Skip} \\
\quad [\!]\ frameStack \neq \varnothing \longrightarrow \\
\qquad \mathbf{if}\ \cdots \\
\qquad [\!]\ pc = i \longrightarrow \\
\qquad\quad A \,;\; pc := j \,;\; Poll \,;\; B \\
\qquad [\!]\ pc = j \longrightarrow B \\
\qquad \cdots \\
\qquad \mathbf{fi} \,;\; Poll \,;\; X \\
\quad \mathbf{fi}
\end{array}$$

This rule, like many of the compilation rules, acts upon *Circus* actions of a generalised form. Where dots ($\cdots$) are shown on the left hand side the rule, it indicates that any syntactically valid *Circus* at that point may match the rule, but remains unaffected by the rule, as indicated by corresponding dots on the right hand side of the rule. The left hand side of this rule is in the form of the *Running* action, with a loop that continues until *frameStack* is empty and a choice over the value of *pc* to select the instruction to execute. The rule unrolls the loop, sequencing the instructions at $pc = i$ with the instructions executed after them at $pc = j$. An occurrence of *Poll* is placed inbetween to permit thread switches. The preconditions for the application of this rule are that i and j not be the same (since that would be a loop), and that the instructions at $pc = i$ preserve the nonemptiness of the *frameStack* (to fulfil the loop condition of *Running*).

We note that the *pc* assignment that causes the sequential composition remains in the code after the application of this rule. It is removed in the data refinement on line 9 of Algorithm 1, since removing it as part of the rule would complicate the preconditions.

The other compilation rules have a similar form to Rule 1 but introduce other constructs such as loops, conditionals, and method calls. An account of all the laws used in the strategy can be found in [2].

7 Discussion

Our work is the first on verified compilation from Java bytecode to C. Our results may be of value in the compilation of standard Java programs, but they are

specific to SCJ. Although SCJ uses the same bytecode instructions as standard Java, SCJ does not have dynamic class loading, which substantially changes the semantics of the bytecode instructions. The class initialisers must also be executed at the start of the program and the program execution must be coordinated according to the SCJ mission paradigm, both performed by the *Launcher* in our model. Finally, the instructions must rely on the SCJVM services, so, for example, the `new` instruction must communicate with the SCJVM memory manager to ensure SCJ's memory model is followed.

We have also considered the introduction of control flow constructs to the compiled C code. This differs from previous work, which translates branch instructions in the bytecode using `goto` statements in the C code. Avoiding the use of `goto` statements permits more optimisation by the C compiler, makes the control flow of the code more readable, and brings the code in line with the restrictions of MISRA C. This has been one of the most challenging parts of our work, since we require a strategy for identifying the control flow constructs of the Java bytecode. In our strategy, we handle branches in the Java bytecode by analysing the structure of the control flow graph for each method. Unconditional jumps are handled in the same way as sequential composition, while conditional branches are handled by introducing C conditionals. Where the jump introduces a loop, we instead introduce looping constructs corresponding to C while loops.

We have also had to consider the difficulties raised by the features of Java when compiling bytecode. Chief among these is the issue of how inheritance and dynamic dispatch are handled. The class where a method is defined must be identified when it is invoked, since a method's bytecode instructions require constant pool information about the class in order to be executed correctly. Since SCJ requires that all classes be available before program execution, the possible classes for a particular method call can be determined statically. When there is a unique class (as will always be the case for `invokestatic` and `invokespecial` instructions), we can replace the method call with a reference to the correct method in the ResolveMethodCalls algorithm (line 6 of Algorithm 1). If there is no unique class then we must determine the set of all possible classes where it can be defined. In ResolveMethodCalls, we compute this set and replace the method call with a choice over the class identifier of the object, the branches of the choice corresponding to the possible methods.

In our strategy we do not handle recursion. This is not a strong restriction, since it is in line with the constraints imposed by MISRA C. Detecting recursion in object-oriented programs is complicated by dynamic dispatch, since mutual recursion may or may not arise depending on dynamic dispatch. However, the fact that SCJ does not allow dynamic class loading means that all the classes are available during compilation, which means dynamic dispatch is constrained by the classes available, making detection of potential recursion feasible.

SCJ also presents several issues of its own in terms of memory management and scheduling. These are handled by the SCJVM services part of our model However, there are many places where a program must interact directly with the SCJ infrastructure, such as entering memory areas or registering an event handler with its mission. To handle such interactions correctly we handle the

calls to methods that cause these interactions in a special way, allowing them to interact with the *Launcher* and the SCJVM services.

