An Approach to Verification of Safety-Critical Java Virtual Machines with Ahead-of-time Compilation

James Baxter
Abstract
In recent years Java has been increasingly considered as a language for safety-critical embedded systems. However, some features of Java are unsuitable for such systems and this has resulted in the creation of Safety-Critical Java (SCJ). The different scheduling and memory management model of SCJ means that a specialised virtual machine is required to run SCJ programs. Given the safety-critical nature of the applications, it must be ensured that the virtual machine is correct, but so far no SCJ virtual machine has been formally verified. In this dissertation, we propose a framework for verification of SCJ virtual machines. We consider the differences between SCJ and standard Java, and discuss some of the existing virtual machines for SCJ. Seeing that many SCJ virtual machines precompile to native code, we then survey some of the literature on compiler correctness. Finally, we present some preliminary results identifying the requirements of the services of an SCJ virtual machine and constructing a formal model of those requirements.
Contents

1 Introduction 4
  1.1 Motivation .............................................. 4
  1.2 Objectives .............................................. 5
    1.2.1 SCJ Virtual Machine Specification .................. 6
    1.2.2 Compilation Strategy ............................... 6
    1.2.3 Formal Model and Proofs ............................ 6
    1.2.4 Mechanisation ....................................... 7
    1.2.5 Summary ............................................ 7
  1.3 Document Structure ..................................... 7

2 Compilers and Virtual Machines for Java-like languages in the Safety-critical Domain 8
  2.1 Java for Safety-critical systems ........................ 8
  2.2 Safety-Critical Java ..................................... 9
  2.3 Virtual Machines for Safety-Critical Java ................ 11
    2.3.1 icecap HVM and HVMTP ............................... 11
    2.3.2 Fiji VM ............................................ 11
    2.3.3 OVM ................................................ 11
    2.3.4 PERC Pico .......................................... 11
  2.4 Compiler Correctness ................................... 12
    2.4.1 Commuting-diagram Approach ......................... 12
    2.4.2 Algebraic Approach ................................ 13
    2.4.3 Correctness of Java Compilers ...................... 15
  2.5 Circus .................................................. 16
  2.6 Final Considerations ................................... 19

3 Safety-Critical Java Virtual Machine Services 20
  3.1 Memory Manager API ..................................... 21
  3.2 Scheduler API .......................................... 23
  3.3 Real-time Clock API .................................... 25
  3.4 Formal Model ........................................... 26
    3.4.1 Memory Manager ..................................... 26
    3.4.2 Scheduler ........................................... 40
    3.4.3 Real-time Clock .................................... 61
    3.4.4 Complete VM Services Model ....................... 63
  3.5 Final Considerations ................................... 64

4 The Core Execution Environment 65
  4.1 Overview .............................................. 65
  4.2 Launcher ................................................. 69
  4.3 Bytecode Interpreter Model .............................. 73
    4.3.1 Bytecode Subset .................................... 74
    4.3.2 Classes ............................................. 77
    4.3.3 Object Manager ..................................... 79
    4.3.4 Interpreter ......................................... 81

1
Chapter 1

Introduction

This chapter begins by explaining the motivation for the work described in this dissertation. Afterwards, the objectives of the work, which come from the motivation, are described and, finally, the structure of the remainder of this dissertation is described.

1.1 Motivation

Since its release in 1995, the Java programming language \( [35] \) has increased in popularity and is now in use on many platforms. This popularity means that Java has been used in a wide variety of areas including desktop applications, on the internet in the form of Java applets, on smartcards \( [22] \) and on mobile devices \( [79] \). Several languages derived from Java have also been created, including Scala \( [30] \) and Ceylon \( [59] \), as well as older variants of Java such as MultiJava \( [23] \) and Pizza \( [77] \), which have in turn contributed to the development of Java. Scala adds functional programming features to Java, some of which have been incorporated into Java 8. Ceylon extends Java’s type system with features such as union types, allowing some common Java errors to be checked at compile time through the type system.

One use of Java that is of particular interest is in embedded systems. While early versions of Java were developed for programming embedded systems, particularly TV set-top boxes, the technology was not well received. It was only in the growing sector of the internet that Java initially found a market \( [43] \). However, it was soon realised that the portability, modularity, safety and security benefits of Java could be of great use in embedded systems \( [72] \). This required the creation of specialised Java virtual machines as the standard JVM is too large for most embedded systems. Much research has gone into making smaller and smaller virtual machines to widen the range of devices that Java can be used on \( [18] \) \( [98] \).

Many embedded systems are also real-time systems, and features of Java such as the garbage collector and the concurrency model make it unsuitable for real-time systems, for which strict guarantees about timing properties must be made. To address this issue the Real-Time Specification for Java (RTSJ) \( [34] \) was created. The RTSJ extends Java with a scoped memory model and a more predictable scheduling system.

While the RTSJ addresses real-time requirements of embedded systems, many embedded systems are also safety-critical. For these conformance to certain standards, such as DO-178C and ISO 26262, is required. To support the development of safety-critical programs that meet these requirements in Java, the Safety-Critical Java (SCJ) specification \( [60] \) has been created. SCJ is a subset of the RTSJ that removes the features that cannot be easily statically reasoned about, which means that features such as the garbage-collected heap and dynamic class loading are absent from SCJ. This facilitates the creation of SCJ programs that fulfil formal specifications; indeed work has already been done on developing correct SCJ programs from formal specifications \( [20] \) \( [21] \).

On the other hand, even if it can be shown that SCJ programs are correct, it must still be ensured those programs are executed correctly. In the case of Java-like languages, this generally means ensuring the
Java compiler and Java Virtual Machine (JVM) are correct.

Work has been done 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 [95], who present a model of the full Java language and virtual machine, along with a formally verified compiler, although for an older version of Java than is current. Other work has also been done on modelling the JVM and Java compilation using refinement techniques [29]. Additionally there has been work considering machine checked models of Java virtual machines and compilers [50, 74, 96]. Work has also been done on the semantics of Java bytecode and verification of standard JVMs [12, 44].

However, SCJ has a number of differences from standard Java. Firstly, as already indicated the SCJ memory model is rather different to the standard Java memory model, abandoning the garbage collector in favour of a scoped memory model. Garbage collection is less predictable and often quite complex, and so difficult to reason about and 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. Similarly, the SCJ approach to scheduling differs from that of standard Java, using a preemptive priority scheduling approach rather than the unpredictable scheduling of standard Java threads. These differences of SCJ from standard Java mean that the standard JVM is not suitable for running SCJ programs. A specialised virtual machine is required.

In the case of virtual machines for embedded systems, the priorities are usually size and speed, which generally results in machines that are hard to verify. Moreover, virtual machines that rely on interpreting bytecode are unsuitable for real-time embedded systems as they are likely to be slower. An alternative method to run a Java program is to compile it to native code and some authors have suggested doing so either directly [92] or via C [99]. There are several virtual machines that take this approach including Fiji VM [82], Icecap HVM [93] and OVM [2]. This allows correct running of an otherwise correct SCJ program to be viewed as a compiler verification problem.

There has been much research into compiler correctness. Much of the work follows a commuting diagram approach, in which the compilation is shown to be consistent with transformation between the semantics of the source and target languages [69, 97]. This approach is apparent in much of the early work such as that of McCarthy and Painter [65], as well as in more recent work such as the CompCert project [52, 53]. There has also been work that follows this approach and employs automated theorem provers [47, 67, 74]. They provide additional certainty that the proof is correct and can also provide code generation facilities to allow creation of a working compiler.

An alternative is the algebraic approach to compiler verification [38, 88], based on modelling compilation using refinement calculi [5, 68, 70]. This approach appears to be less commonly used but has been applied to Java [28, 29] and hardware description languages [80, 81]. This approach is also quite amenable to automation as it relies on refinement laws that can be applied by a term rewriting system.

There is a clear need for formal verification of SCJ virtual machines due to the safety-critical nature of the systems involved and the fact that safety standards such as DO-178C require it at the highest safety levels. However, there appears to be little work done in that area and, as far as we know, no SCJ virtual machine has been formally verified.

1.2 Objectives

Our objective is to develop an approach to verification of an SCJ virtual machine that allows the production and verification of correct SCJ virtual machines. Although the actual creation and verification of such machines is outside the scope of our work, we provide the following resources for developers and verifiers:

- a specification of the requirements of an SCJ virtual machine,
- a formal model of the virtual machine specification,
- a compilation strategy from Java bytecode to native C code,
• proofs for validation of the formal model and verification of the compilation strategy, and
• a mechanisation of the model and proofs.

We follow the design of existing SCJ virtual machines to ensure that our work is of practical relevance to the SCJ community. We particularly focus on the icecap HVM [93], as that is the only publicly-available SCJ virtual machine that is up-to-date with the SCJ specification. Where there are ambiguities or concerns regarding the description of the virtual machine in the SCJ standard, we take the icecap implementation as a reference to define the requirements and formal model for an SCJ virtual machine. In addition, the native C code generated by our formal compilation strategy is very close to that actually produced by icecap.

Our results can be used to aid the development and verification of an SCJ virtual machine in several different ways. The informal specification provides a reference for the requirements of an SCJ virtual machine.

1.2.1 SCJ Virtual Machine Specification

The first component required is a specification of the requirements for an SCJ virtual machine. This specification will shape the rest of the work and there is at present no clear specification of what is required of an SCJ virtual machine or how it differs from a standard Java virtual machine. The specification of requirements needs to consider the requirements imposed, both explicitly and implicitly, on virtual machines by the SCJ specification [60] as that provides the authoritative source for information on SCJ. It is also helpful to consider the approach taken by some existing SCJ virtual machines on points where the SCJ specification is unclear. The virtual machine must also meet the standard Java Virtual Machine specification [57] on points such as how to interpret Java bytecode instructions. There is much existing work on the semantics of Java bytecode that can be used in our work [12, 44, 95].

1.2.2 Compilation Strategy

As many existing virtual machines for SCJ precompile programs to native code in order to allow faster execution on embedded systems, it seems wise to include that in our approach. We will focus on compilation of Java bytecode to C as that is the approach adopted by several existing virtual machines for embedded systems, including Fiji VM [82] and icecap HVM [93], and C is already widely used for embedded systems software.

There are two main approaches to the specification and verification of compilers: the commuting diagram approach and the algebraic approach. The commuting diagram approach involves specifying the compiler as a function from the source language to the target language and showing that it is consistent with transformation between the semantics of the source and target languages [69, 97]. This approach has been used in much of the work on compiler correctness, including some of the earliest work [65] and recent work such as that of the CompCert project [52, 53].

The algebraic approach involves defining the source and target languages in the same specification space, and using proved specialised rewrite rules to characterise compilation as model transformation in the extended language. This approach was first proposed in the early nineties by Hoare [38] and further developed by Sampaio [40, 88]. The algebraic approach does not seem to be as popular as the commuting diagram approach, but it does have the advantage that the specification of the compilation strategy is correct by construction as the rewrite rules that comprise it have all been proved.

1.2.3 Formal Model and Proofs

In order to ensure that the specification is precise and to facilitate proofs of its correctness, it must have a model written in some formal specification language. We will focus on using the Circus specification language [78], as it has been used in some of the previous work on formalising SCJ [20, 21]. It is important that the correctness of the formal models and compilation strategy can be shown via mathematical proof,
which requires the specification language to have a well-defined semantics. Circus has such a semantics, defined using the model of Unifying Theories of Programming (UTP) [41].

1.2.4 Mechanisation

To prevent mistakes in the proofs, it is helpful to mechanise the formal model and proofs. There are various systems that can be used for this, but we will focus on the proof assistant Isabelle [75, 76]. It has existing mechanisations of Circus [31] and the UTP [32], and has been used in previous work verifying compilers for Java-like languages [47, 58, 96], making it well placed for our work.

1.2.5 Summary

In conclusion, our objective is an approach verification of SCJVMs consisting of mechanised formal models together with proofs of properties about them. These formal models will cover both the services that must be provided by a running SCJVM and a compilation strategy for translating Java bytecode to native code. With our results, SCJVM developers will be able to create provably correct ahead-of-time compiling SCJVM implementations and check the correctness of those implementations.

1.3 Document Structure

Having given a brief overview of the area of study and identified the problem we wish to consider, the remainder of this dissertation proceeds as follows.

In Chapter 2 we examine the literature on safety-critical virtual machines and compilers for Java-like languages. This includes a discussion of why a safety-critical variant of Java is necessary and how it differs from standard Java. We also explain why a specialised virtual machine is necessary for SCJ. This is followed by a survey of the existing virtual machines for Safety-Critical Java and the techniques used in verifying compilers.

In Chapter 3 we present an identification of the requirements of SCJ virtual machine services, with a formal model of those requirements in the Circus specification language. This is followed by a model of the an SCJ virtual machine core execution environment in Chapter 4.

Finally, we conclude in Chapter 6 by summarising our contributions and mentioning the wider context of this research.
Chapter 2

Compilers and Virtual Machines for Java-like languages in the Safety-critical Domain

This chapter begins with a discussion of why Java is being used in safety-critical systems and the need for a specialised version of Java for use in that area. Then, in Section 2.2 we cover the variant of Java developed for safety-critical systems, how it differs from standard Java and why a specialised virtual machine is required, before discussing some of the existing virtual machines for that variant in Section 2.3.

In Section 2.4 we survey some of the literature on compiler correctness, and discuss the two main approaches in Sections 2.4.1 and 2.4.2 before seeing how the techniques of compiler correctness have been applied to Java-like languages in Section 2.4.3.

In Section 2.5 we give an overview of the Circus specification language used for our virtual machine specification, before concluding in Section 2.6.

2.1 Java for Safety-critical systems

In recent years Java has increasingly been considered as a language for writing safety-critical software. Other languages that are generally used in the safety-critical domain are C/C++ and Ada; C and C++ impose challenges concerning reliable use at the highest levels of safety [19], and the number of Ada programmers is not very large [13]. While Java has not traditionally been seen as a language for safety-critical systems, it was originally developed for the area of embedded systems, particularly for use in television set-top boxes, and has seen renewed interest in its use in embedded systems after gaining popularity in programming for the internet [72].

There are, however, several issues with standard Java that make it unsuitable for safety-critical systems. Many safety critical systems are also real-time systems, which are required to be predictable in their scheduling and use of memory. However, standard Java uses a garbage-collected memory model, which makes it hard to predict when memory may be freed or how long the process of freeing memory may take. Standard Java’s thread model also lacks the predictability and control that is required in real-time systems.

To rectify these problems the Real-Time Specification for Java (RTSJ) [34] was created; it augments Java’s memory and scheduling models with a system of scoped memory areas and a preemptive priority scheduler. RTSJ also allows for the standard Java models to be used alongside its own, making it suitable for a wide range of different real-time applications. On the other hand, this makes it hard to certify RTSJ applications and thus renders the RTSJ unsuitable for use in the safety-critical domain.
In order to allow certifiable safety-critical systems in Java, the Safety-Critical Java (SCJ) specification was developed. SCJ is a subset of the RTSJ that leaves out the features from standard Java that are difficult to certify such as the garbage collector. SCJ also provides annotations that allow memory usage to be more easily checked. We discuss SCJ in more detail in the next section.

2.2 Safety-Critical Java

SCJ removes the aspects of the RTSJ that make certification difficult, including standard Java threads and the garbage collector. This leaves 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.

SCJ 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 \texttt{wait()} and \texttt{notify()}.

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 a mission sequencer supplied by an SCJ program. Running a mission proceeds in several phases, as shown in Figure 2.1.

![Mission Sequencer Diagram](image)

Figure 2.1: A diagram showing the phases of SCJ mission execution

The fist phase is initialisation, which consists of setting up the schedulable objects controlled by the mission and creating any data structures required for the mission. Then the mission is executed by starting each of the schedulable objects in the mission and waiting for a request to terminate the mission. When the mission is requested to terminate, each of the schedulable objects in the mission is terminated and the mission’s memory is cleared.

The schedulable objects within a mission are asynchronous event handlers that are released either periodically, at set intervals of time, aperiodically, in response to a 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). At level 2 real-time threads are also allowed, which run continuously from when they start until they finish, unless they are suspended or interrupted by another schedulable object.

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 and allows for more urgent tasks to preempt less urgent tasks.

SCJ allows for assigning schedulable objects to “scheduling allocation domains”, where each domain consists of one or more processors. At Level 1, each scheduling allocation 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 synchronised methods are different than they would be on a single processor system due to the potential for remote blocking).

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 allocation context. Each schedulable object has a memory area associated with it that is used as the 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 need to persist for the duration of the mission or to be shared between the schedulable objects. The amount of memory required for the mission memory must be computed ahead of time and specified by the programmer as part of writing the mission, though there has been some work on automated computation of worst case memory use for SCJ programs [1]. There is also an immortal memory area where objects can be allocated if needed for the entire running of the program (they are never freed). SCJ places restrictions on which objects an object may point to, so as to avoid dangling pointers from being created. Some examples of valid and invalid object references for some asynchronous event handlers are shown in Figure 2.2.

![Memory Layout Diagram](memory_layout.png)

Figure 2.2: An example of the layout of memory areas for four asynchronous event handlers (ASEHs), showing possible valid and invalid references between them.

This system of memory areas makes it easy to predict when memory is freed. It is not supported by standard JVMs as they do not provide memory outside of the heap for allocation and lack a notion of allocation context. The SCJ 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.

Moreover, dynamic class loading is not allowed in SCJ; 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. Finally, SCJ also disallows object finalisers as it is not always easy to predict when they are run.
2.3 Virtual Machines for Safety-Critical Java

Because of the novel features of SCJ, briefly described in the previous section, a specialised virtual machine that provides support for allocation in memory areas and preemptive scheduling is required for SCJ. Although SCJ is a relatively recent development there have been various virtual machines created for SCJ or variations of SCJ, including icecap HVM [93], Fiji VM [82], OVM [2], HVMTP [62] and PERC Pico [4, 86]. These are each described in the following subsections.

2.3.1 icecap HVM and HVMTP

The icecap hardware-near virtual machine (HVM) was created as part of the Certifiable Java for Embedded Systems Project [91] and provides an open-source implementation of SCJ targeted at embedded systems. The approach taken by the HVM is one of precompiling Java bytecode to C in order to allow for faster running programs with fewer memory resources. It includes an implementation of the SCJ libraries that covers most of SCJ level 2, though only for a single processor implementation. This implementation, however, cannot be easily decoupled from the virtual machine itself.

The icecap HVM also provides a lightweight Java bytecode interpreter and allows for interpreted code to be mixed with compiled code. The reason for this is that the bytecode together with the interpreter can often be smaller than the compiled code, though there is a tradeoff for speed. HVMTP is a modification of the icecap HVM’s bytecode interpreter to improve time predictability and ensure that bytecode instructions are executed in constant time, which is important for ensuring real-time properties of the system hold.

2.3.2 Fiji VM

Fiji VM is a proprietary Java implementation designed to run on real-time embedded systems. Similarly to the icecap HVM, Fiji VM uses the strategy of compiling to C in order to improve performance. However, Fiji VM is not specifically targeted at SCJ and works with a range of libraries, including SCJ, RTSJ and the standard Java libraries. Fiji VM does have the advantage of high portability and multiprocessor support, which is lacking in many other SCJ virtual machines.

The fact that Fiji VM works with the SCJ libraries and supports the scoped memory model means it can run SCJ programs. It does not necessarily support all aspects of SCJ properly though.

2.3.3 OVM

OVM was created at Purdue University as part of the PCES project [7], to provide a virtual machine that can execute real-time Java programs with a high level of performance on embedded systems. Similar to Fiji VM and icecap HVM, OVM follows the principle of precompiling code for performance reasons, but translates Java to C++ instead of bytecode to C.

OVM also differs from the icecap HVM and Fiji VM in that it predates SCJ. It is written to implement the RTSJ, though it can still support SCJ programs; indeed, an SCJ implementation for OVM was later created [83]. However, OVM does not appear to have kept up with more recent changes to the draft SCJ standard. OVM is, like icecap HVM, but unlike Fiji VM, single processor.

2.3.4 PERC Pico

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. This systems of annotations provides additional information about how memory is used so that it can be checked. Similarly to other SCJ virtual machines, PERC Pico allows for precompilation of Java code but targets executable machine code rather
than an intermediate programming language. The metadata annotations are used to guide the compiler to produce code that uses the correct scoped memory. PERC Pico does not support the current SCJ standard, though it has been suggested that it could be modified to do so [73].

To summarise, as far as we are aware there is one publicly available virtual machine that has kept up with the developing SCJ specification, the icecap HVM. This 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.

As can be seen from the preceding discussion, a common technique to run Java programs on embedded systems is to precompile them to native code. This means compiler correctness techniques must be considered in verification of such a virtual machine; these techniques are discussed in the next section.

2.4 Compiler Correctness

Due to the importance of compiler correctness, there has been much research over the years in this area. Most of the work done follows a similar approach, which we will term the commuting-diagram approach as it is based on showing that a particular diagram commutes. We will discuss the commuting diagram approach in Section 2.4.1.

An alternative approach to compiler verification is the algebraic approach developed in the early 90s. It is based on the concepts of refinement calculi designed for deriving software from specifications of behaviour. We will explain the algebraic approach in Section 2.4.2 and discuss how it differs from the commuting-diagram approach.

We finish in Section 2.4.3 by reviewing some of the literature on correctness of compilers for Java-like languages. We explain how the techniques of compiler correctness have been applied in the case of Java and compare the different approaches.

2.4.1 Commuting-diagram Approach

Much of the work on compiler correctness can be seen as following the approach identified by Lockwood Morris [69], and later refined by Thatcher, Wagner and Wright [97]. The approach is essentially that a compiler correctness proof is a proof that the diagram shown in Figure 2.3 commutes, that is, $\gamma \circ \psi = \phi \circ \epsilon$.

![Figure 2.3: The commuting diagram used in the traditional approach to compiler verification](image)

Lockwood Morris had the corners of the diagram as algebras, rather than merely sets, with the functions between them being homomorphisms in order to add additional structure to the proof. This differs from the approach of some earlier works, particularly the earliest work by McCarthy and Painter [65], and instead follows work such as that of Burstall and Landin [16].

McCarthy and Painter’s work featured a simple expression language with addition, natural numbers and variables. This was compiled to a simple 4-instruction single-register machine. The arrows of the diagram were simple functions, rather than homomorphisms, and the proof was performed using induction over the source language. This work laid the foundation for the study of compiler correctness.
Burstall and Landin show correctness of a compiler for the same source and target languages as McCarthy and Painter; they use a more algebraic approach that better matches what Lockwood Morris later suggested. Burstall and Landin’s approach involved representing the source and target languages, and their meanings, as algebras, with the compilation functions as homomorphisms. They target several intermediate machines in the proof of correctness. Viewing the languages as algebras allows for simpler proofs as some of the arrows of the commuting diagram can be wholly or partially derived from the algebraic structure. It was this goal of simplifying the proofs that led Lockwood Morris to advocate the use of algebras and homomorphisms.