In ResolveMethodCalls, we replace the calls to these special methods by communications with other components of the SCJVM. Since the *Launcher* and the SCJVM services remain unchanged throughout the strategy, these communications become calls to C functions in the SCJVM infrastructure. A similar system could be applied to handle native method calls, though we view that as future work since it is not a central part of the considerations for an SCJVM. Native methods would be represented via a shallow embedding in *Circus*, in the same way as the output of the compilation, but would be present before compilation with special handling given to calls to them in the interpreter.

The real-time requirements on SCJ scheduling also impose predictability, so that the bytecode instructions processed by the interpreter must appear to be atomic. This is specified in our model by only permitting thread switches inbetween bytecode instructions. This atomicity requirement is preserved throughout our strategy, and the behaviour of polling for thread switches remains inbetween the C code corresponding to each bytecode instruction.

However, an implementation is only required to have the same sequence of externally visible events as our C code model. This means that the thread switches will appear the same in a non-atomic implementation for most bytecode instructions. The bytecode instructions which have effects visible outside the *Interpreter*, which are the `new` instruction, the field access instructions, and instructions that invoke the special methods mentioned above, interact with shared memory and so do have an atomicity requirement. We can only verify an implementation that ensures such operations are not interrupted, usually by employing synchronisation. This is, of course, the case for icecap.

Our work can be used to verify an SCJVM that uses ahead-of-time compiling, or as a specification to create such an implementation. Since the compiled C code only uses core features of C and is compatible with MISRA C, it can be compiled by most C compilers. So, existing work on verification of C compilers, such as that of CompCert [13, 14], can be used to ensure correct execution of the SCJ program. Since SCJ does not modify the syntax of Java, existing Java compilers can be used to produce the bytecode handled by the strategy.

8 Conclusions

We have described our approach to algebraic compilation of SCJ bytecode. Compiler verification can be complex and, for some languages, compiler updates are common. So, it can be easier to verify properties of the compiler output. In the case of an SCJ compiler, however, SCJ is a controlled language and the core of Java bytecode it uses is fairly stable, as is the only fully compliant SCJVM. In addition, the algebraic approach allows for a modular compilation strategy composed of individual compilation rules. Thus, extending or handling any changes to SCJ would require only changing or adding some compilation rules. The parts of the strategy not directly involved with any changes may be left unchanged.

This compilation strategy is the final component needed to create SCJ programs with assurance of correct execution. Other work that contributes to this goal produces correct SCJ programs from *Circus* specifications [5, 6], and verifies Java [9, 12, 15, 28, 29] and C [4, 13, 14] compilers. Together, these can ensure a complete chain of verification from SCJ programs to executable code.

The mapping from bytecode to C code used in icecap can be used as a basis for the construction of other compilers. A sound implementation can also be obtained by a mechanisation of the strategy via tactics of refinement in a rewriting engine such as Isabelle [19] or Maude [7].

The next stage of our work will be the formalisation and mechanisation of correctness proofs for our strategy. The strategy must also be evaluated by applying it to some examples of SCJ programs to ensure it can handle a wide range of SCJ programs. Further work in the future could include the extension of the strategy to cover more Java bytecode instructions or additional transformations such as code optimisations. Our work will eventually allow the formal verification of a complete SCJVM implementation, an effort that has started in [10].

Acknowledgements
The authors gratefully acknowledge useful feedback from Augusto Sampaio on the application of the algebraic approach. We also thank Andy Wellings for his advice on SCJ and Leo Freitas for his help with the use of Z/EVES and understanding of icecap. This work is supported by EPSRC studentship 1511661 and EPSRC grant EP/H017461/1.