The overall goal of pursuing formal proofs of compiler correctness, as proposed by McCarthy and Painter [65], is to allow machine-checked proofs of program correctness. There has been work in that area, the earliest of which is that by Milner and Weyhrauch [67] who show the correctness of an ALGOL-like language. The proof of correctness was partially mechanised in the LCF theorem prover [66] and the authors were of the opinion that the proof was feasible and could be completed relatively easily. A point to note is that Milner and Weyhrauch acknowledged the need for some way of structuring the proof in order to make it amenable to machine-checking. This gives further support to the algebraic commuting diagram approach advocated by Lockwood Morris. Indeed, Milner and Weyhrauch explicitly followed that approach as they were in discussions with Lockwood Morris.

One advantage to making proofs easily machine-checkable, apart from the added certainty that the proof is correct, is that working compilers can be created from the machine-checked proofs. Code generation facilities are available with many theorem provers such as those of Isabelle/HOL [56] and Coq [55, 56]. The fact that the commuting-diagram approach involves treating the compilation as a function between algebras representing the source and target languages fits well with this idea. In this case, there is then a function defined in the mechanised logic for the purposes of conducting proofs about it that can be readily extracted to executable code.

The commuting-diagram approach has been followed in much of the literature through the years, though not always with the algebraic methods recommended by Lockwood Morris. The basic structure of the commuting diagram is a fairly natural approach to take, as seen by work such as that of the ProCoS project [17].

Another piece of work that follows the commuting diagram approach is that of Polak [84], who states that he is more interested in verification of a “real” compiler rather than “abstract code generating algorithms”, and shows the correctness of a compiler for a Pascal-like language. This work focuses much more on pragmatic applications of the commuting-diagram approach, leaving behind the algebraic ideas of earlier papers. It sets a precedent for a simpler verification approach based on considering the functions in the commuting diagram.

The commuting diagram has also been used in recent work, some of the most successful of which is that of CompCert [52–54]. This is a project to create a fully verified realistic compiler for a subset of C, using the theorem prover Coq [64].

There is also recent variation of the commuting-diagram approach, based on an operational semantics of the source language [6]. In this work, the operational semantics of the source language and a way of relating the source and target semantics are used to derive a different operational semantics of the source language acting on the state of the target machine. The semantics of the target language are then identified as part of that operational semantics and it is transformed to extract a compilation function. This approach may be viewed as variant of the commuting-diagram approach in which the compilation function is derived from the source and target semantics and the relationship between them, rather than being verified by those elements of the commuting-diagram.

2.4.2 Algebraic Approach

The second main approach to showing correctness of compilers is the algebraic approach proposed by Hoare in 1991 [38], and further developed by Sampaio [40, 88, 89]. We note that the algebraic approach discussed in this section is largely unrelated to the algebraic commuting-diagram approaches mentioned in the previous section.
The algebraic approach to compilation derives from the concepts of algebraic reasoning about programs and program refinement. These concepts come from the idea, proposed by Hoare in 1984 [39], that programs can be thought of as predicates and so the laws of predicate logic can be used to construct laws for reasoning about programs [42]. As an example of such a law for reasoning about programs, we present below associativity of sequential composition, Equation (2.1), and left and right unit of sequential composition, namely, the program \textit{Skip} that does nothing, Equation (2.2).

\[ P; (Q; R) = (P; Q); R \quad (2.1) \]

\[ P; \textbf{Skip} = \textbf{Skip}; P = P \quad (2.2) \]

The notion of refinement is central to the algebraic approach to compilation. Refinement calculi have been developed, independently, by Back [5], Morris [70] and Morgan [68], following from earlier concepts of program transformation \cite{3,8,9,94}. The basic idea is that there is a relation between programs that captures the idea of one program being “at least as good as” another or, to put it more precisely, at least as deterministic as another. Languages and laws for reasoning about programs with this notion of refinement can then be used to develop programs from specifications. This means that certain aspects of a system can have a nondeterministic specification and several different implementations can refine that specification.

In using refinement to show the correctness of a compiler, the laws of the specification language can be used to prove compilation refinement laws. These compilation laws can be used to transform the source programs into some normal form that represents an interpreter for the target language running the target code. In other words, the code output by the compiler, when executed by on the target machine, must be a refinement of the source program. The compilation laws can be used to prove this refinement and at the same time generate the target code.

As an example, consider the following refinement in which a simple program that performs some arithmetic and stores the results into variables is refined by a normal form representing the target machine and code. The symbol \(\sqsubseteq\) represents the refinement relation here.

\[ \text{var} \ x, \ y, \ z \bullet x := (x + 5) \times (y + z); \ z := z + 1 \sqsubseteq \]

\begin{align*}
\text{var} \ A, \ P, \ M \bullet P := 1; \ &\text{do} \\
& P = 1 \rightarrow A, P := M[2], 2 \\
& \square P = 2 \rightarrow A, P := A + M[3], 3 \\
& \square P = 3 \rightarrow M[4], P := A, 4 \\
& \square P = 4 \rightarrow A, P := M[1], 5 \\
& \square P = 5 \rightarrow A, P := A + 5, 6 \\
& \square P = 6 \rightarrow A, P := A \times M[4], 7 \\
& \square P = 7 \rightarrow M[1], P := A, 8 \\
& \square P = 8 \rightarrow A, P := M[3], 9 \\
& \square P = 9 \rightarrow A, P := A + 1, 10 \\
& \square P = 10 \rightarrow M[3] := A, 11 \\
& \text{od}; \{P = 11\}
\end{align*}

The normal form represents the behaviour of an interpreter for the target code running in a target machine whose structure is defined by the variables \(A, P,\) and \(M\). The variable \(A\) represents a general-purpose register of the target machine, \(P\) represents the program counter of the target machine, and \(M\) is an array representing the memory of the target machine. The normal form consists of a program that initialises \(P\) to 1 and then enters a loop in which the operation performed on each iteration is dependent on the value of \(P\). The loop is exited when \(P\) is set to a value for which there is no operation and it is asserted that \(P\) will be equal to 11 at the end of the program. Each of the statements of the source
program corresponds to several operations in the normal form as complex expressions are broken down into simpler expressions that can be handled by instructions of the target machine.

The compilation proceeds by first applying rules to simplify the assignment statements. The register $A$ is introduced at this stage by splitting assignments of expressions to variables into two assignments that transfer the values to and from $A$. In this way, the assignments are transformed for the target machine that only has instructions involving registers. Particularly complex expressions such as $(x+5) \times (y+z)$ are handled by storing intermediate results in temporary variables. In this case the result of the expression $y+z$ is placed in a temporary variable when $P=3$. The variables used in the source program and introduced compilation are later replaced with locations in the memory array $M$ in a data refinement step. This causes the variables $x$, $y$ and $z$ to be replaced with $M[1]$, $M[2]$ and $M[3]$ respectively. The temporary variable introduced to store the result of $y+z$ is similarly replaced with $M[4]$.

Each of the assignment statements from is then refined by a normal form with an explicit program counter $P$, that is incremented as part of the assignment operation. These normal forms are then combined together by the refinement rule for sequential composition to create the normal form of the full program. The update of the program counter in this program is quite simple but more complex updates would occur for conditionals or loops.

The power of the algebraic approach is that the compilation of individual elements of the source language can be specified and proved separately in different refinement laws. The compilation can also be split into stages, with a set of refinement laws for each stage to modularise the compilation. The separate refinement laws can then be combined to form a compilation strategy.

The first major work done using the algebraic approach was that of Sampaio [88], who used it to specify a correct compiler for a simple language that, nonetheless, covers all the constructs available in most programming languages. The target machine Sampaio used was a simple single-register machine that bears similarity to most real processor architectures. He mechanised the compiler in the OBJ3 term rewriting system [33], showing that working compilers can be easily created from specifications using the algebraic approach. However, the algebraic laws Sampaio used to prove correctness of the compiler were taken as axioms. Sampaio notes that they could be easily proved given a semantics for the reasoning language.

Though there has not been much work done using the algebraic approach, we single out the work of Perna [80, 81], showing correctness of a compiler for a hardware description language. The compilation takes high-level descriptions of hardware written in Handel-C and transforms them into systems of basic hardware components connected by wires. The algebraic approach works well here as the target language is a subset of the source language, albeit in a different form. Perna was able to handle features not covered by most other works on hardware compilation, such as parallelism with shared variables. Also, whereas Sampaio took the basic algebraic laws as axioms, Perna proved the laws from a semantics given using the Unifying Theories of Programming (UTP) model [41]. There has also been work on the correctness of Java compilers using the algebraic approach. This is considered in the next section, where we consider compiler correctness for Java-like languages.

### 2.4.3 Correctness of Java Compilers

The popularity of Java has meant that there has been plenty of work on formalising Java and the JVM [37], but there have been relatively few works on formally verified compilers for Java-like languages. However, the work that has been done uses both of the two main approaches and covers most of the features of Java.

Some of the earliest and most thorough work is that by Särk, Schmid and Börger [95], who formalise most of Java and the JVM before specifying and showing the correctness of a compiler for Java. The approach taken by them uses Abstract State Machines (ASMs) to specify the source and target languages. The ASMs give an operational semantics to Java and the JVM, describing how each construct affects the running of the program. The languages are each specified by multiple ASMs, beginning with an imperative core, then adding classes, objects, exceptions and, finally, threads.
Although this approach is called the ASM approach, it becomes clear from the definition of compiler correctness given in terms of a mapping between ASMs that this work ultimately follows the commuting-diagram approach. This work leaves parts of the proof incomplete (in particular, compilation of threads is not addressed) and applies to an old version of Java. This is, nevertheless, an admirable attempt at producing a verified Java compiler.

Work has also been done by Duran following the algebraic approach [28, 29]. Duran’s work specifies a compiler for a language called Refinement Object-Oriented Language (ROOL) [15], which was created for reasoning about object-oriented languages and bears much similarity to Java. ROOL features constructs for specifying and reasoning about programs as well as object-oriented programming language constructs. This means that there are algebraic laws for ROOL, from which the rewrite rules that form the basis of the algebraic approach can be proved. Duran’s work adds further phases to Sampaio’s compilation strategy in order to deal with the object-oriented features, but does not consider some other aspects of Java such as exceptions and threads. Duran notes that other work has addressed some of those issues.

While the two works already discussed were not machine checked, there have also been compiler correctness proofs for Java-like languages in the Isabelle/HOL proof assistant. The first of these was by Strecker [96], showing correctness of a compiler for a subset of Java called μJava, which already had a formalisation of its semantics in Isabelle/HOL [74]. This work was followed by Klein and Nipkow’s work on a compiler for a slightly larger subset of Java called Jinja [47], which added exception handling. Finally, Lochbihler [58] added threads to Jinja and showed correctness of compilation for Java concurrency. It is notable that this is the only work on Java compilation that properly addresses concurrency. All of these works follow the commuting diagram approach.

Though some work has been done on correct compilers for Java-like languages and many virtual machines for SCJ adopt an approach of compiling to native code, no work has been done on verifying that compilation to native code. Therefore, we will consider correctness of the compilation to native code as part of our work on SCJ virtual machines. We will follow the algebraic approach as it gives greater assurance of correctness, as an additional function mapping source meanings to target meanings is not required, and a good level of modularity, as the compilation is split into separately proved rewrite rules.

In order to represent the normal form we require a specification language and for that purpose will use Circus, which is described in the next section.

2.5 Circus

The Circus specification language [78] is based on CSP [87], which is used to specify processes that communicate over channels, and the Z notation [100], which is used to specify state and data operations. 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.

We illustrate the concepts of Circus using as an example the process for the real-time clock from some of the preliminary work on the specification of an SCJ virtual machine. The specification begins with a declaration of the channels that may be used in the following processes. Type declarations written in Z can also be included at the beginning of a Circus specification. Here, we define a type Time to be the set of natural numbers and create a boolean datatype

\[
\text{Time} ::= \mathbb{N} \\
\text{Bool} ::= \text{True} \mid \text{False}
\]

We declare channels to represent interactions corresponding to calls to methods to get the clock’s time and precision, and set and clear alarms. Channels are also declared to model interactions with the
hardware that accept clock tick interrupts and read the time from the hardware clock.

\[
\text{channel}\ getTime,\ getPrecision,\ setAlarm :\ Time \\
\text{channel}\ clearAlarm \\
\text{channel}\ HWtick \\
\text{channel}\ HWtime :\ Time
\]

We also specify a constant to represent the clock’s precision using a Z axiomatic definition. The value of the constant is required to be nonzero, but is otherwise left unrestricted, so that any nonzero time value is a valid instantiation.

\[
\begin{align*}
\text{precision} & :\ Time \\
\text{precision} & > 0
\end{align*}
\]

After the channel declarations, we can declare processes that use them. Here we declare the \textit{RealtimeClock} process. It is a basic process, that is, its state is defined in Z, and its behaviour using CSP constructs and Z data operations.

\[
\text{process}\ \textit{RealtimeClock} \triangleq \text{begin}
\]

In this example, the state records the current time, whether an alarm is set, and the time of the alarm that may be set. An invariant specifies that if an alarm is set, then the time of the alarm must not be in the past.

\[
\begin{align*}
\text{RTCState} \\
\text{currentTime} & :\ Time \\
\text{alarmSet} & :\ \text{Bool} \\
\text{currentAlarm} & :\ Time
\end{align*}
\]

\[
\begin{align*}
\text{alarmSet} = \text{True} \Rightarrow \\
\text{currentAlarm} & \geq \text{currentTime}
\end{align*}
\]

The behaviour is described using actions, written in a mixture of Z and CSP. The first action is a Z initialisation operation, \textit{Init0}. Its final state is represented by variables obtained by placing a prime on the names of the state components. Here, the initialisation takes as input the initial time, represented by the variable \textit{initTime}? . The current time is defined to be equal to the initial time and no alarm is initially set. The initial time of the alarm is arbitrary, that is, nondeterministically chosen from elements of its type, since the initialisation imposes no restrictions on it.

\[
\begin{align*}
\text{Init0} \\
\text{RTCState}' \\
\text{initTime}? :\ Time
\end{align*}
\]

\[
\begin{align*}
\text{currentTime}' & = \text{initTime}? \\
\text{alarmSet}' & = \text{False}
\end{align*}
\]

The action \textit{Init}, defined below, uses a CSP prefixing to specify an input communication before the initialisation operation \textit{Init0}. The initial time of the clock is read from the hardware clock and then the initialisation specified by the Z schema is performed.

\[
\text{Init} \triangleq \text{HWtime}?\text{initTime} \rightarrow \text{Init0}
\]
The action that returns the current time simply uses CSP to output the current time from the state over the `getTime` channel. The action ends with the special action `Skip`, which indicates the end of an action.

\[
\text{GetTime} \triangleq \text{getTime}! \text{currentTime} \rightarrow \text{Skip}
\]

Setting a new alarm is a more complex operation that involves Z schemas that specify two different scenarios in which this operation may be used. In the first case, the new alarm is not in the past. The symbol \(\Delta\) denotes a change of state. The operation stores the time of the new alarm and sets a flag to indicate an alarm is set in this case.

\[
\text{SetAlarm}^0 \triangleq \Delta \text{RTCState} \quad \text{newAlarm}^? : \text{Time}
\]
\[
\text{newAlarm}^? \geq \text{currentTime}
\]
\[
\text{currentAlarm}' = \text{newAlarm}^?
\]
\[
\text{alarmSet}' = \text{True}
\]
\[
\text{currentTime}' = \text{currentTime}
\]

In the second case, the new alarm is in the past and so the alarm is not set (we have omitted the error reporting for the sake of simplicity). The symbol \(\Xi\) denotes that the state remains the same.

\[
\text{SetAlarm}^1 \triangleq \Xi \text{RTCState} \quad \text{newAlarm}^? : \text{Time}
\]
\[
\text{newAlarm}^? < \text{currentTime}
\]

The two Z schemas are combined using a logical disjunction, allowing either to specify the behaviour when a request to set the alarm takes place.

\[
\text{SetAlarm} \triangleq \text{setAlarm}^? \text{newAlarm} \rightarrow (\text{SetAlarm}^0 \lor \text{SetAlarm}^1)
\]

In addition to Z and CSP constructs, Circus also has other constructs more familiar to programmers, such as if statements and do loops. One of these constructs, the assignment operator, is used in the action that clears the current alarm to update part of the state without requiring a Z schema. The alarm is cleared by simply setting `alarmSet` to `False`, without updating any other state variables.

\[
\text{ClearAlarm} \triangleq \text{clearAlarm} \rightarrow \text{alarmSet} := \text{False}
\]

Each of the actions the process can perform are joined together with the CSP external choice operator, which chooses an action to take based on the channel communications that the environment is willing to perform. This includes the actions above, as well as some other actions that have been omitted here. The choice is repeated in a loop.

\[
\text{Loop} \triangleq (\text{GetTime} \sqcap \text{SetAlarm} \sqcap \text{ClearAlarm} \sqcap \cdots) ; \text{Loop}
\]

The Circus process then ends with the main action that specifies the overall behaviour of the process. Here, the process simply performs the initialisation and then enters the loop.

- `Init ; Loop`
In addition to the constructs presented here, Circus also contains operators for composing processes in parallel, with or without synchronisation on channels. These operators are used both to specify actual parallelism and to represent composition of requirements. In this way, several Circus specifications of individual components can be combined to form a specification of the entire system.

A detailed account of Circus can be found in [78]. In what follows, we explain the notation as needed.

2.6 Final Considerations

We have seen that Java is increasingly being considered as a language for safety-critical embedded systems and that the modifications to Java required to make it suitable for such systems require a specialised virtual machine. The developing Safety-Critical Java specification has several differences from standard Java, particularly in the areas of scheduling and memory management, that make standard JVMs unsuitable for running SCJ programs. We have considered several virtual machines that have been developed for running SCJ programs and noted that none of them has been formally verified and that most of them adopt an approach of precompiling programs to native code.

With that in mind, we have considered the techniques used to verify the correctness of compilers and found that there are two main approaches: the commuting-diagram approach and the algebraic approach. In the commuting-diagram approach the source semantics, target semantics, compilation function, and a function mapping the source meanings to the target meanings, are shown to commute. This approach is popular and has had much research done on it but relies on the definition of the function from the source meanings to the target meanings.

The algebraic approach defines the source and target languages within the same specification language, which is additionally equipped with a refinement relation between programs. Laws of the specification language are then used to prove refinement rules that are applied according to some compilation strategy. The algebraic approach has the advantage that it does not require the additional function that is required in the commuting-diagram approach, since the source and target languages are defined in terms of the same specification language. The algebraic approach also permits a modular approach to proof and allows for the compiler to be easily implemented by application of the refinement rules using a term rewriting system.

Given the considerations above, we have decided to adopt the algebraic approach when specifying the compilation to native code employed by many SCJ virtual machines. This means that a specification language is required in which to define the source and target languages, as well as for the purposes of specifying other aspects of the virtual machine. We have chosen Circus as the specification language as it contains a wide variety of constructs that allow for specification of both data and behaviour, has a well-defined semantics with many laws already proved, and has been used for previous work on the specification of SCJ programs. Circus also has some existing mechanisation and tool support, which can help give greater assurance of the correctness of specifications.
Chapter 3

Safety-Critical Java Virtual Machine Services

In order to reason about a Safety-Critical Java virtual machine (SCJVM), we first require an identification of the requirements of an SCJVM and a formal model of those requirements. For the purposes of our model, we consider an SCJVM to have the components illustrated in Figure 3.1. An SCJVM is divided into two main parts: the core execution environment and the SCJVM services that may make use of the services of an underlying operating system or hardware abstraction layer.

The core execution environment manages the execution of Java bytecode, whether that be via interpretation, just-in-time compilation or ahead-of-time compilation. The core execution environment must also manage data that relates to the execution of bytecode instructions, such as the representation of classes and objects.

The SCJVM services represent the additional services that must be offered by an SCJVM in order to support the SCJ infrastructure. These services may be supplied as standalone services and so do not need to be handled by the compilation strategy. We consider the virtual machine services to be divided into three areas:

- the memory manager, which manages backing stores for memory areas and allocation within them;
• the scheduler, which manages threads and interrupts, and allows 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.

A model of the core execution environment is presented in Chapter 4. In this chapter, we present the
requirements for each area of the SCJVM services: the memory manager in Section 3.1, the scheduler
in Section 3.2, and the real-time clock in Section 3.3. The formal model of the SCJVM requirements is
presented in Section 3.4. A complete version of the model can be found in Appendix A.

The memory manager model has been subject to proof using Z/Eves. The theorems proved about the
memory manager can be found in Appendix D.3, with the Z/Eves proof scripts in Appendix D.4. Many
additional lemmas about objects in the Z/Eves mathematical toolkit have been proved in the course of
carrying out these proofs. As these can be of use outside our work, we have included them separately in
Appendix D.1 with their proofs in Appendix D.2.

Part of an earlier version of this model was presented at the 13th International Workshop on Java
Technologies for Real-time and Embedded Systems [11] with the full earlier version made available as a
technical report [10].

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
cannot be resized or destroyed, so that there is always a fixed base for the layout of memory. The root
backing store is 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 3.2. 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.

![Figure 3.2: An example memory layout](image)

The operations of the memory manager API are summarised in Table 3.1. 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 in Table 3.1 as well.

The root backing store 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.
Table 3.1: The operations of the SCJVM memory manager

<table>
<thead>
<tr>
<th>Operation</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Error Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRootBackingStore</td>
<td>(none)</td>
<td>backing store identifier</td>
<td>(none)</td>
</tr>
<tr>
<td>getTotalSize</td>
<td>backing store identifier</td>
<td>size in bytes</td>
<td>invalid identifier</td>
</tr>
<tr>
<td>getUsedSize</td>
<td>backing store identifier</td>
<td>size in bytes</td>
<td>invalid identifier</td>
</tr>
<tr>
<td>getFreeSize</td>
<td>backing store identifier</td>
<td>size in bytes</td>
<td>invalid identifier</td>
</tr>
<tr>
<td>findBackingStore</td>
<td>memory pointer</td>
<td>backing store identifier</td>
<td>no backing store found</td>
</tr>
<tr>
<td>allocateMemory</td>
<td>backing store identifier</td>
<td>size in bytes</td>
<td>invalid identifier</td>
</tr>
<tr>
<td></td>
<td></td>
<td>memory pointer</td>
<td>insufficient free memory</td>
</tr>
<tr>
<td>makeBackingStore</td>
<td>backing store identifier</td>
<td>backing store identifier</td>
<td>invalid identifier</td>
</tr>
<tr>
<td></td>
<td>size in bytes</td>
<td></td>
<td>insufficient free memory</td>
</tr>
<tr>
<td>clearBackingStore</td>
<td>backing store identifier</td>
<td>(none)</td>
<td>nested backing store in use</td>
</tr>
<tr>
<td></td>
<td>size in bytes</td>
<td></td>
<td>invalid identifier</td>
</tr>
<tr>
<td>resizeBackingStore</td>
<td>backing store identifier</td>
<td>backing store identifier</td>
<td>backing store in use</td>
</tr>
<tr>
<td></td>
<td>size in bytes</td>
<td></td>
<td>backing store is root</td>
</tr>
<tr>
<td>createStack</td>
<td>size in bytes</td>
<td>stack identifier</td>
<td>backing store not empty</td>
</tr>
<tr>
<td>destroyStack</td>
<td>stack identifier</td>
<td>(none)</td>
<td>insufficient free space</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>no space for memory overhead</td>
</tr>
</tbody>
</table>

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.

The backing store in which a particular memory address lies can also be queried. 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 a given backing store. This operation is provided in order for the core execution environment to implement the `new` bytecode instruction and is not directly available to the program or infrastructure. Though the memory manager allocates space for objects, there is no notion of objects in the memory manager since they only exist at the level of Java code, and so are dealt with by the core execution environment. Dealing solely with blocks of memory in the SCJVM services allows for objects to be represented in a way appropriate to the structure of the core execution environment. 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, whose memories are freed with the rest of the backing store. This would create a problem if the parent backing store were cleared while another thread is using a backing store within it as an allocation context. Such a situation should not occur as the backing stores of mission memory and immortal memory are the only ones 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. An attempt to clear a backing store with a nested backing store would fail and would have to be retried.
store in use is handled as an error case.

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. 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. Due to the fact that resizing a backing store can move it and that a backing store identifier may be a pointer to the backing store, the identifier may change and so the new identifier is output from this operation.

These operations on backing stores each take a backing store identifier as input since the memory manager does not handle allocation contexts. Management of allocation contexts is instead left to the core execution environment, which must pass the appropriate backing store identifier when using the memory manager services.

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 to the space allocated for the stack.

The memory manager must interact with the scheduler to obtain the current thread when it needs to operate on the current allocation context. The next section gives an overview of the scheduler.

### 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 are useful, for example, 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 3.2.

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 in the implementation of 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. The entry point is expressed as the class and identifier of the method that the thread is to run, along with any arguments for the method. 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
Table 3.2: The operations of the SCJVM scheduler

<table>
<thead>
<tr>
<th>Operation</th>
<th>Inputs</th>
<th>Outputs</th>
<th>Error Conditions</th>
</tr>
</thead>
<tbody>
<tr>
<td>getMaxSoftwarePriority</td>
<td>(none)</td>
<td>priority level</td>
<td>(none)</td>
</tr>
<tr>
<td>getMinSoftwarePriority</td>
<td>(none)</td>
<td>priority level</td>
<td>(none)</td>
</tr>
<tr>
<td>getNormSoftwarePriority</td>
<td>(none)</td>
<td>priority level</td>
<td>(none)</td>
</tr>
<tr>
<td>getMaxHardwarePriority</td>
<td>(none)</td>
<td>priority level</td>
<td>(none)</td>
</tr>
<tr>
<td>getMinHardwarePriority</td>
<td>(none)</td>
<td>priority level</td>
<td>(none)</td>
</tr>
<tr>
<td>getMainThread</td>
<td>(none)</td>
<td>thread identifier</td>
<td>(none)</td>
</tr>
<tr>
<td>makeThread</td>
<td>(none)</td>
<td>thread identifier</td>
<td>(none)</td>
</tr>
<tr>
<td>startThreads</td>
<td>(none)</td>
<td>thread identifier</td>
<td>(none)</td>
</tr>
<tr>
<td>getCurrentThread</td>
<td>(none)</td>
<td>thread identifier</td>
<td>(none)</td>
</tr>
<tr>
<td>destroyThread</td>
<td>(none)</td>
<td>thread identifier</td>
<td>(none)</td>
</tr>
<tr>
<td>suspendThread</td>
<td>(none)</td>
<td>(none)</td>
<td>invalid identifier</td>
</tr>
<tr>
<td>resumeThread</td>
<td>(none)</td>
<td>thread identifier</td>
<td>invalid identifier</td>
</tr>
<tr>
<td>setPriorityCeiling</td>
<td>pointer to object</td>
<td>priority level</td>
<td>invalid priority</td>
</tr>
<tr>
<td>takeLock</td>
<td>pointer to object</td>
<td>(none)</td>
<td>lock in use</td>
</tr>
<tr>
<td>releaseLock</td>
<td>pointer to object</td>
<td>(none)</td>
<td>lock not held</td>
</tr>
<tr>
<td>attachInterruptHandler</td>
<td>backing store identifier</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>detachInterruptHandler</td>
<td>interrupt identifier</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>getInterruptPriority</td>
<td>interrupt identifier</td>
<td>priority level</td>
<td>(none)</td>
</tr>
<tr>
<td>disableInterrupts</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>enableInterrupts</td>
<td>(none)</td>
<td>(none)</td>
<td>(none)</td>
</tr>
<tr>
<td>endInterrupt</td>
<td>(none)</td>
<td>(none)</td>
<td>not in interrupt</td>
</tr>
</tbody>
</table>

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. Threads are started using the `startThreads` operation, which takes a list of threads to start, together with the backing stores and stacks associated with them. They must be started by the infrastructure when its enclosing mission starts. The reason for the separation between thread creation and thread start is to facilitate the implementation of the SCJ control flow, which requires that threads all start together after mission initialisation has been completely finished. A backing store is provided when a thread is started to serve as the allocation context of the thread since the per-release memory of an event handler is only created as the handler thread is started. The backing store supplied is only used to set the backing store in the memory manager and core execution environment when the thread starts and is not stored by the scheduler.

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.

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.
A thread that has been created can then be destroyed with the `destroyThread` operation, which removes the thread 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, which is a mechanism to avoid priority inversion when threads synchronise via locking of objects. In priority ceiling emulation, each object has a priority ceiling, which is the priority of the highest priority thread that may lock the object. When locking an object, a thread’s active priority is temporarily raised to the priority ceiling of the object to ensure it is not blocked by higher priority threads waiting to access the same object. 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. From our perspective it does not matter what the default priority ceiling, only that it is a constant value for all threads for a given run of an SCJVM. 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 lower 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 must be scheduled along with threads. 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 is run with the priority of the interrupt it is associated to when that interrupt fires. An interrupt handler interrupts any lower-priority interrupt handlers and any running threads, and blocks lower-priority interrupts from occurring until it has finished. The priority associated with each interrupt can be obtained by the `getInterruptPriority` operation.

Interrupts can be disabled and re-enabled using the `disableInterrupts` and `enableInterrupts` operations. 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.

Finally, an interrupt can be ended using the `endInterrupt` operation. This operation should be used by the infrastructure to ensure normal thread scheduling resumes when an interrupt handler has finished execution. This operation cannot be used outside of an interrupt handler.

Though the scheduler manages most interrupts, the clock interrupt is managed by the real-time clock, which is the subject of the next section.

### 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 3.3.
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 (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 needs to 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.

This concludes our discussion of the API of SCJVM services. A formal account of each of the operations in Tables 3.1, 3.2, and 3.3 is the subject of the next section.

### 3.4 Formal Model

We now present the formal model of the SCJVM services in the Circus specification language. The model is structured using a single process for each group of SCJVM services described above, which are then combined in parallel to form a complete model of the SCJVM services. We describe the model of the memory manager in Section 3.4.1, the scheduler in Section 3.4.2, and the real-time clock in Section 3.4.3. Finally, the parts of the model are combined in Section 3.4.4.

#### 3.4.1 Memory Manager

As already said, the SCJVM memory manager is the component that manages the backing stores that underlie memory areas, and provides operations for creating, clearing, and resizing backing stores, and allocating within them. The memory manager also handles allocation and freeing of stack space.

In our formal model, we first declare the types and channels needed for the memory manager model, then build up the model in several layers, beginning with memory blocks that allow operations such as allocation, clearing, and resizing, then adding in the structure of backing stores that may contain other backing stores nested inside. Afterwards, the global memory manager covering all the backing stores is specified, and thread handling considerations are taken into account. Finally, the stack memory management is defined, with the stack area based on the memory blocks model. In this section, we present a Circus process that defines the memory manager; the paragraphs of this process include a Z specification that defines each of these layers separately.

Each backing store is identified by an implementation-defined backing store identifier, which may simply be a pointer to the backing store’s location in memory. In our model, we define a given set `BackingStoreID`...
that contains all possible backing store identifiers.

\[ \text{BackingStoreID} \]

The memory allocated by the SCJVM is in the form of raw contiguous blocks of memory. Memory addresses are modelled as natural numbers on the assumption that there are countably many memory addresses.

\[ \text{MemoryAddress} == \mathbb{N} \]

We use natural number ranges to define the concept of a contiguous memory block, which is central to the formalisation of the requirement that backing stores must not be fragmented.

\[ \text{ContiguousMemory} == \{ m : \mathbb{P} \text{MemoryAddress} \mid \exists a, b : \text{MemoryAddress} \bullet m = a \ldots b \} \]

In addition to managing backing stores, the memory manager must also manage stacks, which are also referred to by unique identifiers. The given set, \( \text{StackID} \), of valid stack identifiers is introduced below.

\[ \text{StackID} \]

We declare channels for each of the operations of the memory manager. Each channel name begins with \( \text{MM} \), to indicate that it corresponds to an operation of the memory manager API, followed by the name of the service. Operations that return a value have a separate channel to pass that value, the name of which is the name of the service channel with \( \text{Ret} \) appended to it. For example, the channels for \( \text{getRootBackingStore} \) are \( \text{MMgetRootBackingStore} \), which carries no values as the operation takes no inputs, and \( \text{MMgetRootBackingStoreRet} \), which communicates backing store identifiers output by the operation. We omit the channel declarations here for the sake of brevity. The definition of all channels and all other definitions we omit here can be found in Appendix A.

Each memory manager function reports a value signalling whether an error occurred and, if so, what error. These error values are of the type \( \text{MMReport} \), whose definition is sketched below, and are reported over the channel \( \text{MMreport} \).

\[ \text{MMReport} ::= \text{MMokay} \mid \text{MMoutOfMemory} \mid \text{MMnotEmpty} \mid \ldots \]

\text{channel MMreport : MMReport}

Lastly, we declare a channel through which the memory manager’s initialisation information can be supplied.

\text{channel MMinit : ContiguousMemory \times \mathbb{N} \times ContiguousMemory}

Having declared the channels, we begin the process declaration.

\text{process MemoryManager \equiv begin}

A certain amount of memory overhead can be included in allocated blocks of memory and backing stores to allow for implementation of a memory management algorithm. Memory allocation operations must ensure that there is enough memory available for both the requested amount of memory and the additional overhead. The overhead values must be constant, but may be zero for some memory management algorithms.

\[ \text{allocationOverhead, backingStoreOverhead} : \mathbb{N} \]

We cover each part of the memory manager model in a separate subsection: memory blocks in Section 3.4.1.1, backing stores in Section 3.4.1.2, the global memory manager in Section 3.4.1.3 and stacks in Section 3.4.1.4. Finally, we describe how the Z schemas are lifted to \text{Circus} operations in Section 3.4.1.5.
3.4.1.1 Memory Blocks

Memory is allocated within memory blocks that keep a record of the amount of used, free, and total memory. Memory blocks form the basis for both backing stores and stack allocation space. It is required that the free memory and the total memory are not fragmented. It is not required that the used memory be fragmentation free, however the union of the used and free memory must not be fragmented in order for allocation to work correctly. The used and free memory may not cover all of the memory in the memory block as there may be some overhead, as mentioned above. The used and free memory must be disjoint.

\[
\begin{align*}
\text{MemoryBlock} & : \text{ContiguousMemory} \\
\text{free}, \text{total} & : \mathbb{F} \text{MemoryAddress} \\
\text{used} & : \mathbb{F} \text{MemoryAddress} \\
\text{used} \cup \text{free} & \in \text{ContiguousMemory} \\
\text{used} \cup \text{free} & \subseteq \text{total} \\
\text{used} \cap \text{free} & = \emptyset
\end{align*}
\]

A memory block must be initialised with the total memory covered by the block, including space for any overhead, and initially has no used memory. The exact size of the overhead is nondeterministic as this is refined by backing stores and the stack area, which have different overheads.

\[
\begin{align*}
\text{MemoryBlockInit} & : \text{ContiguousMemory} \\
\text{MemoryBlock} & : \text{ContiguousAddress} \\
\text{addresses}? & : \text{ContiguousMemory} \\
\text{total}' & = \text{addresses}? \\
\text{free}' & \subseteq \text{addresses}? \\
\text{used}' & = \emptyset
\end{align*}
\]

 Allocation of memory within memory blocks is performed as described in the MBAllocate schema, which takes the requested allocation size as an input and outputs the allocated contiguous block of memory addresses, allocated!. There must be sufficient free memory for the requested allocation size. This operation removes allocated!, requiring that it be of the given size, from the free memory and adds it to the used memory, returning the allocated block.

\[
\begin{align*}
\text{MBAllocate} & : \Delta \text{MemoryBlock} \\
\text{size} & : \mathbb{N} \\
\text{allocated}! & : \text{ContiguousMemory} \\
\text{size}? & \leq \# \text{free} \\
\# \text{allocated}! & = \text{size}? \\
\text{allocated}! & \subseteq \text{free} \\
\text{used}' & = \text{used} \cup \text{allocated}! \\
\text{free}' & = \text{free} \setminus \text{allocated}! \\
\text{total}' & = \text{total}
\end{align*}
\]

Since free is required to be a ContiguousMemory, the removal of allocated! is guaranteed not to introduce fragmentation.

The operation of clear a memory block makes all its used memory free. This is done by setting the free memory to the union of the used and free memory, and setting the used memory to be empty.
The operation for resizing a memory block just sets the memory block to use a new set of contiguous addresses. This is described by `MBResize`, which takes a new contiguous memory set, `newAddresses`, as input and gives the old memory set, `oldAddresses!`, as output. The new set must have room for the implicit overhead (the difference between the size of `total` and the size of `free`). To avoid fragmentation, this is only possible when the memory block is empty. The set `newAddresses?` becomes the new `total` memory and a new `free` memory is chosen within it. The size of implicit overhead must be preserved and the used memory remains empty. The `oldAddresses!` output is simply the previous `total` memory.

We also have an operation to resize all or part of the used memory within a memory block. This takes as input a contiguous set of memory addresses, `oldAddresses!`, that is the area of memory that is resized by this operation and the size, `newSize?`, of the requested new area. The new set of contiguous memory addresses withing the memory block, `newAddresses!`, is output. The `oldAddresses?` set is required to be part of the used memory and is also required to be contiguous with the `free` memory so that a new area of memory can be allocated from their union. There must be enough space in the union of `oldAddresses?` and the `free` memory to allocate new memory of the requested size. The operation allocates memory, as described by `MBAllocate`, with the `free` memory taken to include `oldAddresses?` and the used memory having `oldAddresses?` removed. So the operation effectively frees `oldAddresses?` and allocates a new set, `newAddresses!`, of the requested size.

There are also operations to read the total, used and free sizes of a memory block. The schemas `MBGetTotalSize`, `MBGetUsedSize` and `MBGetFreeSize`, that define these operations are very simple, since the required information is directly available in the state. So, we have omitted them here.
The operations on the memory manager must be made into robust operations by adding error reporting. Errors are reported by returning a value to indicate the type of error, taken from the `MMReport` type.

The successful completion of an operation is indicated by returning `MMokay`. This is described in the schema `Success`, which is combined with the schemas just defined that describe the successful case of the operations and so does not need impose any requirements on the state.

\[
\begin{align*}
\text{Success} & \\
report! : \text{MMReport} & \\
report! = \text{MMokay}
\end{align*}
\]

The specification of the error cases follow a common pattern. They all do not change the state and output a value \(report!\) of type `MMReport` that specifies which error has occurred. Each error case has as its precondition a predicate specifying when the error is triggered. The inputs to the error case are the minimum needed to specify the precondition required. As an example of an error case, we present the schema `MBOutOfMemory` that takes a natural number \(size?\) as input and reports an out of memory error if \(size?\) is greater than the amount of free memory.

\[
\begin{align*}
\text{MBOutOfMemory} & \\
\exists MemoryBlock & \\
size? : \mathbb{N} & \\
report! : \text{MMReport} & \\
\neg size? \leq \# \text{free} & \\
report! = \text{MMoutOfMemory}
\end{align*}
\]

Other error cases are defined similarly, so we omit their specifications.

The operations on memory blocks can then be lifted to robust versions. The robust operations are named by prefixing \(R\) to the name of the lifted operation. Each is formed by taking the conjunction of the operation schema with `Success`, effectively adding an `MMokay` output to the operation; the error cases are placed in disjunction with it. In that way the error cases define what happens when the precondition of the error case is true and the precondition of the operation is false, making the robust operations total since all cases are covered. As an example of a robust operation, we present the robust memory allocation operation, which has `MBOutOfMemory`, defined above, as its only error case.

\[
\begin{align*}
\text{RMBAllocate} & = (\text{MBAlocate} \land \text{Success}) \lor \text{MBOutOfMemory}
\end{align*}
\]

Having modelled memory blocks and the operations upon them, we now proceed to specialise memory blocks to form a model of backing stores. Memory blocks are also used later as the basis for the stack space specification.

### 3.4.1.2 Backing Stores

The memory manager deals with memory in the form of backing stores, represented by the schema `BackingStore` shown below. `BackingStore` contains two `MemoryBlock` s: an `objectSpace` in which objects are allocated, and a `bsSpace` in which nested backing stores can be allocated. The backing stores nested directly within a backing store, which we refer to as its `children`, are represented by a finite set of backing store identifiers. Backing stores nested deeper are not included in the set of `children`. The full structure of backing store nesting is specified later in the global memory manager. Each backing store in this model also stores its own identifier, `self`. A backing store is required to not be a child of itself. The union of `used` and `free` space in `objectSpace` and `bsSpace` is required to be contiguous, so any overhead must be at the beginning or end of the backing store’s space. The `objectSpace` and `bsSpace` must not overlap. The overhead is also specified to be equal in size to `backingStoreOverhead`. 
Backing stores are initialised with a contiguous address range \( addresses \), plus the identifier \( self \) of the backing store and a natural number \( objectSpaceSize \), which indicates the required size of the object space. The object space and bsSpace of the backing store are initialised as described by MemoryBlockInit, with address ranges that partition \( addresses \). The amount of free space in the object space must be \( objectSpaceSize \) and the non-free size in bsSpace and object space must be equal to the backingStoreOverhead. The self identifier is initialised to the \( self \) input and there are initially no children.

The operation of allocating a new child backing store is based on the MBAlocate schema. Additional updates to the BackingStore state must also be made to set the value of children. The identifier of the new child backing store must also be returned and it must be ensured that the total size of the child backing store is large enough to include the backingStoreOverhead. These additional requirements are specified in a separate schema BSAllocateChild0, which operates over the parent BackingStore. It takes an input \( size \), which represents the size of the space to be allocated, and gives an output \( childID \),
which is the identifier of the allocated child BackingStore. The size? must be sufficient to contain the backingStoreOverhead. The childID! must not be one of the children of the parent backing store, nor may it be the identifier self of the parent backing store. The childID! is added to the children set and self is unchanged. The other components of the parent BackingStore are not constrained by BSAllocateChild as they are updated by promoted MemoryBlock operations.

```
BSAllocateChild0
 ΔBackingStore
 size? : N
 childID! : BackingStoreID

 size? ≥ backingStoreOverhead
 childID! ∉ children ∧ childID! ≠ self
 children' = children ∪ {childID!}
 self' = self
```

The BSAllocateChild schema is combined with an MBAlocate operation promoted to act over the bsSpace of the BackingStore using the PromoteMsToBS schema. The input and output of MBAlocate are renamed to remove the decoration applied to make MBAlocate act over the bsSpace. The objectSpace is unaffected by this operation.

```
BSAllocateChild == ∃∃MemoryBlock1; ΔMemoryBlock2 •
 BSAllocateChild0 ∧ MBAlocate2[size?/size?2, allocated!/allocated!2] ∧ PromoteMsToBS
```

The other BackingStore operations are specified in a similar fashion, promoting MemoryBlock operations to act upon the objectSpace and bsSpace of a BackingStore, and specifying additional conditions in a separate schema.