References

1. Armbruster, A., Baker, J., Cunei, A., et al.: A real-time Java virtual machine with applications in avionics. ACM Trans. Embed. Comput. Syst. 7(1), 5:1–5:49 (Dec 2007)
2. Baxter, J.: An Approach to verification of Safety-Critical Java Virtual Machines with Ahead-of-time compilation. Tech. rep., University of York (2017), `www-users.cs.york.ac.uk/~jeb531/2017report.pdf`
3. Baxter, J., Cavalcanti, A., Wellings, A., Freitas, L.: Safety-Critical Java Virtual Machine Services. In: JTRES '15, pp. 7:1–7:10. ACM (2015)
4. Blazy, S., Dargaye, Z., Leroy, X.: Formal Verification of a C Compiler Front-End. In: FM 2006: Formal Methods, Lecture Notes in Computer Science, vol. 4085, pp. 460–475. Springer Berlin Heidelberg (2006)
5. Cavalcanti, A., Wellings, A., Woodcock, J., Wei, K., Zeyda, F.: Safety-critical Java in Circus. In: JTRES '11, pp. 20–29. ACM (2011)
6. Cavalcanti, A., Zeyda, F., Wellings, A., Woodcock, J., Wei, K.: Safety-critical Java programs from Circus models. Real-Time Systems 49(5), 614–667 (2013)
7. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martı-Oliet, N., Meseguer, J., Quesada, J.F.: Maude: specification and programming in rewriting logic. Theoretical Computer Science 285(2), 187–243 (2002)
8. Dijkstra, E.W.: Guarded commands, nondeterminacy and formal derivation of programs. Communications of the ACM 18(8), 453–457 (1975)
9. Duran, A.: An Algebraic Approach to the Design of Compilers for Object-Oriented Languages. Ph.D. thesis, Universidade Federal de Pernambuco (2005)

10. Freitas, L., Baxter, J., Cavalcanti, A., Wellings, A.: Modelling and Verifying a Priority Scheduler for an SCJ Runtime Environment. In: iFM '16, pp. 63–78. Springer (2016)
11. Kalibera, T., Parizek, P., Malohlava, M., Schoeberl, M.: Exhaustive testing of safety critical java. In: JTRES '10. pp. 164–174. ACM (2010)
12. Klein, G., Nipkow, T.: A machine-checked model for a Java-like language, virtual machine, and compiler. ACM Transactions on Programming Languages and Systems 28(4), 619–695 (2006)
13. Leroy, X.: Formal verification of a realistic compiler. Communications of the ACM 52(7), 107–115 (2009)
14. Leroy, X.: A formally verified compiler back-end. Journal of Automated Reasoning 43(4), 363–446 (2009)
15. Lochbihler, A.: Verifying a compiler for Java threads. In: Gordon, A.D. (ed.) Programming languages and systems, pp. 427–447. Springer (2010)
16. Locke, D., et al.: Safety-Critical Java Technology Specification, `https://jcp.org/aboutJava/communityprocess/edr/jsr302/index2.html`
17. Marriott, C.: Checking Memory Safety of Level 1 Safety-Critical Java Programs using Static-Analysis without Annotations. Ph.D. thesis, University of York (2014)
18. Motor Industry Software Reliability Association Guidelines: Guidelines for Use of the C Language in Critical Systems, 2012
19. Nipkow, T., Paulson, L.C., Wenzel, M.: Isabelle/HOL: a proof assistant for higher-order logic, vol. 2283. Springer Science & Business Media (2002)
20. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for Circus. Formal Aspects of Computing 21(1-2), 3–32 (2009)
21. Perna, J.: A verified compiler for Handel-C. Ph.D. thesis, University of York (2010)
22. Pizlo, F., Ziarek, L., Vitek, J.: Real time Java on resource-constrained platforms with Fiji VM. In: JTRES '09. pp. 110–119. ACM (2009)
23. Proebsting, T.A., Townsend, G., Bridges, P., et al.: Toba: Java for applications a way ahead of time (wat) compiler. In: Proceedings of the 3rd Conference on Object-Oriented Technologies and Systems (1997)
24. Roscoe, A.W.: Understanding Concurrent Systems. Texts in Computer Science, Springer (2011)
25. Sampaio, A.: An algebraic approach to compiler design. World Scientific (1997)
26. Sawadpong, P., Allen, E.B., Williams, B.J.: Exception handling defects: An empirical study. In: HASE 2012. pp. 90–97. IEEE (2012)
27. Søndergaard, H., Korsholm, S.E., Ravn, A.P.: Safety-critical Java for low-end embedded platforms. In: JTRES '12. pp. 44–53. ACM (2012)
28. Stärk, R., Schmid, J., Börger, E.: Java and the Java Virtual Machine. Springer-Verlag (2001)
29. Strecker, M.: Formal verification of a Java compiler in Isabelle. In: Voronkov, A. (ed.) Automated Deduction — CADE-18, pp. 63–77. Springer (2002)
30. Varma, A., Bhattacharyya, S.S.: Java-through-C compilation: An enabling technology for Java in embedded systems. In: Proceedings of the conference on Design, automation and test in Europe-Volume 3. p. 30161. IEEE Computer Society (2004)
31. Woodcock, J., Davies, J.: Using Z: specification, refinement, and proof. Prentice-Hall, Inc. (1996)