Clearing a backing store removes all of its children and does not affect its self identifier, as specified in BSClear0 below.

```
BSClear0
 ΔBackingStore

 children' = ∅
 self' = self
```

The operation of clearing a backing store is then specified by BSClear, which is a combination of BSClear0 and MBClear operations promoted to act over both objectSpace and bsSpace.

```
BSClear == ∃ΔMemoryBlock1; ΔMemoryBlock2 •
 BSClear0 ∧ MBClear1 ∧ MBClear2 ∧ PromoteMsToBS
```

Allocating object memory within a backing store is performed with the additional inputs and output defined in BSAllocate0. There is an input size?, which is the required size of the object memory to be allocated, and the allocated memory is provided via an output allocated!. Since space for the allocationOverhead must be allocated when object memory is allocated, an actualSize value is computed by adding the allocationOverhead to size?. The children and self components of the BackingStore are unaffected by this operation.

```
BSAllocate0
 ΔBackingStore
 size? : N
 allocated! : ContiguousMemory
 actualSize : N

 actualSize = size? + allocationOverhead
 children' = children
 self' = self
```
The allocation operation is then specified by \textit{BSAllocate} below, which promotes \textit{MBAllocate} to act over the \texttt{objectSpace} of the \textit{BackingStore}. The \textit{actualSize} is used as the \textit{size?} input to \textit{MBAllocate} and hidden so that the \textit{size?} input to \textit{BSAllocate} is the only input of \textit{BSAllocate}.

\begin{align*}
\textit{BSAllocate} & \equiv \exists \Delta \textit{MemoryBlock}_1; \Xi \textit{MemoryBlock}_2; \textit{actualSize} : \mathbb{N} \bullet \\
& \quad \textit{BSAllocate0} \land \textit{MBAllocate} [\textit{actualSize}/\textit{size?}_1, \textit{allocated}/\textit{allocated}_1] \\
& \quad \land \textit{PromoteMBsToBS}
\end{align*}

The operation of resizing a backing store adjusts the sizes of its \texttt{objectSpace} and \texttt{bsSpace}, and so it is specified by the \textit{BSResize} schema, rather than being promoted from a \textit{MemoryBlock} operation. There is one input to \textit{BSResize}, which is \textit{newSize?}, the desired new size of the \texttt{objectSpace}. The \textit{newSize?} must be large enough to contain any existing allocations in the \texttt{objectSpace} (i.e. the \textit{used} part of \texttt{objectSpace}) and those allocations are unaffected by this operation. The \texttt{objectSpace} is resized so that the combination of its \textit{used} and \textit{free} space is as large as \textit{newSize}. The additional \textit{free} space in \texttt{objectSpace} is taken from the \texttt{free} space of the \texttt{bsSpace} cannot contain any \textit{used} space, since that may get in way of the resizing, and the \texttt{used} space remains empty after the operation. As a consequence of the \textit{used} part of \texttt{bsSpace} being empty, \textit{children} must also be empty, since there can be no child backing stores. The \textit{self} identifier is unaffected.

\begin{align*}
\textit{BSResize} & \equiv \Delta \textit{BackingStore} \\
& \quad \textit{newSize?} : \mathbb{N} \\
& \quad \textit{newSize?} \geq \# \texttt{objectSpace}.\textit{used} \\
& \quad \texttt{objectSpace}'\textit{.used} = \texttt{objectSpace}.\textit{used} \\
& \quad \#(\texttt{objectSpace}'\textit{.used} \cup \texttt{objectSpace}'\textit{.free}) = \textit{newSize?} \\
& \quad \texttt{objectSpace}'\textit{.free} \cup \texttt{bsSpace}'\textit{.free} = \texttt{objectSpace}.\textit{free} \cup \texttt{bsSpace}.\textit{free} \\
& \quad \texttt{bsSpace}.\textit{used} = \emptyset = \texttt{bsSpace}'\textit{.used} \\
& \quad \texttt{children} = \emptyset = \texttt{children}' \\
& \quad \textit{self}' = \textit{self}
\end{align*}

The other operations on backing stores can be defined by promoting the memory block operations to operate on the \texttt{objectSpace} of a backing store, keeping the set of \textit{children} and the \textit{self} identifier the same.

\begin{align*}
\textit{BSGetTotalSize} & \equiv \exists \Delta \textit{MemoryBlock}_1; \Xi \textit{MemoryBlock}_2; \textit{MBGetTotalSize}[\texttt{size}/\textit{size}_1] \land \textit{PromoteMBsToBS} \\
\textit{BSGetUsedSize} & \equiv \exists \Delta \textit{MemoryBlock}_1; \Xi \textit{MemoryBlock}_2; \textit{MBGetUsedSize}[\texttt{size}/\textit{size}_1] \land \textit{PromoteMBsToBS} \\
\textit{BSGetFreeSize} & \equiv \exists \Delta \textit{MemoryBlock}_1; \Xi \textit{MemoryBlock}_2; \textit{MBGetFreeSize}[\texttt{size}/\textit{size}_1] \land \textit{PromoteMBsToBS} \\
\textit{BSGetRemainingBS} & \equiv \exists \Xi \textit{MemoryBlock}_1; \Delta \textit{MemoryBlock}_2; \textit{MBGetRemainingBS}[\texttt{size}/\textit{size}_2] \land \textit{PromoteMBsToBS}
\end{align*}

These operations must then be made into robust operations that report error values if their preconditions are not met. Some of the error reporting schemas for memory blocks can be reused, but there are new preconditions in the backing store operations based on the memory block operations that must be accounted for. These require additional schemas, but we have omitted their definitions here as they are similar in form to the memory block error cases.

Using these schemas, the operations on backing stores can be made into robust operations. This lifting to robust operations is similar to that for the memory block operations. As an example, we present the robust backing store initialisation operation. The initialisation schema presented earlier is combined with \textit{Success} to output \textit{MMokay} in the event of a successful initialisation. Its only error case is that in which the provided set of addresses is too small to contain the backing store overhead. As the schema for
this error case is used for other operations, it has an initial state that is not present during initialisation and so it must be hidden using existential quantification.

\[
R_{\text{BackingStoreInit}} \equiv (\text{BackingStoreInit} \land \text{Success}) \lor (\exists \text{BackingStore} \bullet \text{BSSizeTooSmall})
\]

We omit the other robust operations as they are similar to the robust memory block operations.

This concludes our model of backing stores as individual structures. Next we specify the global memory manager, which contains all backing stores and whose invariant records the relations between them.

### 3.4.1.3 Global Memory Manager

The memory manager must hold information on all backing stores and the identifier of the one that is the root backing store. All backing stores must be nested within the root backing store.

Some of the memory manager operations operate on a backing store that is designated as the current allocation context, which may be different for each thread. The memory manager stores the current allocation context for each thread so that, given the current thread identifier by the scheduler, the correct current allocation context can be determined.

The information about all the backing stores is held in the the global memory manager state, which we split into several parts to make specification of invariants easier. The first part is `GlobalStoresManager`, which contains a map, `stores`, from backing store identifiers to backing stores. This map is partial, since not all backing store identifiers may be used, and finite, since there will only ever be a finite number of backing stores in use. This is because none of the operations on the memory manager creates an infinite number of backing stores. The `GlobalStoresManager` also contains the identifier of the root backing store, `rootBackingStore`, since that is used in specifying several of the invariants of the memory manager.

\[
\begin{align*}
\text{GlobalStoresManager} & \\
\text{stores} : \text{BackingStoreID} \rightarrow \text{BackingStore} & \\
\text{rootBackingStore} : \text{BackingStoreID} & \\
\forall \text{bsid} : \text{dom stores} & \\
(\text{stores bsid}).\text{self} = \text{bsid} & \\
(\text{stores bsid}).\text{children} \subseteq \text{dom stores} & \\
(\lambda \text{childID} : (\text{stores bsid}).\text{children} & \\
(\text{stores childID}).\text{objectSpace}.\text{total} \cup (\text{stores childID}).\text{bsSpace}.\text{total} & \\
\text{partition} (\text{stores bsid}).\text{bsSpace}.\text{used} & \\
\end{align*}
\]

The first invariant of the `GlobalStoresManager` state requires that the `rootBackingStore` identifier be in the domain of `stores`. The remaining invariants of the `GlobalStoresManager` are specified to hold for any backing store identifier `bsid` in the domain of `stores`. The `self` identifier of the backing store `bsid` is mapped to under `stores` must be the same as `bsid` itself. This ensures that a backing store identifier cannot be mapped to a completely different backing store and imposes an injectivity condition on `stores` whereby two backing store identifiers cannot be mapped to the same backing store. The `children` of the backing store identified by `bsid` must also be in the domain of `stores`.

Then the `GlobalMemoryManager` schema represents the full state of the backing stores in the memory manager. It contains `GlobalStoresManager` and also contains a relation, `childRelation`, that represents the structure of backing store nesting, relating backing store identifiers to the identifiers of their children.

\[
\begin{align*}
\text{GlobalMemoryManager} & \\
\text{GlobalStoresManager} & \\
\text{childRelation} : \text{BackingStoreID} \leftrightarrow \text{BackingStoreID} & \\
\forall \text{bsid} : \text{dom stores} & \bullet \text{childRelation} \{ \{ \text{bsid} \} \} = (\text{stores bsid}).\text{children} & \\
\text{dom stores} = (\text{childRelation}^*) \{ \{ \text{rootBackingStore} \} \} & \\
\forall \text{bsid} : \text{dom stores} & \bullet \text{bsid} \notin \text{childRelation}^* \{ \{ \text{bsid} \} \} & \\
\end{align*}
\]

34
The first invariant of GlobalMemoryManager defines childRelation by stating that the image of a given backing store identifier, bsid, under childRelation is the children set of the corresponding backing store. The remaining two invariants restrict the structure of childRelation to that of a tree. The first requires every identifier in the domain of stores to be reachable from the rootBackingStore by stating that the image of rootBackingStore under the reflexive transitive closure of childRelation must be equal to the domain of stores. The second ensures there will be no loops by stating that each backing store cannot be related to itself under the transitive closure of childRelation.

Initially there must be one backing store provided, which is the root backing store. The memory manager must be initialised with the set of memory addresses to be used for the root backing store. The root backing store is initialised with these addresses as described in RBackingStoreInit, with the input self? set to the rootBackingStore identifier, which may be any available backing store identifier. The childRelation is initially empty because there is only one backing store that initially has no children.

The operations on the global memory manager are specified using the Z idiom of promotion in which operations on a local state are lifted to operations over a global state that stores multiple different local states. Promotion works by using the local operation to describe the update of a local state and capturing the local state components using the Z schema binding operator \( \theta \). The captured local state is then used to update the global state.

In our case, operations on backing stores are promoted to operations over the global memory manager, updating the backing stores in the stores map. This is handled by the PromoteBS schema, defined below, which takes a backing store identifier as input. The state components for both the local state and the global state are brought into scope so that the promotion can act on both of them. The update of the local state is performed by the operation schema, which is combined with the promotion schema. The backing store identifier is required to be in the domain of stores and the initial state of the corresponding backing store is captured from the global state. The final state of the local operation is then captured and used to update stores.

As an example of a promotion, we present the operation of allocating memory. The local state used in the promotion is hidden using existential quantification so that the operation is an operation on the global state. The operation over the local state is combined with the promotion schema declared above to form the operation over the global state. We also adjust the operation to return the address of the start of the memory block. This is achieved by conjoining another schema to it that contains an address!
output, which is set to be the minimum of the allocated! addresses output by \texttt{RBSAllocate}.

\[
\text{GlobalAllocateMemory} == \exists \Delta \text{BackingStore}; \text{allocated! : ContiguousMemory} \diamond \text{RBSAllocate} \land \text{PromoteBS} \land \\
\left[\text{allocated! : ContiguousMemory}; \text{address! : MemoryAddress} \mid \text{address!} = \text{min allocated!}\right]
\]

The other promoted operations are the operations for getting the total, used and free size of a given backing store: \texttt{GlobalGetTotalSize}, \texttt{GlobalGetUsedSize} and \texttt{GlobalGetFreeSize}. These are defined similarly, so we omit their definitions here.

Some of the global memory manager operations differ from the standard form for promotion as the need to promote more than one schema at once or update the global state in an unusual way. Those operations are explained here.

The operation of making a new backing store inside a given backing store is performed by allocating space inside the parent backing store, as specified by \texttt{RBSAllocateChild}, and initialising the new child backing store, as specified by \texttt{RBackingStoreInit}. The schema that describes this operation, \texttt{GlobalMakeBS}, is defined below by promoting both of these operations.

\[
\text{GlobalMakeBS}
\]
\[
\Delta \text{GlobalMemoryManager} \\
\text{size? : } \mathbb{N} \\
\text{objectSpaceSize? : } \mathbb{N} \\
\text{parentID? : } \text{BackingStoreID} \\
\text{childID! : } \text{BackingStoreID} \\
\text{parentID? } \in \text{dom stores} \\
\exists \text{actualSize : } \mathbb{N} \mid \text{actualSize} = \text{size?} + \text{backingStoreOverhead} \\
\exists \text{allocated! : ContiguousMemory} \\
\exists \text{Parent, child} : \text{BackingStore} \\
\left(\exists \Delta \text{BackingStore}; \text{report! : MMReport} \mid \\
\text{RBSAllocateChild}[\text{actualSize/size?}] \diamond \right.
\left.\theta \text{BackingStore} = \text{stores parentID?} \land \\
\text{Parent} = \theta \text{BackingStore}' \land \\
\text{childID}! \notin \text{dom stores} \land \\
\text{report!} = \text{MMokay} \right) \land
\left(\exists \text{BackingStore}'; \text{report! : MMReport} \mid \\
\text{RBackingStoreInit}[\text{allocated!}/\text{addresses?}, \text{childID!}/\text{self'?}] \diamond \right.
\left.\text{self'} = \text{childID!} \land \\
\text{child} = \theta \text{BackingStore}' \land \\
\text{report!} = \text{MMokay} \right) \land
\text{stores'} = \text{stores} \oplus \{\text{parentID? } \mapsto \text{Parent, childID! } \mapsto \text{child}\}
\]

\[
\text{rootBackingStore'} = \text{rootBackingStore}
\]

The \texttt{GlobalMakeBS} schema takes as input the required size of the new backing store, \text{size?} and the identifier of its parent, \text{parentID?}. The identifier of the new child backing store, \text{childID!}, is given as output. There is a precondition that the parent identifier must be in the domain of \text{stores}, since it must be valid backing store identifier. We also define a value \text{actualSize} to be \text{size?} plus \text{backingStoreOverhead}, so that the \text{size?} is the actual amount of usable space in the backing store, without any overhead. A local variable \text{allocated!} is brought into scope using existential quantification to hold the addresses output by \texttt{RBSAllocateChild}. Two local variables \text{Parent} and \text{child} are also introduced to store the final local states of the promoted operations. The promotions of each of the local state operations are then specified. The operation \texttt{RBSAllocateChild} is promoted to act on the local state of the parent, with \text{actualSize} replacing its size input, and its final state is stored in \text{Parent}. The \text{childID!} and \text{allocated!} variable is identified with the outputs from \texttt{RBSAllocateChild} of the same name. It is required that \text{childID!} not be already in the domain of \text{stores}. The error report from the promoted operations must be a report of success for the global operation to work; the cases where it is not are handled as separate error cases. The operation \texttt{RBackingStoreInit} is promoted to initialise a local state for the newly created backing store.
The outputs \( \text{allocated!} \) and \( \text{childID!} \) from \( \text{RBSAllocateChild} \) are used to replace the \( \text{addresses?} \) and \( \text{self?} \) inputs to \( \text{RBackingStoreInit} \). The new backing store is stored in \( \text{child} \). The \( \text{stores} \) map is updated to contain \( \text{Parent} \) and \( \text{child} \).

The operation of clearing a backing store is described by the schema \( \text{GlobalClearBS} \), which takes a backing store identifier, \( \text{toClear?} \), as input and promotes the \( \text{RBSClear} \) operation to act over the corresponding backing store. The \( \text{toClear?} \) identifier is required to be a valid backing store identifier. The error report is required to be a report of success, as for the promotions in the \( \text{GlobalMakeBS} \) schema above. This promotion differs from that described by \( \text{PromoteBS} \) in that it removes backing stores nested within the cleared backing store from the \( \text{stores} \) map. The identifiers of the nested backing stores are those reachable via the transitive closure of the child relation, so we define a set \( \text{reachable} \) as the image of the cleared backing stores identifier under the transitive closure of the child relation. This set of backing store identifiers is removed from the domain of \( \text{stores} \) using the domain antirestriction operator, \( \prec \). The child relation is also updated, with the nested backing stores being removed from its range using the range antirestriction operator, \( \vdash \).

\[
\text{GlobalClearBS}
\]
\[
\Delta \text{GlobalMemoryManager}
\]
\[
toClear? \in \text{dom stores}
\]
\[
\exists \Delta \text{BackingStore}; \text{report!} : \text{MMReport} \mid \text{RBSClear} \circ
\]
\[
\theta \text{BackingStore} = \text{stores toClear?} \land
\]
\[
\text{report!} = \text{MMokay} \land
\]
\[
\exists \text{reachable} : \forall \text{BackingStoreID} \mid
\]
\[
\text{reachable} = \text{childRelation} \circ \text{toClear?} \land
\]
\[
\text{stores'} = (\text{reachable} \circ \text{stores}) \cup \{\text{toClear?} \rightarrow \theta \text{BackingStore'}\} \land
\]
\[
\text{childRelation'} = \text{childRelation} \bowtie \text{reachable}
\]
\[
\text{rootBackingStore'} = \text{rootBackingStore}
\]

It must also be possible to determine which backing store a given memory address belongs to, in order to implement the \( \text{getMemoryArea} \) method of \( \text{MemoryArea} \). This is handled by the \( \text{GlobalFindAddress} \) schema, which does not affect the local state of any backing stores or the state of the memory manager. The address to search for is taken as an input, \( \text{address?} \), to the operation and must be within the total memory of the \( \text{rootBackingStore} \) for this operation to work. The backing store containing the address is obtained by first finding all backing stores that contain the address, the identifiers of which are defined as a set \( \text{containingStores} \). The backing store which contains the address is then taken to be the backing store which is contained by all of those backing stores. Since that backing store is unique and well-defined, we obtain it using the Z unique specification operator, \( \mu \). This operator picks the identifier \( \text{bsid} \) in \( \text{containingStore} \) that satisfies the property that it is a child of all backing stores in \( \text{containingStores} \), which is returned as the \( \text{backingStore!} \) output.

\[
\text{GlobalFindAddress}
\]
\[
\exists \text{GlobalMemoryManager}
\]
\[
\text{address?} : \text{MemoryAddress}
\]
\[
\text{backingStore!} : \text{BackingStoreID}
\]
\[
\text{address?} \in (\text{stores rootBackingStore}).\text{objectSpace}.\text{total}
\]
\[
\cup (\text{stores rootBackingStore}).\text{objectSpace}.\text{total}
\]
\[
\text{backingStore!} = (\mu \text{bsid} : \text{dom stores} \mid \text{address?} \in (\text{stores bsid}).\text{objectSpace}.\text{total})
\]

There must also be an operation to obtain the identifier of the root backing store from the global memory manager. This is provided by the schema \( \text{GlobalGetRootBackingStore} \), which we omit here as it just provides the value of the state component \( \text{rootBackingStore} \).

These operations on the global memory manager are made into robust operations that report errors. The robust versions of the backing store operations have been used in specifying some of the robust global
memory manager operations. Some new error reporting schemas are also required to handle errors that
can occur at the level of the global memory manager and errors in the reports from promoted schemas.
As the structure of making the operations robust is similar to that used for memory blocks and backing
stores, we omit it here.

This concludes the Z definition of backing store operations; they will be lifted to Circus actions after the
definition of the stack management operations, presented next.

### 3.4.1.4 Stack Memory Manager

As previously discussed, in addition to providing facilities for backing stores and memory allocation,
the SCJVM must allow for allocating thread stacks. The stacks should be allocated in an area separate
from the root backing store, set aside for the allocation of stacks when the SCJVM starts. The SCJVM
memory management need only provide the memory for the stacks; management of the stack contents
must be handled by the core execution environment.

The stack area is a memory block that holds additional information about allocated stacks so that they
can be deallocated when the thread is removed. So, thread stacks may have their own memory overhead
associated with them.

- \( stackOverhead : \mathbb{N} \)

The stack memory manager controls a memory block using a function mapping stack identifiers to the
memory of the associated stack. The memory allocated for stacks must partition the used stack memory.

| StackMemoryManager
| MemoryBlock
| stacks : StackID → ContiguousMemory

The stack manager is initialised with a given area of memory for allocating stacks. The initialisation
schema is based on the initialisation of memory blocks. There are initially no stacks allocated, so \( stacks \)
is empty.

| StackMemoryManagerInit
| StackMemoryManager′
| MemoryBlockInit
| stacks′ = \( \emptyset \)

The operation to create a new stack of a given size is defined by \( StackCreate \) and is based on \( RMBAllocate \).
The stack overhead must be taken into account in this operation as we are allocating stacks, not space
for objects or backing stores. The new stack’s identifier, \( newStack! \) must be one not already in use. The
new identifier is stored in the map \( stacks \), mapping it to the allocated memory, and is also output from
the operation.

| StackCreate
| ΔStackMemoryManager
| size\? : \mathbb{N}
| newStack! : StackID

- \( newStack! \notin \text{dom} \, \text{stacks} \)
- \( \exists \text{actualSize} : \mathbb{N} \mid \text{actualSize} = \text{size}\? + \text{stackOverhead} \)
- \( \exists \text{report!} : \text{MMReport} \mid \text{allocated!} : \text{ContiguousMemory} \)
  - \( \text{RMBAllocate}[\text{actualSize} / \text{size}\?] \land \text{report!} = \text{MMokay} \land \text{stacks′} = \text{stacks} \oplus \{ \text{newStack!} ↦ \text{allocated!} \} \)
There is also an operation to delete a stack, freeing the memory used for it. This is defined by the schema \textit{StackDelete}, which takes the identifier of the stack to delete as input. The identifier is required to be an existing valid identifier, i.e. in the domain of \textit{stacks}. The space allocated for the stack is removed from the used memory and added to the free memory. The free memory with the stack allocation added to it must be contiguous. The identifier of the deleted stack is removed from the domain of \textit{stacks}.

\[
\begin{align*}
\text{StackDelete} \\
\Delta \text{StackMemoryManager} \\
\text{stack}? : \text{StackID} \\
\text{stack}? \in \text{dom stacks} \\
\text{used'} = \text{used} \setminus \text{stacks stack}? \\
\text{free'} = \text{free} \cup \text{stacks stack}? \in \text{ContiguousMemory} \\
\text{stacks'} = \{\text{stack}?\} \cup \text{stacks} \\
\text{total'} = \text{total} 
\end{align*}
\]

The stack memory manager operations are made into robust operations.

So far, we have defined the memory manager operations as a Z data model. The operations must now be made available via the \textit{Circus} channels for the memory manager process.

### 3.4.1.5 Memory Manager Operations

We make the operations defined by the schemas above available as services accessible via \textit{Circus} channels. Each of the services described in Section 3.1 are provided. The correspondence between the services described in section Section 3.1, the \textit{Circus} actions described here, and the Z schemas defined earlier is shown in Table 3.4. There are also two additional operations, \textit{addThread} and \textit{removeThread} that are only used internally for communication with the scheduler and are not part of the public interface of the memory manager.

<table>
<thead>
<tr>
<th>Service</th>
<th>Circus action</th>
<th>Z schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>getRootBackingStore</td>
<td>GetRootBackingStore</td>
<td>RGlobalGetRootBackingStore</td>
</tr>
<tr>
<td>getTotalSize</td>
<td>GetTotalSize</td>
<td>RGlobalGetTotalSize</td>
</tr>
<tr>
<td>getUsedSize</td>
<td>GetUsedSize</td>
<td>RGlobalGetUsedSize</td>
</tr>
<tr>
<td>getFreeSize</td>
<td>GetFreeSize</td>
<td>RGlobalGetFreeSize</td>
</tr>
<tr>
<td>getRemainingBackingStore</td>
<td>GetRemainingBS</td>
<td>RGlobalGetRemainingBS</td>
</tr>
<tr>
<td>findBackingStore</td>
<td>FindBackingStore</td>
<td>RGlobalFindAddress</td>
</tr>
<tr>
<td>allocateMemory</td>
<td>AllocateMemory</td>
<td>RGlobalAllocateMemory</td>
</tr>
<tr>
<td>makeBackingStore</td>
<td>MakeBackingStore</td>
<td>RGlobalMakeBS</td>
</tr>
<tr>
<td>clearBackingStore</td>
<td>ClearBackingStore</td>
<td>RGlobalClearBS</td>
</tr>
<tr>
<td>resizeBackingStore</td>
<td>ResizeBackingStore</td>
<td>RGlobalResizeBS</td>
</tr>
<tr>
<td>createStack</td>
<td>CreateStack</td>
<td>RStackCreate</td>
</tr>
<tr>
<td>destroyStack</td>
<td>DestroyStack</td>
<td>RStackDestroy</td>
</tr>
</tbody>
</table>

Table 3.4: The relationship between the memory manager services and the \textit{Circus} actions and Z schemas defining them

The state of the memory manager process is made up of both the global memory manager and the stack memory manager.

\[
\text{state } \text{GlobalMemoryManager } \land \text{StackMemoryManager}
\]

The memory manager is initialised by taking the root backing store and stack space as inputs and using the initialisation schemas for both the global memory manager and the stack memory manager. The
error value from the global memory manager initialisation is reported.

\[
\text{Init} \triangleq \text{var report : MMReport} \quad \bullet \\
\text{MMinit} \downarrow \text{addresses} \uparrow \text{objectSpaceSize} \uparrow \text{stackSpace} \rightarrow \quad \\
\left( \text{RGlobalMemoryManagerInit} \land \text{StackMemoryManagerInit} \right); \\
\text{MMreport} \downarrow \text{report} \rightarrow \text{Skip}
\]

The lifting of operations to Circus actions follows a common pattern, which can be seen here in the definition of the \text{GetRootBackingStore} action. The request to perform the operation, along with any inputs, is received on the operation’s channel. The operation is then performed as specified by a corresponding schema and any outputs from the operation are communicated on the return channel for the operation. The error report is communicated on the error reporting channel before the operation ends.

\[
\text{GetRootBackingStore} \triangleq \text{var report : MMReport; rbs : BackingStoreID} \quad \bullet \\
\text{MMgetRootBackingStore} \rightarrow \left( \text{RGlobalGetRootBackingStore} \right); \\
\text{MMgetRootBackingStoreRet} \downarrow \text{rbs} \rightarrow \text{MMreport} \downarrow \text{report} \rightarrow \text{Skip}
\]

The memory manager continuously presents all its operations in a loop. Any operation can be chosen once the previous operation has completed.

\[
\text{Loop} \triangleq \text{GetRootBackingStore} \diamond \text{GetCurrentAllocationContext} \diamond \cdots \diamond \text{Loop}
\]

The main action of the memory manager process first requires initialisation and then enters the operation loop declared above.

\[
\bullet \text{Init}; \text{Loop} \\
\text{end}
\]

This concludes the specification of the memory manager. We have built the memory manager in several layers, first defining the concept of a memory block, in which allocations can occur and which is used as the basis for specifying backing stores and the stack space. We then specified backing stores, which are memory blocks that keep a record of other backing stores nested within them. The backing store operations have then been promoted to act over a global memory manager with a view of all backing stores. For the operation of allocating memory, which must work with the current allocation context, operations were added to track the current allocation context of each running thread and memory allocation was promoted to act upon it. Allocation and deallocation of space for stacks has also been specified, with the stack space treated as a memory block to allow memory for stacks to be allocated within it. Finally, we lifted the operations to Circus actions, making them available over channels, via which the inputs to the operation (if any) are provided. Outputs from operations with output are provided via a separate return channel and all operations also report whether an error occurred via a separate error reporting channel.

Having specified the SCJVM services related to memory management in this section, we cover the next group of services, relating to scheduling, in the next section.

### 3.4.2 Scheduler

The SCJVM scheduler must manage separate threads of execution, which involves tracking information about threads, selecting which thread to run, handling locks, and blocking threads. The scheduler must also manage interrupts as they interfere with thread scheduling.

Threads are identified by unique implementation-defined thread identifiers of the \text{ThreadID} type.

\[
\text{[ThreadID]}
\]

There are two particular \text{ThreadID} values that identify special threads that exist from the start of the program. These are \text{idle}, which identifies the idle thread that does nothing and runs when no other
thread is available to run, and main, which identifies the thread used during SCJVM startup. These two identifiers must be distinct.

\[
\text{idle, main : ThreadID}
\]

\[
idle \neq \text{main}
\]

Threads are scheduled according to their priorities. Priorities are divided into hardware priorities, which are used for interrupt handlers, and software priorities, which are used for threads. There must be support for at least 28 priorities, with hardware priorities being higher than software priorities. One software priority must be designated as the normal priority.

\[
\begin{align*}
\text{minHwPriority, maxHwPriority} & : \mathbb{N} \\
\text{minSwPriority, maxSwPriority} & : \mathbb{N} \\
\text{normSwPriority} & \leq \text{minSwPriority} < \text{maxSwPriority} < \text{minHwPriority} < \text{maxHwPriority} \\
\text{(maxHwPriority - minHwPriority) + (maxSwPriority - minSwPriority)} & \geq 28
\end{align*}
\]

We define separate types for thread priorities and interrupt priorities so it can be checked in the model that a thread is not started with an interrupt priority.

\[
\begin{align*}
\text{ThreadPriority} &= \text{minSwPriority}..\text{maxSwPriority} \\
\text{InterruptPriority} &= \text{minHwPriority}..\text{maxHwPriority}
\end{align*}
\]

For the situations where either a thread or interrupt priority could be used, we us a type formed by joining the two sets.

\[
\text{Priority} == \text{ThreadPriority} \cup \text{InterruptPriority}
\]

The threads represent threads of execution of Java bytecode programs in the core execution environment. The scheduler must be able to inform the core execution environment when a thread switch occurs so that it can swap the stack and program counter. This is done using the \text{CEEswitchThread} channel.

\[
\text{channel CEEswitchThread : ThreadID} \times \text{ThreadID}
\]

The scheduler must also be able to provide the information required when a thread starts, which is the thread’s initial backing store, class, method and arguments. To declare the appropriate channel, the types of class identifiers, method identifiers and virtual machine words are required. We declare the types of class and method identifiers as the given types \text{ClassID} and \text{MethodID}

\[
\begin{align*}
\text{[ClassID, MethodID]}
\end{align*}
\]

The type, Word, of virtual machine words is defined to be the same as the type \text{MemoryAddress} since virtual machine words may hold memory addresses.

\[
\text{Word} == \text{MemoryAddress}
\]

The information is communicated to the core execution environment via the \text{CEEstartThread} channel.

\[
\text{channel CEEstartThread : ThreadID} \times \text{BackingStoreID} \times \text{StackID} \times \text{ClassID} \times \text{MethodID} \times \text{seq Word}
\]

The scheduler is also able to inform the core execution environment when a thread is no longer valid via the \text{CEEremoveThread} channel.

\[
\text{channel CEEremoveThread : ThreadID}
\]

\[
\text{channel CEEproceed : ThreadID}
\]
While the concept of objects is mainly handled by the core execution environment, the scheduler must have some notion of object identifiers in order to manage locks on objects. This is provided by the ObjectID type, which may simply represent an opaque pointer to the object. Object identifiers are drawn from the same space as virtual machine words, as that is how objects are referenced by bytecode instructions.

\[
\text{ObjectID} = \text{Word} \quad \text{null : ObjectID}
\]

The SCJVM scheduler also manages interrupts, which also have unique identifiers. The precise set of identifiers will likely depend on what interrupt vectors the hardware offers.

\[[\text{InterruptID}]\]

The hardware interrupts, along with the interrupt identifier, are received through the HWinterrupt channel.

\[
\text{channel HWinterrupt : InterruptID}
\]

The hardware is also required to permit enabling and disabling interrupts. This is represented in the model by the channels HWenableInterrups and HWdisableInterrups.

\[
\text{channel HWenableInterrups, HWdisableInterrups}
\]

Although it is mainly left implementation-defined as to what interrupts are offered, it is required that there is a clock interrupt that is fired at regular intervals. This interrupt is not directly handled by the scheduler but is instead used by the real-time clock described in the next section.

\[
\text{clockInterrupt : InterruptID}
\]

When the real-time clock has an alarm trigger, it passes on the clock interrupt to the scheduler to run a handler for it. In this model that is represented by the RTCclockInterrupt channel.

\[
\text{channel RTCclockInterrupt}
\]

The SCJVM scheduler offers services made available through channels. Similarly to the memory manager channels, the channels are named after the service names given in Table 3.2 prefixed with S to indicate that they are handled by the scheduler. Services with both inputs and outputs have an additional return channel, named with the suffix Ret. Services that provide an output and take no inputs simply have one channel on which output is communicated. For brevity we do not include the full channel list here.

As with the memory manager operations, each operation of the scheduler reports whether or not an error occurred and, if so, what error. These error values are of type SReport and are reported over the channel Sreport.

\[
\text{SReport ::= Okay | NonexistentThread | ThreadAlreadyStarted | ...}
\]

\[
\text{channel Sreport : SReport}
\]

With the channels and datatypes declared, we begin the process declaration.

\[
\text{process Scheduler} \equiv \text{begin}
\]

We cover each part of the scheduler model in a separate subsection: information about threads in Section 3.4.2.1, the priority scheduler in Section 3.4.2.2, priority ceiling emulation in Section 3.4.2.3, and interrupt handling in Section 3.4.2.4. Finally, we describe how the Z schemas are lifted to Circus operations in Section 3.4.2.5.
3.4.2.1 Threads

The SCJVM scheduler manages threads and stores information about them. The thread information stored by the scheduler is represented in the ThreadInfo schema, defined below. The scheduler stores the class, identifier and arguments for the initial method executed by each thread. Each thread also has a base and current priority, which may change due to the priority ceiling emulation system described later. Each of these pieces of thread information is represented via a partial function from thread identifiers to the type of the information and all the functions are required to have the same domain. It is required that the current priority is not less than the base priority as the priority can only be temporarily raised, not lowered.

\[
\text{ThreadInfo} \\
\text{threadClass : ThreadID} \rightarrow \text{ClassID} \\
\text{threadMethod : ThreadID} \rightarrow \text{MethodID} \\
\text{threadArgs : ThreadID} \rightarrow \text{seq Word} \\
\text{basePriority : ThreadID} \rightarrow \text{Priority} \\
\text{currentPriority : ThreadID} \rightarrow \text{Priority} \\
\]

\[
\text{dom threadClass = dom threadMethod = dom threadArgs = } \\
\text{dom currentPriority = dom basePriority} \\
\forall t : \text{dom currentPriority} \bullet \text{currentPriority} t \geq \text{basePriority} t
\]

Because a lot of operations, particularly those involved in priority ceiling emulation, only change the currentPriority, we define an additional schema, FixedThreadInfo, that specifies all components of ThreadInfo except currentPriority remain the same.

\[
\text{PreserveThreadInfo} == \Xi \text{ThreadInfo} \setminus (\text{currentPriority})
\]

We also define an operation RemoveThreadInfo, that removes an input identifier, thread?, from the domain of all the functions in ThreadInfo.

\[
\Delta \text{ThreadInfo} \\
\text{thread? : ThreadID} \\
\text{threadClass'} = \{\text{thread?}\} \triangleleft \text{threadClass} \\
\text{threadMethod'} = \{\text{thread?}\} \triangleleft \text{threadMethod} \\
\text{threadArgs'} = \{\text{thread?}\} \triangleleft \text{threadArgs} \\
\text{basePriority'} = \{\text{thread?}\} \triangleleft \text{basePriority} \\
\text{currentPriority'} = \{\text{thread?}\} \triangleleft \text{currentPriority}
\]

SCJVM threads may be in one of several states at any given time. The information on which threads are in each state is represented as specified in the ThreadManager schema, which contains sets of identifiers recording the threads that are in each of these states, along with the identifiers of some special threads. An SCJVM thread may be either waiting to start, started but not running (because a higher priority thread is running), blocked or the currently running thread. There is only a single current thread, so it is represented by a single identifier, whereas multiple threads may be in the other states, so they are represented as sets of threads. In addition to these states, there is also a set of free thread identifiers. There must also be an idle thread that is always available to run and not in any of the sets free, created, started or blocked (though it may be the current thread). One thread must be designated as the main thread, which cannot be freed or stopped, only blocked or preempted. The thread states partition the space of thread identifiers into free identifiers, created but not started threads, started but not running threads, blocked threads and the current or idle threads. The current and idle thread identifiers are in a partition together, since idle may be the same as current. The main thread must be either started, blocked or the current thread, since it cannot be destroyed and there is never a time when it is not started. The main thread and the idle thread must be different.
Initially all thread identifiers are in the free set, except the ones used for the idle and main threads. The main thread is initially the current thread and there are no threads in any of the other states.

Because most operations only affect two sets in ThreadManager, we define some operations to specify that only two are changed. These are named using Change followed by the names of the components permitted to change. Since they are all similar in form we only present the first one here

```
ChangeFreeCreated == ∼ ThreadManager \ (free, created)
```

Having defined all the relevant information concerning threads, we now describe how they are scheduled according to their priorities.

### 3.4.2.2 Priority Scheduler

The SCJVM scheduler is a preemptive priority scheduler, which stores queues of thread identifiers for each priority. We define these queues and the operations upon them separately. A queue is represented using a Z sequence. We take the front of the sequence to be the back of the queue to ensure the correct ordering of queue elements when the priority queues are flattened into a single queue. The sequence is taken to be injective since no thread identifier can occur more than once in the same queue.

```
Queue == iseq ThreadID
```

We define operations `pushFront` and `pushBack` to push identifiers onto a given queue. As mentioned above, the first element of the sequence is taken to be the last element of the queue, so `pushFront` pushes to the back of sequence, `pushBack` pushes to the front of the sequence.

```
pushFront, pushBack : ThreadID → Queue → Queue

∀ thread : ThreadID; queue : Queue ●
pushFront thread queue = queue ^ (thread) ∧
pushBack thread queue = (thread) ^ queue
```

We also provide operations `queueFront` and `removeFromQueue` to obtain the identifier at the front of the queue and remove an identifier from a queue. The `queueFront` operation is simply the last operation for taking last element of the sequence, and the `removeFromQueue` operation uses the Z filtering operator, \(\mid\), to filter the identifier out of the queue.

```
queueFront : Queue → ThreadID
removeFromQueue : ThreadID → Queue → Queue

queueFront = last
∀ thread : ThreadID; queue : Queue ●
removeFromQueue thread queue = queue \mid \{thread\}
```
The state of the scheduler is defined in the `Scheduler` schema, which contains all the components of `ThreadInfo` and `ThreadManager`. The scheduler also contains a queue of thread identifiers for each priority. We represent the priority queues by a function from `Priority` to the `Queue` type defined above. There is additionally a set of threads identifiers that identify threads executing interrupt handlers. The identifiers in the priority queues must be all the identifiers of the `started` threads and the identifier of the `current` thread, but not the identifier of the `idle` thread, even if it is the `current` thread. This is because the `idle` thread is only selected to run if there are no other threads available and so does not fit into the normal ordering of threads. The identifiers in two different priority queues are required to be disjoint, since a thread cannot have two different priorities. A related requirement is that each thread identified in the priority queues have its current priority the same as the priority of its queue. The functions defined in `ThreadInfo` are related to the states defined in `ThreadManager` by requiring that the domain of `currentPriority` (and hence all the other functions) be the union of all the thread sets except `free`. The interrupt threads must be within the `started` and `current` threads, since interrupt threads cannot self suspend and are created as needed.

```
Scheduler
| ThreadInfo
| ThreadManager
priorityQueues : Priority → Queue
interruptThreads : P ThreadID

∪{ q : ran priorityQueues • ran q } = (started ∪ {current}) \ {idle}
disjoint (λ p : Priority • ran (priorityQueues p))
∀ p : Priority • ∀ t : ran(priorityQueues p) • currentPriority t = p
dom currentPriority = created ∪ started ∪ blocked ∪ {current, idle}
interruptThreads ⊆ started ∪ {current}
```

Since operations usually only need to update one priority queue, we provide a function to simplify such updates.

```
updatePriorityQueue
| : Priority → (Queue → Queue) → (Priority → Queue) → (Priority → Queue)

∀ priority : Priority; f : Queue → Queue; pqs : Priority → Queue •
updatePriorityQueue priority f pqs = pqs ⊕ {priority → f (pqs priority)}
```

The initialisation of the scheduler is as specified in the schema `SchedulerInit`, defined below. The thread states, together with the `main` and `idle` thread identifiers are initialised as described by the schema `ThreadManagerInit`. The `main` thread has a priority supplied as an input to the initialisation and the `idle` thread has the lowest possible priority. These are initially used for both the base and current priorities. The initial values of the other components of `ThreadInfo` do not matter since they are only used for starting a thread. The queue for the `main` thread’s priority initially contains only the `main` thread’s identifier and all other priority queues are empty. There are initially no interrupt threads.

```
SchedulerInit
| Scheduler
| ThreadManagerInit
mainPriority? : ThreadPriority

currentPriority' = basePriority' = {main → mainPriority?, idle → minSwPriority}
priorityQueues' mainPriority? = ⟨main⟩
∀ p : Priority • p ≠ mainPriority? • priorityQueues' p = ⟨⟩
interruptThreads' = ∅
```

When a new thread is created, its class, method, arguments and priority are stored. The operation of creating a new thread is defined by the schema `ThreadCreate`, which takes this information as input. The
new thread is given an identifier from the free identifier set; that identifier is an output from the operation. The functions in ThreadInfo are updated to include the new thread identifier and the information in put to the operation. The priority input to the operation is used for both the current and base priority. The new thread’s identifier is removed from the free identifiers and added to the created set. The other thread states are unaffected and the interrupt threads do not change as this operation is not used for creating interrupt threads.

\[
\begin{align*}
\text{ThreadCreate} & \quad \Delta \text{Scheduler} \\
& \quad \text{ChangeFreeCreated} \\
& \quad \text{priority}\!: \text{ThreadPriority} \\
& \quad \text{class}\!: \text{ClassID} \\
& \quad \text{method}\!: \text{MethodID} \\
& \quad \text{args}\!: \text{seq Word} \\
& \quad \text{newID}\!: \text{ThreadID} \\
& \quad \text{newID}! \in \text{free} \\
& \quad \text{threadClass}' = \text{threadClass} \oplus \{\text{newID}! \mapsto \text{class}!\} \\
& \quad \text{threadMethod}' = \text{threadMethod} \oplus \{\text{newID}! \mapsto \text{method}!\} \\
& \quad \text{threadArgs}' = \text{threadArgs} \oplus \{\text{newID}! \mapsto \text{args}!\} \\
& \quad \text{currentPriority}' = \text{currentPriority} \oplus \{\text{newID}! \mapsto \text{priority}!\} \\
& \quad \text{basePriority}' = \text{basePriority} \oplus \{\text{newID}! \mapsto \text{priority}!\} \\
& \quad \text{free}' = \text{free} \setminus \{\text{newID}!\} \\
& \quad \text{created}' = \text{created} \cup \{\text{newID}!\} \\
& \quad \text{interruptThreads}' = \text{interruptThreads} \land \text{priorityQueues}' = \text{priorityQueues}
\end{align*}
\]

A thread is started by adding it to the back of the queue for its priority. This is described by the schema ThreadStart, which takes as input the identifier of the thread to be started. The thread is required to be in the created set to ensure it has been created but not yet started. The identifier of the thread is removed from the created set and added to the started set. The sets for the other thread states remain unaffected. The priority queues are updated by adding the thread’s identifier to the queue for its currentPriority using the pushBack function.

\[
\begin{align*}
\text{ThreadStarts} & \quad \Delta \text{Scheduler} \\
& \quad \text{ChangeCreatedStarted} \\
& \quad \text{Σ ThreadInfo} \\
& \quad \text{toStart}\!: \text{F ThreadID} \\
& \quad \text{previous}\!: \text{ThreadID} \\
& \quad \text{toStart} \subseteq \text{created} \\
& \quad \text{created}' = \text{created} \setminus \text{toStart} \\
& \quad \forall t : \text{toStart}\! \bullet \text{priorityQueues}' = \text{updatePriorityQueue} (\text{currentPriority}\! t) (\text{pushBack}\! t) \text{priorityQueues} \\
& \quad \exists \text{started}0 == \text{started} \cup \text{toStart}! \bullet \text{PickNewCurrent}[\text{started}0/\text{started}] \\
& \quad \text{interruptThreads}' = \text{interruptThreads}
\end{align*}
\]

46
Starting a thread does not necessarily cause it to run since it is the thread at the front of the highest priority queue that runs. The selection of a thread to run is described by the schema RunThread and occurs whenever a change has taken place that may affect which thread is running, such as a thread starting or changing priority. The new current thread is chosen by first flattening the priority queues using the distributed concatenation operator, \( \smallfrown / \), which joins the queues together in order in one long queue with the higher priority threads nearer the front (recall that we are taking the back of the sequence as the front of the queue to enable this operation to be applied). The idle thread is then placed at the back of the queue to ensure it is not empty and the thread at the front of the queue is taken as the new current thread. The old current thread is added to the started thread set. The new current thread and the idle thread are both removed from the set, to ensure the idle thread is not in the set and in case the old current thread is the same as either of those. The other state components are unaffected as this operation does not make any changes besides choosing a new current thread.

Destruction of a thread is handled by the ThreadDestroy schema, which takes the identifier of the thread to be destroyed as input. The thread identifier is required not to be in free, or to be the main, idle or current thread. The thread also cannot be an interrupt thread, since this operation is for destroying ordinary threads, not signalling the end of an interrupt. The thread’s identifier is removed from the domain of all the thread information functions, removing the information about it from the scheduler’s state as specified by RemoveThreadInfo. The thread is added to the free thread set, and removed from the created, started and blocked threads. The thread’s identifier is removed from its priority queue by the removeFromQueue operation. The current, main, idle and interrupt threads are unaffected as they cannot be destroyed by this operation.

<table>
<thead>
<tr>
<th>ThreadDestroy</th>
<th>Δ Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>RemoveThreadInfo</td>
<td>thread? : ThreadID</td>
</tr>
<tr>
<td>previous! : ThreadID</td>
<td></td>
</tr>
<tr>
<td>thread? \in blocked \cup started \cup {current}</td>
<td></td>
</tr>
<tr>
<td>thread? \notin {idle, main}</td>
<td></td>
</tr>
<tr>
<td>free' = free \cup {thread?}</td>
<td></td>
</tr>
<tr>
<td>created' = created</td>
<td></td>
</tr>
<tr>
<td>blocked' = blocked \setminus {thread?}</td>
<td></td>
</tr>
<tr>
<td>\exists priority == currentPriority thread? • priorityQueues' = updatePriorityQueue priority (removeFromQueue thread?) priorityQueues</td>
<td></td>
</tr>
<tr>
<td>\exists started0 == started \setminus {thread?} • PickNewCurrent[started0]/started</td>
<td></td>
</tr>
<tr>
<td>interruptThreads' = interruptThreads</td>
<td></td>
</tr>
</tbody>
</table>

An SCJVM thread may be suspend itself, causing it to pause running and block. This operation is defined by the schema ThreadSuspend, which does not affect the thread information and takes no inputs. A thread can only suspend itself and so the thread to be suspended must be the current thread. The current thread must not be the idle thread or an interrupt thread since those threads cannot self-suspend. The current thread is added to the blocked thread set and filtered out from the threads in its priority queue as in DestroyThread. A new current thread is chosen much as in RunThread, but using the updated priority queues with the suspended thread removed. The newly chosen current thread is removed from the started thread set. The other state components remain unaffected.
A suspended thread remains suspended until it is signalled to resume by the operation defined by `ThreadResume`. This operation does not affect the thread information and takes as input the identifier of the thread to be resumed, which must be one that is blocked. Its identifier is removed from the `blocked` set and added to the `started` set. The other thread sets are unaffected. The resumed thread is placed at the back of the queue for its current priority.

It should be noted that SCJ programs do not have direct access to the functionality of suspending and resuming threads. It is provided for the infrastructure to implement things such as device access and mission initialisation.

We have specified threads and how they are scheduled, but the interactions between threads when taking the lock on an object must also be specified. This is covered next, where we describe the priority ceiling emulation policy that SCJ requires for locking.

### 3.4.2.3 Priority Ceiling Emulation

The SCJVM must support priority ceiling emulation and locking of objects. This is accounted for by `PCEScheduler`, which is an extension of the `Scheduler` state to include information required for priority ceiling emulation and locking. The state contains a function `priorityCeiling` that associates a ceiling priority to each object identifier. This function is total as the scheduler does not need to be aware of which objects actually exist and can simply assign a ceiling priority to all the possible identifiers, locking on those passed to it from outside. There are also functions `lockHolder` and `lockCount` that map object identifiers to the identifier of the thread that holds each object’s lock and the number of times each lock has been taken (a thread may retake the lock on an object it has already locked, forming multiple nested locks). These functions are partial as it only makes sense to hold this information for an object that has been locked. For convenience, we also have a function, `locksHeld`, mapping threads to the sets of objects
they hold locks for, which may be empty. The domains of the functions \textit{lockCount} and \textit{lockHolder} are required to be the same, and the range of \textit{lockHolder} is required to be within the started and current thread identifiers since those are the only threads that can hold locks. The \textit{locksHeld} function is defined to map the relational image of the inverse of \textit{lockHolder} for a given thread. The current priority of a thread under priority ceiling emulation is given by the maximum of the thread’s base priority and the priority ceilings of all the objects it holds locks on.

<table>
<thead>
<tr>
<th>\textit{PCEScheduler}</th>
<th>\textit{Scheduler}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{priorityCeiling} : ObjectID \rightarrow Priority</td>
<td></td>
</tr>
<tr>
<td>\textit{lockHolder} : ObjectID \rightarrow ThreadID</td>
<td></td>
</tr>
<tr>
<td>\textit{lockCount} : ObjectID \rightarrow \mathbb{N}_1</td>
<td></td>
</tr>
<tr>
<td>\textit{locksHeld} : ThreadID \rightarrow \mathbb{P} ObjectID</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{dom lockCount} = \text{dom lockHolder} \\
\text{ran lockHolder} \subseteq \text{started} \cup \{\text{current}\} \\
\forall t : \text{ThreadID} \bullet \text{locksHeld} t = \text{lockHolder}^{-1} \{t\} \\
\forall t : \text{ThreadID} \bullet \\
\quad \text{currentPriority} t = \\
\quad \text{max} \left( \{\text{basePriority} t\} \cup \{ o : \text{locksHeld} t \bullet \text{priorityCeiling} o \} \right)
\]

The \textit{PCEScheduler} is initialised as for \textit{Scheduler}, with additional initialisation of the state components introduced in \textit{PCEScheduler}. Initially all objects have the default priority ceiling, which is the maximum software priority. The \textit{lockHolder} and \textit{lockCount} maps are empty, since no locks are initially held. The state invariant defining \textit{locksHeld} is sufficient to uniquely determine it so an explicit initialisation is not provided for it here.

<table>
<thead>
<tr>
<th>\textit{PCESchedulerInit}</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textit{PCEScheduler} \text{'}</td>
</tr>
<tr>
<td>\textit{SchedulerInit}</td>
</tr>
</tbody>
</table>

\[
\forall x : \text{ObjectID} \bullet \text{priorityCeiling} x = \text{maxSwPriority} \\
\text{lockHolder} x = \emptyset \\
\text{lockCount} x = \emptyset
\]

Taking the lock on an object is specified by considering two cases: the case in which the lock is not held and the case in which an object is attempting to retake a lock it already holds. The handling of the first case is described by the schema \textit{PCETakeLock}, defined below, which does not affect the state in \textit{ThreadManager} or \textit{ThreadInfo}, other than updating \textit{currentPriority}, and takes as input the identifier of the object to lock. The object is required to not be in the domain of \textit{lockHolder} for this case to ensure it does not already have a thread locking it, and the priority of the current thread must also be less than the object’s priority ceiling. The object is added to \textit{lockHolder}, associated with the current thread, and also to \textit{lockCount}, associated with 1, since this is the first time the thread has taken the lock on this object. The current priority of the current thread is set to the maximum of the thread’s current priority and the priority ceiling of the object. The priority queues are updated by first removing the current thread’s identifier from the queue for its old priority then adding it to the front of the queue for its new priority. The other state components are unchanged.
The second case, where a thread already holds the lock on an object, is described by \texttt{PCERetakeLock}, which is similar to \texttt{PCETakeLock} in that it preserves the state of \texttt{ThreadManager} and \texttt{ThreadInfo} and takes an object identifier as input. In this case, the object is required to be in the domain of \texttt{lockHolder} but it must map to the identifier of the current thread, since a thread cannot take a lock held by another thread. The \texttt{lockCount} value for the object is incremented by one and the values for other objects are unchanged. All other state components are unchanged.

The operation of releasing the lock on an object is similarly split into two cases: the case in which the lock is held only once, and the case in which a thread holds multiple nested locks on the same object. The first case is described by \texttt{PCEReleaseLock}, defined below, which does not affect the state in \texttt{ThreadManager} or \texttt{ThreadInfo}, other than updating \texttt{currentPriority}, and takes as input the identifier of the object to be unlocked. The object is required to be in the set of locks held by the current thread and \texttt{lockCount} must be 1. The object’s identifier is removed from the domain of \texttt{lockHolder} and \texttt{lockCount} since it is no longer locked by any thread. The current priority of the current thread is set to the maximum of the thread’s base priority and the priority ceilings of any other objects it holds locks on. The current thread is placed at the front of the priority queue for its new priority in the same way as in \texttt{PCETakeLock}. The other state components are unaffected.
The second case, where the lock held has been taken more than once by the same thread, is described by the schema \texttt{PCEReleaseNestedLock}. This does not affect the components of \texttt{ThreadManager} or \texttt{ThreadInfo} and takes the identifier of the object to be unlocked as input, much like \texttt{PCEReleaseLock}. In this case, the object must be in the set of locks held by the current thread and the \texttt{lockCount} value for the object is required to be greater than 1. The \texttt{lockCount} value for the object is decremented and the values for all other objects are unaffected. The other state components are unchanged.

The priority ceiling of an object can be set using the operation described by \texttt{PCESetPriorityCeiling}, which takes an object identifier and a ceiling priority as input and does not affect the state of \texttt{Scheduler}. The object input must not have its lock held by any object, i.e. it must not be in the domain of \texttt{lockHolder}. The object is updated in the \texttt{priorityCeiling} map so that it maps to the given ceiling priority. The lock state is unaffected by this operation.

In order to prevent deadlock, it is forbidden for a thread to suspend itself while holding a lock. To enforce this condition, we extend \texttt{ThreadSuspend} to the schema \texttt{PCESuspend}, which adds the precondition that
the current thread must hold no locks.

Next, we specify interrupt handlers, which are similar to threads but require some extra handling in how they are started and finished.

### 3.4.2.4 Interrupt Handling

The SCJVM scheduler must manage interrupts, tracking the priority of each interrupt, the handler attached to it and the backing store to be used as the allocation context of the handler. The PCEScheduler state is extended to handle interrupts as specified in the InterruptScheduler schema. The state contains a function mapping each interrupt to a priority, which must be an interrupt priority. The state additionally contains functions mapping interrupts to their handlers, which are pairs of class and object identifiers representing interrupt handlers written as Java objects, and the backing store used as their allocation context. These functions are partial since not every interrupt will have a handler associated to it. There is also a set of identifiers of interrupts masked by currently running interrupts and a boolean flag that indicates if interrupts are enabled or not. Finally, since interrupts are managed as threads, there is a map, interruptThreadMap, from interrupt identifiers to threads running the interrupt handlers, which is partial because such a thread only exists for a running interrupt handler. An interrupt that is running is always masked, so the domain of interruptThreadMap must be a subset of the masked thread set. The range of interruptThreadMap must also be the same as the set of interrupt threads defined earlier in the priority scheduler, since they are the threads used for interrupts. The domains of the interrupt handler and allocation context functions must be the same since they both apply only to interrupts with attached handlers. The base priority of each interrupt thread must be the same as the priority for the corresponding interrupt. This is specified by requiring that the composition of interruptThreadMap and the base priority map be a subset of the interrupt priority map so that an interrupt handler’s actual base priority will match up with its stored priority.

Initialisation of InterruptScheduler occurs as specified in InterruptSchedulerInit, which is based on PCESchedulerInit. Initially, no interrupts have handlers attached, so the interrupt handler and allocation context maps are empty. The interruptThreadMap and the maskedInterrupts set are also empty, since there are no interrupt handlers running. Interrupts are initially enabled, so interruptsEnabled is True. The interrupt priorities are unspecified as they are implementation-defined and cannot be changed by the user.
Attaching a handler to a specified interrupt is provided for by the operation defined by the schema \textit{InterruptAttachHandler}, which preserves the state of \textit{PCEScheduler} and takes as input the identifier of the interrupt to attach the handler to along with the class, object and allocation context of the handler to attach. The interrupt handler map is updated to associate the class and object with the specified interrupt. The interrupt allocation context map is similarly updated to associate the backing store given as allocation context to the interrupt. The other state components that record information for interrupts are unaffected.

\begin{align*}
\text{InterruptAttachHandler} \quad \\
\Delta \text{InterruptScheduler} \quad \\
\Xi \text{PCEScheduler} \quad \\
\text{interrupt}? : \text{InterruptID} \\
\text{handlerClass}? : \text{ClassID} \\
\text{handlerObject}? : \text{ObjectID} \\
\text{ac}? : \text{BackingStoreID} \\
\text{stack}? : \text{StackID} \\
\text{interruptHandler}' = \text{interruptHandler} \oplus \{\text{interrupt}? \mapsto \text{handlerClass}? , \text{handlerObject}?\} \\
\text{interruptAC}' = \text{interruptAC} \oplus \{\text{interrupt}? \mapsto \text{ac}?\} \\
\text{interruptStack}' = \text{interruptStack} \oplus \{\text{interrupt}? \mapsto \text{stack}?\} \\
\text{interruptPriority}' = \text{interruptPriority} \\
\text{maskedInterrupts}' = \text{maskedInterrupts} \\
\text{interruptThreadMap}' = \text{interruptThreadMap} \\
\text{interruptsEnabled}' = \text{interruptsEnabled}
\end{align*}

Detaching an interrupt handler is defined by \textit{InterruptDetachHandler}, which takes a interrupt identifier as input and removes it from the \textit{interruptHandler} and \textit{interruptAC} maps. We omit its definition here as it is similar to \textit{InterruptAttachHandler}.

Getting the priority of a given interrupt is described by the \textit{InterruptGetPriority} schema. We omit it here since it is a simple operation that just applies \textit{interruptPriority} to its input.

The operations of enabling and disabling interrupts simply set the value of the boolean flag indicating whether or not interrupts are enabled. Because these operations only affect one state component, we define a schema \textit{InterruptEnableFixedVars} to more briefly state that all other state components remain the same.

\begin{align*}
\text{InterruptEnableFixedVars} = = \Xi \text{InterruptScheduler} \setminus (\text{interruptsEnabled})
\end{align*}

As the operations of enabling and disabling interrupts are similar, we just present the \textit{InterruptEnable} schema here, omitting the \textit{InterruptDisable} schema.

\begin{align*}
\text{InterruptEnable} \\
\Delta \text{InterruptScheduler} \\
\text{InterruptEnableFixedVars} \\
\text{interruptsEnabled}' = \text{True}
\end{align*}
When an interrupt is fired, it is handled as described by HandleInterrupt, defined below. The identifier of the interrupt to be handled is passed as an input to the operation, and the handler thread identifier and allocation context are output so that they can be communicated to the core execution environment and memory manager. A boolean is also output to indicate whether or not the interrupt was actually handled. For the interrupt to be handled, interrupts must be enabled, the interrupt must not be masked, and the interrupt must have a handler attached. The new interrupt handler thread’s identifier is chosen from the free thread identifiers. That identifier is removed from the free thread identifiers and added to the started thread identifiers. The current and base priority of the new thread are set to the given interrupt’s priority and the thread’s identifier is placed at the back of the queue for its priority. The queues for other priorities are unaffected. The set of interrupt threads is updated to include the identifier of the new handler thread and all threads with priority less than or equal to the interrupts priority are added to the set of masked interrupts. The other state components are unaffected, though the thread class, method and argument maps must be updated to include the new thread. It does not matter what values are included for the new thread, but the other values in the maps are required to remain the same. The value in the allocation context map for the interrupt is output and, since the interrupt was handled, the boolean flag output is true.

If interrupts are disabled, the interrupt is masked or the interrupt has no handler attached then it is silently ignored. This is described in the schema IgnoreInterrupt, which does not change the state. The interrupt’s identifier is taken as an input and a boolean is output to indicate that the interrupt was not handled.
When an interrupt handler ends, the interrupt handler thread is destroyed, removing the information about it from the scheduler as described by the schema \texttt{InterruptEnd}. The current thread must be an interrupt thread for this operation to be used. The information about the current thread is removed from the \texttt{ThreadInfo} maps and the current thread’s identifier is filtered out of the queue for its priority. The queues for the other priorities are unaffected. A new current thread is chosen to replace the stopped current thread by taking the last thread in the sequence formed from the idle thread and the distributed concatenation of the priority queues, as in \texttt{RunThread} and \texttt{ThreadSuspend}. The current thread is removed from the interrupt threads set and the masked interrupts are those with priority less than or equal to the priorities of any threads still running. The other state components are unaffected.

The schemas declared so far are also lifted to robust actions. This concludes the \texttt{Z} portion of the scheduler model. The operations must now be lifted to \texttt{Circus} actions accessed via the channels declared earlier and the interaction with interrupt signals must be specified.

\begin{equation}
\text{ThreadSwitchInfo ::=} \\
\hspace{1em} \text{start} (\langle \text{ThreadID} \times \text{BackingStoreID} \times \text{StackID} \times \text{ClassID} \times \text{MethodID} \times \text{seq Word} \rangle) \\
\hspace{1em} | \hspace{1em} \text{switch} (\langle \text{ThreadID} \times \text{ThreadID} \rangle)
\end{equation}

\begin{equation}
\text{SwitchManager} \\
\hspace{1em} \text{switchQueue : seq ThreadSwitchInfo} \\
\hspace{1em} \text{phantomCurrent : ThreadID}
\end{equation}

\begin{equation}
\forall t : \text{ThreadID} \bullet \text{switch} (t, t) \notin \text{ran switchQueue} \\
\sim \exists t1, t2 : \text{ThreadID} \bullet \langle \text{switch} (t1, t2), \text{switch} (t2, t1) \rangle \text{ infix switchQueue}
\end{equation}
PushThreadSwitchNormal
\[\Delta \text{SwitchManager} \]
from?, to?: ThreadID

\[
\text{from?} \neq \text{to?} \land \neg (\text{switch} (\text{to?}, \text{from?})) \text{ prefix switchQueue}\\
\text{switchQueue'} = (\text{switch} (\text{from?}, \text{to?})) \wedge \text{switchQueue}\\
\text{phantomCurrent'} = \text{phantomCurrent}
\]

PushThreadSwitchSelf
\[\Xi \text{SwitchManager} \]
from?, to?: ThreadID

\[
\text{from?} = \text{to?}
\]

PushThreadSwitchReverse
\[\Delta \text{SwitchManager} \]
from?, to?: ThreadID

\[
(\text{switch} (\text{to?}, \text{from?})) \text{ prefix switchQueue}\\
\text{switchQueue'} = \text{tail switchQueue}\\
\text{phantomCurrent'} = \text{phantomCurrent}
\]

PushThreadSwitch ==
\[\text{PushThreadSwitchNormal} \land \text{PushThreadSwitchSelf} \land \text{PushThreadSwitchReverse}\]

PushThreadStart
\[\Delta \text{SwitchManager} \]
thread?: ThreadID
bsid?: BackingStoreID
stack?: StackID
class?: ClassID
method?: MethodID
args?: seq Word

\[
\text{switchQueue'} = (\text{start} (\text{thread?}, \text{bsid?}, \text{stack?}, \text{class?}, \text{method?}, \text{args}?) ) \wedge \text{switchQueue}
\]

PopThreadSwitch
\[\Delta \text{SwitchManager} \]
from!, to!: ThreadID

\[
\text{switchQueue} \neq \emptyset\\
\text{head switchQueue} \in \text{ran switch}\\
\text{from!} = ((\text{switch} \sim) (\text{head switchQueue})).1\\
\text{to!} = ((\text{switch} \sim) (\text{head switchQueue})).2\\
\text{switchQueue'} = \text{tail switchQueue}
\]
3.4.2.5 Scheduler Operations

The scheduler model offers the services detailed in this section. The operations described in Section 3.2 are all implemented here. Some additional actions are also defined for handling interrupts.

The state of the scheduler process is InterruptScheduler, which contains the state of the priority ceiling emulation manager and priority scheduler as well as the interrupt manager state.

\[ \text{state} \text{ InterruptScheduler} \land \text{SwitchManager} \]

The scheduler is initialised with the main thread’s priority and allocation context via the Sinit channel and the initial state is as described by InterruptSchedulerInit.

\[ \text{Init} \triangleq \text{var mainAC : BackingStoreID} \bullet \text{Sinit?mainPriority?mainAC} \rightarrow (\text{InterruptSchedulerInit}) \]

The services that output constant priority values, such as getMaxSoftwarePriority, simply output the relevant value over their channel and then output a report of Sokay, as shown in the example of the GetMaxSoftwarePriority action below.

\[ \text{GetMaxSoftwarePriority} \triangleq \text{SgetMaxSoftwarePriority!maxSwPriority} \rightarrow \text{Sreport}!\text{Sokay} \rightarrow \text{Skip} \]

The action of getting the main and current threads are specified in a similar way, since they just output the main and current thread identifiers respectively. The other operations are lifted to Circus actions from the Z schemas defined earlier. The correspondence between the services described in section Section 3.2 the Circus actions described here, and the Z schemas defined earlier is shown in Table 3.5. These actions follow a common pattern, seen in the GetInterruptPriority action below, which is similar to the lifting of the memory manager operations in Section 3.4.1. The signal to perform the operation, along with the inputs to the operation, is communicated via the operation’s channel. The operation is then performed as specified by the corresponding Z schema. Any outputs are communicated on a return channel and the error report from the operation is sent on the Sreport channel.

\[ \text{GetInterruptPriority} \triangleq \text{var priority : Priority; report : SReport} \bullet \text{SgetInterruptPriority?interrupt} \rightarrow (\text{RInterruptGetPriority}); \text{SgetInterruptPriorityRet!priority} \rightarrow \text{Sreport!report} \rightarrow \text{Skip} \]

Many of the actions, however, deviate from this pattern since the scheduler must respond to external events and communicate with the core execution environment.

\[ \text{HandleThreadSwitch} \triangleq \text{val report : SReport; val from, to : ThreadID} \bullet \text{if report = Sokay} \rightarrow (\text{PushThreadSwitch}) \]
\[ \text{fi} \]

57
An example of an action that makes use of Schedule is ResumeThread, shown below. This action follows much the same format as that shown for GetInterruptPriority above, but ends with the Schedule action.

\[
\text{ResumeThread} \triangleq \text{var } \text{thread}, \text{previous : ThreadID; report : SReport} \triangleright
\]

\[
\text{SresumeThread?thread} \rightarrow (\text{RThreadResume}); \\
\text{Sreport!report} \rightarrow \text{HandleThreadSwitch}(\text{report}, \text{previous}, \text{current})
\]

The other actions that are followed by Schedule are ReleaseLock, SuspendThread, StartThread, and EndInterrupt.

The StartThreads action requires some additional communication with the core execution environment and must loop over all of the threads in the input set. The input set threads, containing triples of thread, backing store and stack identifiers, is received via the StartThreads channel and an error report value, finalreport, is initialised to $\text{Sokay}$. Using replicated sequential composition over the pairs in threads, the thread identifier, thread, the backing store identifier, bsid, and the stack identifier, stack, are extracted, and the scheduler’s state is updated as described by the RThreadStart schema. If there is no error report, then the core execution environment is informed of the thread’s starting via the CEEstartThread channel, and supplied with the information for the method the thread is to run and the identifier bsid. A backing store is only input to the scheduler in this action, not when the thread is created, since SCJ requires the memory areas for threads to be created as the threads are started. In the case where an error is reported, that report is stored as the finalreport value, so that an error will be output if there is a problem with any of the input threads. After all the threads have been processed in sequence, the finalreport is output over the Sreport channel and a new thread is selected to run as in the Schedule
The action of enabling interrupts must signal the hardware using the `HWenableInterrupts` channel in addition to updating the scheduler’s state as described in `RInterruptEnable`.

\[
\text{EnableInterrupts} \equiv \begin{align*}
\text{var} & \quad \text{report} : \text{SReport} \quad \text{\bullet} \\
\text{SenableInterrupts} & \rightarrow (\text{RInterruptEnable}) \\
\text{HWenableInterrupts} & \rightarrow \text{Sreport}! \text{report} \rightarrow \text{Skip}
\end{align*}
\]

Disabling of interrupts is similar, using the `HWdisableInterrupts` channel.

Interrupt handling must be done in response to a signal from hardware, so it is a separate action although it is not one of the public SCJVM services. An interrupt is handled by calling the `handle()` method of the interrupt handler object, which is represented by a method identifier `handleID` in our model.

\[
\text{handleID} : \text{MethodID}
\]

We define the handling of a given interrupt as a `Circus` action, `Handle`, which takes the identifier of the interrupt as a parameter. The interrupt handling specification is split into two cases: the case where the interrupt is actually handled, as described in `HandleInterrupt`, and the case where the interrupt is ignored, as described in `IgnoreInterrupt`. The \( \text{Z} \) schemas to handle each case are placed in disjunction and a `Circus` if statement is used to check the boolean output in order to determine which case took effect.

If the interrupt was handled, the information about the newly created thread is transmitted to the core execution environment via the `CEEStartThread` channel. We present the information transmitted to the core execution environment as a 5-tuple. The allocation context and class passed to the core execution environment are those given when the handler was attached to the interrupt. The method identifier used is `handleID` and the object identifier of the handler object is given as the only argument of the method.

\[
\text{Schedule} \rightarrow \text{SelectNewThread}
\]

In the case where the interrupt was ignored, the action simply terminates.

\[
\text{Handle} \equiv \begin{align*}
\text{val} & \quad \text{interrupt} : \text{InterruptID} \quad \text{\bullet} \\
\text{var} & \quad \text{handler}, \text{previous} : \text{ThreadID}; \ ac : \text{BackingStoreID}; \ handled : \text{B}; \ stack : \text{StackID} \quad \text{\bullet} \\
& \quad (\text{HandleInterrupt} \lor \text{IgnoreInterrupt}) \\
\text{if} & \quad \text{handled} = \text{True} \rightarrow \\
& \quad \text{var} \quad \text{class} : \text{ClassID}; \ method : \text{MethodID}; \ args : \text{seq Word} \quad \text{\bullet} \\
& \quad \text{class}, \text{method}, \text{args} := \quad \text{(interruptHandler interrupt).1}, \text{handleID}, ((\text{interruptHandler interrupt}).2); \\
& \quad (\text{PushThreadStart}([\text{handler}/\text{thread?}, \text{ac}/\text{bsid?}]) \\
& \quad [\text{handled} = \text{False} \rightarrow \text{Skip}
\end{align*}
\]
There are two signals that cause the scheduler to handle an interrupt. The first is an interrupt coming from hardware via the \textit{HWinterrupt} channel. We use input prefixing to require that this interrupt not be the clock interrupt since the clock interrupt is handled by the real-time clock. The interrupt is handled as described by the \textit{Handle} action above.

\begin{align*}
\text{HandleNonclockInterrupt} & \equiv \\
\text{HWinterrupt}\?\text{interrupt} : (\text{interrupt} \neq \text{clockInterrupt}) \rightarrow \text{Handle} (\text{interrupt})
\end{align*}

The second signal that causes interrupt handling is the clock interrupt forwarded from the real-time clock via the \textit{RTCclockInterrupt} channel. This is handled as if it had the identifier of the clock interrupt.

\begin{align*}
\text{HandleClockInterrupt} & \equiv \\
\text{RTCclockInterrupt} \rightarrow \text{Handle} (\text{clockInterrupt})
\end{align*}

\begin{align*}
\text{EndThread} & \equiv \text{var} \text{previous} : \text{ThreadID}; \text{report} : \text{SReport} \bullet \\
\text{SendThread} & \rightarrow (\exists \text{thread} \?== \text{phantomCurrent} \bullet \text{RInterruptEnd} \lor \text{RThreadDestroy}); \\
\text{HandleThreadSwitch} & (\text{report}, \text{previous}, \text{current})
\end{align*}

\begin{align*}
\text{StartThread} & \equiv \\
\text{var} \text{threadStartInfo} \\
& : \text{ThreadID} \times \text{BackingStoreID} \times \text{StackID} \times \text{ClassID} \times \text{MethodID} \times \text{seq Word} \bullet \\
& (\text{switchQueue} \neq \emptyset \land \text{head} \text{switchQueue} \in \text{ran} \text{start}) \land \\
& (\text{PopThreadStart}); \text{CEEstartThread}!\text{threadStartInfo} \rightarrow \text{Skip}
\end{align*}

\begin{align*}
\text{SwitchThread} & \equiv \text{var} \text{from}, \text{to} : \text{ThreadID} \bullet \\
& (\text{switchQueue} \neq \emptyset \land \text{head} \text{switchQueue} \in \text{ran} \text{switch}) \land \\
& (\text{PopThreadSwitch}); \text{CEEswitchThread}!\text{from}!\text{to} \rightarrow \text{phantomCurrent} := \text{to}
\end{align*}

\begin{align*}
\text{Proceed} & \equiv (\text{switchQueue} = \emptyset) \land \text{CEEproceed}!\text{current} \rightarrow \text{Skip}
\end{align*}

The scheduler continuously presents all its operations in a loop. Any operation can be chosen once the previous operation has completed.

\begin{align*}
\text{Loop} & \equiv \text{GetMainThread} \Box \text{MakeThread} \Box \text{StartThreads} \cdots ; \text{Loop}
\end{align*}

The main action of the scheduler process first requires initialisation and then enters the operation loop declared above.

\begin{align*}
& \bullet \text{Init}; \text{Loop} \\
& \text{end}
\end{align*}

This concludes the specification of the scheduler. We have specified threads and information about them, including their priority, whether they are available to run or not, and the method information required to begin execution of the thread. We specified the priority scheduler, which sorts the executable threads into queues by priority and selects the thread at the front of the highest non-empty priority queue to run. This includes the operation to create, start, destroy, suspend and resume threads. A mechanism for locking objects to prevent interference has also been specified, with priority ceiling emulation as a mechanism for avoiding priority inversion problems. We have also described the mechanism by which interrupt handlers are specified and how interrupt processing is performed by starting interrupt threads. Finally, we have lifted the scheduler operations to \textit{Circus} actions accessed via channels and specified the relation of the scheduler to the hardware, memory manager and core execution environment.
3.4.3 Real-time Clock

The SCJVM real-time clock provides an interface to a hardware real-time clock, which is used by the SCJ clock API. The periodic clock interrupt from the hardware is handled by the SCJVM clock and used to manage alarms that trigger when a certain time is reached. If an alarm is set, the interrupt is passed to the scheduler when the alarm triggers, the SCJ API implementation should attach an interrupt handler to it that simply calls the `triggerAlarm()` method of `Clock` for the real-time clock.

The type used for interrupt identifiers is the same as that used by the scheduler. We declare a type `Time`, representing time values using the set of natural numbers. The SCJ API represents time as two numbers representing milliseconds and nanoseconds, but it is easier for the purposes of specification to ignore that detail, since a pair of numbers is only used in the SCJ API because Java has no type large enough to contain the information from both. Instead, we take `Time` values to represent the total number of nanoseconds.

\[
\text{Time} = \mathbb{N}
\]

The clock must have a `precision` value representing the number of nanoseconds between occurrences of the hardware clock interrupt. The `precision` cannot be zero.

\[
\begin{align*}
\text{precision} : & \text{Time} \\
\text{precision} > & 0
\end{align*}
\]

The SCJVM real-time clock relies on the existence of a hardware real-time clock that must be capable of giving the current time in nanoseconds. We declare the channel `HWtime` for receiving the current value of the real-time clock from hardware.

\[
\text{channel HWtime : Time}
\]

We also declare channels for each of the services of the real-time clock described in Section 3.3.

\[
\begin{align*}
\text{channel RTCgetTime, RTCgetPrecision : Time} \\
\text{channel RTCsetAlarm : Time} \\
\text{channel RTCclearAlarm}
\end{align*}
\]

The SCJVM also uses the hardware interrupt channel, `HWinterrupt`, and has a channel to pass the clock interrupt on to the scheduler when appropriate, `RTCclockInterrupt`, which was declared earlier. There is also a type, `RTCReport`, and channel, `RTCreport`, for reporting erroneous inputs to operations, as for the memory manager and scheduler. Having defined the channels and types, the process definition can be presented.

\[
\text{process RealtimeClock} \equiv \begin{align*}
\text{begin} \\
\end{align*}
\]

The real-time clock’s state, `RTCState`, stores the current time value, `currentTime`, of the clock (accurate to within the clock’s precision). The `RTCState` also contains a component to represent the time `currentAlarm` of the alarm set (if any) as well as a boolean component, `alarmSet`, indicating whether or not there is an alarm set. If an alarm is set, then it must be in the future.

\[
\begin{align*}
\text{RTCState} \\
\text{currentTime} : & \text{Time} \\
\text{currentAlarm} : & \text{Time} \\
\text{alarmSet} : & \mathbb{B} \\
\text{alarmSet} = & \text{True} \Rightarrow \text{currentAlarm} \geq \text{currentTime}
\end{align*}
\]

This `RTCState` schema is the state of the `Circus` process modelling the real-time clock.

\[
\text{state RTCState}
\]

61
The clock’s state is initialised with a time value, \( \text{initTime} \), to which the \( \text{currentTime} \) is set. Initially no alarm is set, so \( \text{alarmSet} \) is \text{False} and \( \text{currentAlarm} \) is allowed to take any value since it is unused.

\[
\begin{align*}
\text{RTCInit} \quad \\
\text{RTCState'} \quad \\
\text{initTime'? : Time} \\
\text{currentTime' = initTime?} \\
\text{alarmSet' = False}
\end{align*}
\]

The \( \text{initTime}? \) value is obtained from the hardware real-time clock via the \( \text{HWtime} \) channel.

\[
\text{Init} \equiv \text{HWtime?initTime} \rightarrow (\text{RTCInit})
\]

The operation of getting the clock’s time value simply outputs \( \text{currentTime} \) on the \( \text{RTCgetTime} \) channel and then outputs a report of \( \text{RTCokay} \).

\[
\text{GetTime} \equiv \text{RTCgetTime!currentTime} \rightarrow \text{RTCreport}!\text{RTCokay} \rightarrow \text{Skip}
\]

The operation to get the precision of clock is similar, outputting \( \text{precision} \) over the \( \text{RTCgetPrecision} \) channel.

The operation of setting a new alarm is described by \( \text{RTCSetAlarm} \), which takes the time of the alarm, \( \text{alarmTime}? \), as input. Since this is the only operation that has an error case, we do not have a separate lifting to robust operations, so we also provide a report! output. The \( \text{alarmTime}? \) must be greater than or equal to the \( \text{currentTime} \), since an alarm cannot be set at a time in the past. The \( \text{currentAlarm'} \) is set to the input \( \text{alarmTime}? \) and \( \text{alarmSet'} \) to \text{True}, since an alarm has been set. This operation does not affect the \( \text{currentTime} \). We output the report! value \( \text{RTCokay} \) because this specifies the successful case of the operation.

\[
\begin{align*}
\text{RTCSetAlarm} \quad \\
\Rightarrow \text{RTCState} \\
\text{alarmTime'? : Time} \\
\text{report! : RTCReport} \\
\text{alarmTime'?} \geq \text{currentTime'} \\
\text{currentAlarm'} = \text{alarmTime'?} \\
\text{alarmSet'} = \text{True} \\
\text{currentTime'} = \text{alarmTime'?} \\
\text{report!} = \text{RTCokay}
\end{align*}
\]

The error case for this operation occurs if the given time is in the past. It is described by the schema \( \text{TimeInPast} \), which has the same components as \( \text{RTCSetAlarm} \) but does not change the state. This case applies if \( \text{alarmTime}? \) is less than \( \text{currentTime} \) and results in a report! of \( \text{RTCtimeInPast} \).

\[
\begin{align*}
\text{TimeInPast} \quad \\
\Leftrightarrow \text{RTCState} \\
\text{alarmTime'? : Time} \\
\text{report! : RTCReport} \\
\text{alarmTime'?} < \text{currentTime'} \\
\text{report!} = \text{RTCtimeInPast}
\end{align*}
\]

The action that specifies the complete behaviour of the operation receives the alarm time via the \( \text{RTCsetAlarm} \) channel, and behaving as the disjunction of \( \text{RTCSetAlarm} \) and \( \text{TimeInPast} \). The report output is communicated via the \( \text{RTCreport} \) channel.

\[
\text{SetAlarm} \equiv \text{var report : RTCReport} \bullet \\
\text{RTCsetAlarm?alarmTime} \rightarrow (\text{RTCSetAlarm} \lor \text{TimeInPast}); \\
\text{RTCreport!report} \rightarrow \text{Skip}
\]
The operation of clearing the alarm is defined using a Circus assignment action to set the alarmSet flag to False in response to a signal on the RTCclearAlarm channel. The currentAlarm value can be left at its previous value, since it is not used when there is no alarm set. This operation ends with a report of RTCokay.

\[
\begin{align*}
\text{ClearAlarm} & \triangleq \text{RTCclearAlarm} \rightarrow \text{alarmSet} := \text{False} ; \text{RTCreport!RTCokay} \rightarrow \text{Skip}
\end{align*}
\]

This concludes the definition of the public services of the real-time clock. The clock must also respond to triggering of alarms and hardware clock interrupts. When an alarm triggers, the clock interrupt is sent to the scheduler via the RTCclockInterrupt channel, as described by the TriggerAlarm action. The current alarm is then cleared by setting alarmSet to False.

\[
\begin{align*}
\text{TriggerAlarm} & \triangleq \text{RTCclockInterrupt} \rightarrow \text{alarmSet} := \text{False}
\end{align*}
\]

Clock tick interrupts, which come periodically from the hardware with a period equal to precision are handled as described by Tick. The interrupts come via the HWinterrupt channel and are required to have the identifier of the clock interrupt (the non-clock interrupts are handled by the scheduler). An if statement is used to check if the currentTime with the precision value added to it is greater than currentAlarm. If it is, then the alarm triggers as described in TriggerAlarm. The currentTime value is then incremented by the clock’s precision. Resolving alarms before updating currentTime is required to ensure the state invariant is maintained.

\[
\begin{align*}
\text{Tick} & \triangleq \text{HWinterrupt?interrupt} : (\text{interrupt} = \text{clockInterrupt}) \rightarrow \text{if} \\
& \quad \text{currentTime} + \text{precision} \geq \text{currentAlarm} \rightarrow \text{TriggerAlarm} \\
& \quad \text{currentTime} + \text{precision} < \text{currentAlarm} \rightarrow \text{Skip} \\
& \quad \text{fi}; \text{currentTime} := \text{currentTime} + \text{precision}
\end{align*}
\]

Any of the actions available to the user may be chosen, and the process loops to allow another action to be taken. The process may handle an incoming clock tick instead of a user action.

\[
\begin{align*}
\text{Loop} & \triangleq \text{SetAlarm} \text{□} \text{ClearAlarm} \text{□} \text{GetTime} \text{□} \text{GetPrecision} \text{□} \text{Tick} ; \text{Loop}
\end{align*}
\]

The main action of the process begins by performing the initialisation and then enters the loop.

\[
\begin{align*}
\text{Init}; \text{Loop} \\
\text{end}
\end{align*}
\]

We have now specified the real-time clock that tracks the current time and any alarm that may be set. Operations are provided to set and clear the alarm. The state of the clock is updated when a clock interrupt signal is received and the clock is checked against the alarm, forwarding the interrupt signal to the scheduler if the alarm time has passed.

### 3.4.4 Complete VM Services Model

Having defined the three processes that model the three components of the VM services, we now compose them in parallel to form the complete model of the VM services.

Certain channels are used to communicate between the different components. The RTCInterface channel set contains the channel used to pass the clock interrupt from the real-time clock to the scheduler.

\[
\text{channelset RTCInterface} \equiv \{ \text{RTCclockInterrupt} \}
\]

We define the VMServices process by composing each of the components of the SCJVM services in parallel, with the Scheduler and RealtimeClock synchronising on RTCInterface, which is then hidden since it is an internal channel.

\[
\begin{align*}
\text{process VMServices} & \triangleq ((\text{MemoryManager} || \text{Scheduler}) \\
& \quad || \text{RTCInterface} || \text{RealtimeClock}) \\
& \quad \text{\backslash RTCInterface}
\end{align*}
\]

So the VMServices process represents a complete model of the SCJVM services.
3.5 Final Considerations

In this chapter, we have presented the services that must be provided by an SCJVM in order to support the core execution environment and the SCJ API. We have divided these services into three areas, the memory manager, the scheduler, and the real-time clock, and detailed the services provided in each area. We have also presented our model of the SCJVM services in the Circus specification language, of which a full version can be found in Appendix A.

Our model is composed of a Circus process for each of the three classes of services we have identified. The memory manager process largely consists of Z data operations on the state of the memory, which are then lifted to Circus actions that can be accessed via channels. The scheduler also consists of a large Z model, but requires more reliance on Circus to specify interaction with interrupts. The real-time clock model is mainly made up of Circus actions with few Z schemas, though it is also a smaller component than the other two due to the small number of services it provides.

Overall, the division of the SCJVM services into the three areas we have chosen appears to give a good separation between the components with little coupling. This is shown in Figure 3.3, where it can be seen that only one channel, RTClockInterrupt, is required for communication between the processes in the model. The use of Circus has allowed us to specify the few necessary points of communication between these processes, and also their relation to hardware interrupts and the core execution environment.

The fact that the requirements of scheduler and memory manager model are largely expressed in Z allows them to be checked using Z proof tools. Indeed, we have already partially subjected the memory manager model to proof using Z/Eves. The proofs we have performed are consistency proofs and proofs that functions are not applied outside their domain. We have performed these proofs for the first two parts of the memory manager model, covering memory blocks and backing stores, and also the global memory manager state. The theorems we have proved about the memory manager, along with their proofs and some additional lemmas about mathematical toolkit objects that we have proved in the course of our work, can be found in Appendix D.

![Figure 3.3: The structure of the SCJVM services model, showing the channels used for communication between the processes in the model](image-url)
Chapter 4

The Core Execution Environment

This chapter describes the core execution environment (CEE) of an SCJVM, which handles execution of an SCJ program. In addition, the CEE of an SCJVM manages the flow of execution dictated by the SCJ programming model, including, for example, Safelet setup and mission execution.

This is the part of our SCJVM model that is handled by our compilation strategy. So, it may take the form of a bytecode interpreter, which is the starting point for the compilation, or C code, which is the output of the compilation. We describe both of these in this chapter (Sections 4.2, 4.3 and 4.4) while the compilation strategy for transforming between them is described in the next chapter. We begin with an overview of the CEE’s structure in the next section. We conclude with some final considerations in Section 4.5.

4.1 Overview

The CEE has three components, two of which depend on whether it is interpreting bytecodes or executing C code. For the CEEs that use a bytecode interpreter, the components are listed below and shown in Figure 4.1:

• the object manager, which manages information about objects created during execution of the bytecode;
• the interpreter itself, which handles execution of bytecode instructions; and
• the launcher, which coordinates the startup of the SCJVM, the execution of missions, and the execution of methods in the interpreter.

The components after compilation to C are similar, but the object manager is replaced with a struct manager, which manages C struct types representing objects, and the interpreter is replaced with the C program itself. The launcher remains unchanged throughout the compilation. It is assumed that it is already in the form of native code that can be called from the C code.

The CEE is combined with the SCJVM services to form the complete SCJVM; this is indicated in Figure 4.1 which shows the same structure described in Figure 3.1 in the previous chapter, but has a focus on the CEE components. The SCJVM services are unaffected by the compilation strategy and can be implemented as a separate library.

Each of the components of the CEE is represented by a single Circus process in our model. These processes interact as shown in Figure 4.2. The overall pattern of the interaction is unaffected by the compilation, that is, the model of the compiled code has the same overall flow of communication, although the components have different names and different channels are used for communication.

The launcher manages the startup procedure for the SCJVM and the execution of missions. This involves communication with the interpreter (or C program) to execute initialisation methods. The interpreter
then communicates back with the launcher when it requires services that are provided by the SCJ infrastructure and API, such as registering a schedulable object with the current mission. Allocation of backing stores for the schedulable objects and entering the corresponding memory areas involves communication with both the object (or struct) manager in the CEE and the memory manager of the SCJVM services. The launcher must also communicate with the scheduler to indicate when threads should be started or suspended during mission execution.

The interpreter must accept the requests to execute methods on the main thread from the launcher, and it must also respond to requests from the scheduler to start the other threads. When a thread has finished execution, the interpreter signals to the scheduler that the thread has finished so that it is no longer scheduled. The interpreter must also communicate with the launcher to handle calls to methods that are provided by the SCJ infrastructure, such as the methods to enter memory areas. Handling of memory allocation during method execution is performed via communication with the object manager, which then communicates with the SCJVM memory manager. Additionally, the interpreter communicates inputs and outputs to some console input/output device, which is the only such device required by the SCJ specification. Supporting a full range of hardware connections is beyond the scope of this work.

The interactions just described are modelled by channel communications. Those with the SCJVM services memory manager and scheduler use the channels already described in Sections 3.4.1 and 3.4.2. The types of values communicated by those channels are also used by the CEE processes. These include the type of object identifiers, ObjectID, the type of thread identifiers, ThreadID, the type of backing store identifiers, BackingStoreID, and the type of virtual machine data words, Word. We also use the ClassID and MethodID types, which are the types of class and method identifiers that are declared in
the scheduler model to permit the declaration of the CEEstartThread channel. Additionally, we declare a field identifier type, FieldID.

[FieldID]

The class, method and field identifiers may be the full names used in Java class files or some shorter representation, such as unique identification numbers. In any case, type information needs to be taken into account so that methods and fields with the same name, but different type signatures, have different identifiers. This is because the identifiers in Java class files include the type information and the correct operation of method overloading relies on it.

The channels used for communication between the CEE processes are summarised in Figure 4.1 with the full channel declarations shown in Appendix B.2. In addition to presenting the name and type for each channel, in the first two columns of the table. We also indicate which components of the CEE make use of the channel. The channels output and input are used for communication with the console device mentioned earlier. As we do not model the console device itself, these are left as externally visible channels when the component processes are composed into the complete SCJVM model. Some channels are marked with various symbols (*, †, + and ‡) so that we can refer to them later in the text.

Most of the channels are part of pairs, with one channel to communicate a signal to begin an operation and supply any inputs, and a return channel to communicate back when the operation has finished and supply any outputs. The return channel is named by appending Ret to the name of the channel used to initiate the operation.

There are some channels that deviate this pattern of having a return channel. The executeMethod channel is used to signal to the interpreter that it should begin execution of a method on a given thread. The interpreter signals on executeMethodRet channel when it has finished execution of the method. Since the launcher may need to take some action, such as exiting a memory area, after the interpreter has finished executing a method, the interpreter waits until it receives a signal on the continueExecution channel before continuing to execute. Since the continueExecution channel forms part of this communication pattern, it does not have its own return channel.

Before the interpreter can execute methods on the main thread, the stack space for the main thread must allocated by the launcher and communicated to the interpreter. This is handled by the initMainThread channel, which carries the StackID for the stack space allocated for the main thread. The interpreter waits for communication on the executeMethod channel before commencing execution, so the launcher does not need to wait for the interpreter to finish registering the main thread’s stack.

As mentioned above, while executing a method, the interpreter may signal back to the launcher for handling of special methods. The channels used for this are the ones marked with a * or a + in Table 4.1. The channels marked with a * represent calls to infrastructure methods that are part of the SCJ API. The inputs and outputs of these methods (and hence the types of the channels associated with them) are taken from the SCJ specification. The channels marked with a † are methods that do not return a value and involve execution of a method in the interpreter as part of their handling. Thus, the interpreter waits for a signal on the executeMethod channel after signalling the launcher to handle one of these methods. The methods marked with a † do not, therefore, require separate return channels. The channels marked with a + expose SCJVM scheduler operations to the code executed in the interpreter, in order to allow for the implementation of event handlers. Their types follow those of the scheduler’s channels.

As mentioned previously, the output and input channels are used to communicate Word values to and from a console device. The rest of the channels are used by the launcher and the interpreter to communicate with the object manager. The enterBackingStore channel is used by the launcher to signal to the object manager when a memory area is entered so that it can record that the corresponding backing store has been entered. This carries the ThreadID of the thread to be entered, since the backing stores entered are recorded separately for each thread, and the BackingStoreID of the backing store to be entered. There is no corresponding return channel, since it is not necessary for the launcher to wait while the object manager records the entry to a backing store. Similarly, the exitBackingStore channel is used to signal an exit from the backing store that is the current allocation context of the given thread. This
<table>
<thead>
<tr>
<th>Name</th>
<th>Parameter Type</th>
<th>Communication from</th>
<th>Communication to</th>
</tr>
</thead>
<tbody>
<tr>
<td>executeMethod</td>
<td>ThreadID × ClassID × MethodID × seq Word</td>
<td>L</td>
<td>I</td>
</tr>
<tr>
<td>executeMethodRet</td>
<td>ThreadID × Word</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>continueExecution</td>
<td>ThreadID</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>initMainThread</td>
<td>StackID</td>
<td>L</td>
<td>I</td>
</tr>
<tr>
<td>*† register</td>
<td>ThreadID × ObjectID</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>*† enterPrivateMemory</td>
<td>ThreadID × N × ObjectID</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>*† executeInAreaOf</td>
<td>ThreadID × ObjectID × ObjectID</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>*† executeInOuterArea</td>
<td>ThreadID × ObjectID</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>† enterPerReleaseMemory</td>
<td>ThreadID × ObjectID</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>+ suspend</td>
<td>&lt;no parameters&gt;</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>+ suspendRet</td>
<td>&lt;no parameters&gt;</td>
<td>L</td>
<td>I</td>
</tr>
<tr>
<td>+ resumeThread</td>
<td>ThreadID</td>
<td>I</td>
<td>L</td>
</tr>
<tr>
<td>+ resumeThreadRet</td>
<td>&lt;no parameters&gt;</td>
<td>L</td>
<td>I</td>
</tr>
<tr>
<td>output</td>
<td>Word</td>
<td>I</td>
<td>&lt;ext.&gt;</td>
</tr>
<tr>
<td>input</td>
<td>Word</td>
<td>&lt;ext.&gt;</td>
<td>I</td>
</tr>
<tr>
<td>enterBackingStore</td>
<td>ThreadID × BackingStoreID</td>
<td>L</td>
<td>OM</td>
</tr>
<tr>
<td>exitBackingStore</td>
<td>ThreadID</td>
<td>L</td>
<td>OM</td>
</tr>
<tr>
<td>exitBackingStoreRet</td>
<td>BackingStoreID × B</td>
<td>L</td>
<td>OM</td>
</tr>
<tr>
<td>getCurrentAC</td>
<td>ThreadID</td>
<td>L</td>
<td>OM</td>
</tr>
<tr>
<td>getCurrentACRet</td>
<td>BackingStoreID</td>
<td>OM</td>
<td>L</td>
</tr>
<tr>
<td>newObject</td>
<td>ThreadID × ClassID</td>
<td>I/L</td>
<td>OM</td>
</tr>
<tr>
<td>newObjectRet</td>
<td>ObjectID</td>
<td>OM</td>
<td>I/L</td>
</tr>
<tr>
<td>† getClassIDOF</td>
<td>ObjectID × ClassID</td>
<td>I/L</td>
<td>OM</td>
</tr>
<tr>
<td>† getField</td>
<td>ObjectID × FieldID</td>
<td>I</td>
<td>OM</td>
</tr>
<tr>
<td>† getFieldRet</td>
<td>Word</td>
<td>OM</td>
<td>I</td>
</tr>
<tr>
<td>† putField</td>
<td>ObjectID × FieldID × Word</td>
<td>I</td>
<td>OM</td>
</tr>
<tr>
<td>† putStatic</td>
<td>ClassID × FieldID</td>
<td>I</td>
<td>OM</td>
</tr>
<tr>
<td>† putStaticRet</td>
<td>Word</td>
<td>OM</td>
<td>I</td>
</tr>
<tr>
<td>† addThreadMemory</td>
<td>ThreadID × BackingStoreID</td>
<td>I</td>
<td>OM</td>
</tr>
<tr>
<td>removeThreadMemory</td>
<td>ThreadID</td>
<td>I</td>
<td>OM</td>
</tr>
</tbody>
</table>

Table 4.1: The channels used for communication between CEE processes before compilation. In the final two columns, L refers to the launcher, I refers to the interpreter, OM refers to the object manager, I/L indicates a channel shared by the interpreter and launcher in interleaving, and <ext.> indicates an external channel.

does have a return channel, since the launcher must be informed if the backing store was cleared due to no longer being in use by any thread. The BackingStoreID of the exited backing store and a boolean value indicating if the backing store was cleared are therefore communicated back to the launcher on a return channel. Additionally, the getCurrentAC channel (and its return channel) is used to obtain the BackingStoreID of the backing store used as the current allocation context for a given thread from the object manager, in order to handle some cases of entering memory areas.

The remaining channels used by the launcher to communicate with the object manager are used by both the launcher and the interpreter. These are the newObject channel, which is used to allocate space for new objects in the current allocation context, and the getClassIDOF channel, which is used to obtain the ClassID for the class of the object associated with a given ObjectID. The newObject channel carries the ThreadID of the current thread, since there is a separate allocation context for each thread, and the ClassID of the class of the object to be allocated. The object manager returns the ObjectID of the newly allocated object via the corresponding return channel. The getClassIDOF channel carries both the input and output to the operation on the same channel, since it is a simple data accessing operation that can be dealt with in a single communication.
The other channels used by the interpreter are the channels for accessing fields of objects and classes. The getField channel is used for obtaining the value stored in a given field of a given object. It carries the ObjectID of the object whose field is to be accessed and the FieldID of the field to be accessed. The object manager then returns the Word value stored in the field. For putting a value into an object’s field, the putField channel is used, which carries the Word value to store in the field in addition to the ObjectID and FieldID that identify the object and field to update. As this just updates the field and does not return any information, there is no need for a return channel. Channels for accessing static fields, getStatic and putStatic, are also provided. These operate similarly to the channels for object fields but use ClassID values rather than ObjectID values, since static fields are attached to classes rather than objects.

The final channels used by the interpreter are the addThreadMemory and removeThreadMemory channels. The addThreadMemory is used to inform the object manager of a thread’s initial allocation context when the thread starts. It carries the ThreadID of the thread and the BackingStoreID of the backing store that serves as the thread’s initial allocation context. When a thread has finished execution, it informs the object manager via the removeThreadMemory channel, which carries the ThreadID of the thread.

As mentioned earlier, some channels used by the interpreter to communicate with the object manager are replaced with different channels during compilation. Those channels are marked with a ‡ in Table 4.1. After compilation these channels are replaced with channels to obtain the struct representing the contents of an object and to store an object’s struct after updating it. Note that the getClassIDOf channel is shared between the launcher and interpreter. After compilation, the interpreter accesses a struct field storing the ClassID for an object. However, the launcher is unaffected by the compilation and is agnostic as to whether the program is in the form of bytecode or C code. Therefore, the launcher continues to use the getClassIDOf channel after compilation, which represents a service offered by the object manager or struct manager to obtain the ClassID by whatever means are appropriate to the form of the object. As an optimisation in an implementation, the launcher could be changed to access struct fields in the same way as the interpreter. We discuss the form of field accesses in the C code and the channels used for them in more detail in Sections 4.4.1 and 4.4.2.

Next, in Section 4.2 we describe our model of the launcher. We then detail the bytecode interpreter model in Section 4.3 and the C code model in Section 4.4.

### 4.2 Launcher

As mentioned in the previous section, the launcher is the component of the CEE that manages the SCJVM startup and coordinates mission execution. It is described by the Launcher process.

The launcher remains unaffected throughout the compilation strategy, because it is agnostic to the class and bytecode information. However, the launcher must know where to begin execution, so it takes a parameter, safeletClass, which is the ClassID of the Safelet class. This can be seen in the the Launcher process definition, the beginning of which is shown below.

```plaintext
process Launcher = safeletClass : ClassID; initOrder : seq ClassID • begin
```

In what follows, we describe the definition of Launcher, focusing on the aspects relevant for the compilation. The complete definition can be found in Appendix B.6.

The state of the Launcher is divided into three parts. The first part contains the identifiers of the objects that form the SCJ mission model, so that the Launcher can call methods of those objects during SCJVM startup. The second part contains information on the memory-area objects of the program, including the relationship between the memory-areas and the backing stores they represent, so that methods for
entering and exiting memory-areas can be handled. The final part of the state describes the relationship between the schedulable objects of SCJ and the threads used by the CEE so that the threads can be started when mission execution begins.

We use separate Z schemas to specify each part of the state. The first part is described by the MissionManager schema, shown below. It contains the identifiers of three objects:

- `safelet`, the instance of the class implementing the Safelet interface for the program;
- `missionSequencer`, the mission sequencer returned by the safelet’s `getSequencer()` method; and
- `currentMission`, the mission that is currently executing.

Methods of these objects are called at various points throughout SCJVM startup and mission execution.

```
MissionManager
safelet, missionSequencer, currentMission : ObjectID
```

The second part of the Launcher’s state is described by the MemoryAreaManager schema below. It contains the identifiers of the memory-area objects for the immortal memory, `immortalMemory`, and mission memory, `missionMemory`. There is a map, `backingStores`, that relates these identifiers and the identifiers of the other memory-area objects, to the identifiers of the backing stores they represent. We also record the backing store identifiers of the per-release memories for each thread in the `perReleaseMemories` map. Finally, to make sure that nested private memories can be reused, there is a map from backing store identifiers to the identifiers of private backing stores they contain, `privateMemoryMap`.

```
MemoryAreaManager
immortalMemory, missionMemory : ObjectID
backingStores : ObjectID => BackingStoreID
perReleaseMemories : ThreadID => BackingStoreID
privateMemoryMap : BackingStoreID => BackingStoreID
```

Each of the maps in the MemoryAreaManager is injective, since each memory-area object has a distinct backing store and memory-areas cannot share a nested private memory-area. The invariants of MemoryAreaManager are elided above. They ensure that each memory-area object has a corresponding backing store in backingStores, and that areas which are not nested private memories do not appear in the range of privateMemoryMap.

The final part of the state is specified in the SchedulableManager schema below. It contains a map, `schedulableThreads`, from the identifiers of schedulable objects to the identifiers of the threads associated with them. This map must be injective, since every schedulable object has a separate thread.

```
SchedulableManager
schedulableThreads : ObjectID => ThreadID
```

The state of the process is then the conjunction of these three schemas.

```
state LauncherState == MissionManager \& MemoryAreaManager \& SchedulableManager
```

The Launcher state is initialised as described in LauncherInit, which is shown below. The object identifiers are initialised to the null identifier. They are later filled with non-null identifiers as the corresponding objects are created during SCJVM execution. Similarly, each of the maps is initialised to the empty set.
The main action of the Interpreter proceeds as shown below. The state is first initialised as described by LauncherInit and then the actions Startup and RunNextMission follow in sequence. Startup defines the SCJVM startup procedure that must be performed once at the start of SCJVM execution, whereas RunNextMission defines the procedure that must be performed for each mission run. We do not handle mission termination in our Launcher model. This is because the SCJ mission termination procedure has almost no effect on our compilation strategy; a single mission is sufficient for our examples to evaluate the compilation strategy. A formal account of it is available elsewhere [21, 61, 101]. Thus, RunNextMission is only executed once.

- \((\text{LauncherInit}) \; ; \; \text{Startup} \; ; \; \text{RunNextMission}\)

The definition of Startup is shown below. It performs a number of actions in sequence, following the startup procedure for an SCJVM:

- creating the main thread’s stack and passing on the \textit{initMainThread} channel, in MakeMainStack;
- executing the class initialisers in the order given in \textit{initOrder}, in RunClassInitialisers;
- creating the immortal memory object that corresponds to the root backing store and storing it in \textit{immortalMemory}, in CreateImmortalMemory;
- creating the \textit{Safelet} object and storing it in \textit{safelet}, in CreateSafelet;
- calling the \textit{immortalMemorySize()} and \textit{globalBackingStoreSize()} methods of the \textit{safelet}, and checking that the size of the root backing store matches those values, in CheckImmortalMemory and CheckRemainingBackingStore;
- calling the \textit{initializeApplication()} method of the \textit{safelet}, in InitializeApplication;
- calling the \textit{safelet’s getSequencer()} method and storing the returned value in \textit{missionSequencer}, in GetSequencer; and
- creating the \textit{missionMemory} object with its corresponding backing store, in CreateMissionMemory.

\[
\text{Startup} \equiv \text{MakeMainStack} \; ; \; \text{RunClassInitialisers} \; ; \; \text{CreateImmortalMemory} \; ; \; \text{CreateSafelet} \; ; \\
\text{CheckImmortalMemory} \; ; \; \text{CheckRemainingBackingStore} \; ; \; \text{InitializeApplication} \; ; \\
\text{GetSequencer} \; ; \; \text{CreateMissionMemory}
\]

RunNextMission begins with calling the \textit{getNextMission()} method of \textit{missionSequencer}, in the action GetNextMission. The returned mission is stored in \textit{currentMission}. Its \textit{missionMemorySize()} method is then executed, and the backing store of \textit{missionMemory} is resized to match, in ResizeMissionMemory. Next, in InitializeMission, the mission’s \textit{initialize()} method is executed, during which the schedulable objects for the mission are registered. Afterwards, in InitialiseAndStartThreads, the registered schedulable objects have their stacks and backing stores created, after which the threads for all the schedulable objects are started. Finally, in WaitForExecution, the main thread suspends itself and the Launcher then waits, handling special methods for the threads of the program when necessary. Since termination is not handled, this phase of the program continues indefinitely.

\[
\text{RunNextMission} \equiv \text{GetNextMission} \; ; \; \text{ResizeMissionMemory} \; ; \; \text{InitializeMission} \; ; \\
\text{ InitialiseAndStartThreads} \; ; \; \text{WaitForExecution}
\]
During these actions, methods are executed using the channels `executeMethod`, `executeMethodRet` and `continueExecution`, discussed earlier. The identifiers of the methods, which may be standard methods from the SCJ API, or implementation-defined API methods required by the launcher, are represented by constants in the model. Although most of the methods used by the `Launcher` are executed simply by communicating on each of the channels mentioned above in turn, as noted above, in `InitializeMission`, the `initialize()` method of a mission requires handling of the `register()` method for each schedulable object. We must also provide handling for other special methods. This is done in the `HandleSpecialMethodsMainLoop` action below, which offers handling of the special methods while waiting for return from the `initialize()` method on the `executeMethodRet` channel. A similar action, without the final choice accepting `executeMethodRet`, is used to handle special methods in `WaitForExecution`.

```
HandleSpecialMethodsMainLoop ≜ val memoryEntries : ThreadID → N; res retVal : Word •

□ t : ThreadID •
  EnterMemory(t);
  HandleSpecialMethodsMainLoop(memoryEntries ≡ {t→ memoryEntries t + 1}, retVal))

□ t : ThreadID •
  (memoryEntries t > 0) & ExitMemory(t);
  HandleSpecialMethodsMainLoop(memoryEntries ≡ {t→ memoryEntries t − 1}, retVal))

((Register □ Suspend □ Resume);
  HandleSpecialMethodsMainLoop(memoryEntries, retVal))

((∀ t : ThreadID • memoryEntries t = 0) &
  executeMethodRet?thr : (thr = main)?r → continueExecution!main → retVal := r)
```

`HandleSpecialMethodsMainLoop` takes a value parameter, `memoryEntries`, which is a map recording how many times a memory area has been entered for each thread. It also take a value parameter, `retVal`, which captures the return value from the execution of the method on the `main` thread. It offers a choice of handling a memory-area entry, handling the corresponding memory-area exit, handling a special method that does not enter memory-areas, or accepting return from the execution of the method on the `main` thread (handled in the usual way using `executeMethodRet` and `continueExecution`, with the return value stored in `retVal`).

`HandleSpecialMethodsMainLoop` handles memory-area entering methods, which execute another method after entering a memory-area, during which further special methods may be called. Each entry to a memory-area must be matched by a corresponding exit from the memory-area when this extra method execution returns. Thus, the entries to memory-areas are tracked in the `memoryEntries` map.

The number stored in `memoryEntries` for a thread identifier `t` is incremented after handling a memory-area entry on that thread as described in `EnterMemory(t)`. Similarly, it is decremented after handling exit from the memory-area in `ExitMemory(t)`, which is only offered if the value is already greater than zero. After handling memory-area entry or exit, or another special method (handled in the actions `Register`, `Suspend` and `Resume`), `HandleSpecialMethodsMainLoop` recurses to allow further special methods to be handled. The return from the top-level method execution on the `main` thread is only permitted once all memory areas have been exited and `memoryEntries` is zero for all threads.

To illustrate how the nested method execution after memory-area entry is performed, we show the `ExecuteInAreaOf` action below, which is one of the actions offered in external choice in `EnterMemory`, along with actions to handle other memory-area entering operations. `ExecuteInAreaOf` takes a thread identifier `thread` as a parameter and only accepts communications from that thread, so that we can separate out memory-area entries for each thread. The identifier of a `Runnable` object, `runnable`, is received via the `executeInAreaOf` channel, which is the object that indicates the method to be executed in the memory-area. Such an identifier is received for all of the memory-area entering methods. In the case of `ExecuteInAreaOf`, another identifier, `object`, is also received and a `FindBackingStore` action is used to communicate with the memory manager to determine its backing store. This backing store is then entered, via communication on the `enterBackingStore` channel. The class of the `runnable` object is
determined and its \( run() \) method (represented here by the \( \text{run} \) identifier) is executed by signalling on the \( \text{executeMethod} \) channel. No communication on the \( \text{executeMethodRet} \) channel is waited for in this action, because it is handled separately in the \( \text{ExitMemory} \) action since other special methods (including further memory entries) may occur inbetween.

\[
\text{ExecuteInAreaOf} \triangleq \begin{align*}
\text{val} & \quad \text{thread} : \text{ThreadID} \odot \text{var} \quad \text{runnable}, \text{object} : \text{ObjectID}; \text{bs} : \text{BackingStoreID} \odot \\
\text{executeInAreaOf} & \odot t : (t = \text{thread})?\text{obj}?r \rightarrow \text{object}, \text{runnable} := \text{obj}, r; \\
\text{FindBackingStore} & \odot (\text{object, bs}); \\
\text{enterBackingStore} & \odot \text{thread, bs} \rightarrow \text{getClassIDOf}\text{runnable}\text{runnableClass} \\
& \rightarrow \text{executeMethod}\text{thread!runnableClass!run!(runnable)} \rightarrow \text{Skip}
\end{align*}
\]

The return from the \( \text{run()} \) method, and the exit from the memory-area, is specified by the \( \text{ExitMemory} \) action below. This, as with the \( \text{ExecuteInAreaOf} \) action, takes a \( \text{thread} \) parameter. A return from a method executing on that thread is accepted on the \( \text{executeMethod}\text{run()} \) channel, and its return value is discarded as the \( \text{run()} \) method is \( \text{void} \). The exit from the memory-area is then performed using the \( \text{exitBackingStore} \) and \( \text{exitBackingStoreRet} \) channels. The \( \text{Launcher} \) state may afterwards be updated to account for the exited memory-area being cleared (due to no longer being in use by any thread), which is specified in the \( \text{ClearPrivateMemory} \) schema. After the exit from the memory-area has been handled, the \( \text{Launcher} \) signals that normal execution on \( \text{thread} \) may continue using the \( \text{continueExecution}\text{channel} \).

\[
\text{ExitMemory} \triangleq \text{val} \quad \text{thread} : \text{ThreadID} \odot \\
\text{executeMethodRet} \odot t : (t = \text{thread})?\text{void} \\
& \rightarrow \text{exitBackingStore!thread} \rightarrow \text{exitBackingStoreRet?bsid?isCleared} \rightarrow \\
\text{if} \quad \text{isCleared} = \text{True} \rightarrow \left(\text{ClearPrivateMemory}[\text{bsid}/\text{toClear}]\right) \\
\text{[isCleared} = \text{False} \rightarrow \text{Skip} \\
\text{fi} \quad \text{continueExecution}\text{!thread} \rightarrow \text{Skip}
\]

The \( \text{register()} \) method also needs to execute a method in the interpreter to obtain a thread’s priority. Further special methods are not handled in this case, since the method to obtain a thread’s priority is expected to be a simple method that does not call special methods.

This handling of special methods is used by the interpreter process (or \( \text{C program, after compilation} \)), which must communicate with the \( \text{Launcher} \) when such methods are encountered. To allow for the behaviour of executing methods after entering memory areas (or during the \( \text{register()} \) method), the interpreter must also be prepared to execute another method after signalling to the \( \text{Launcher} \). This is transformed to nested method executions during the compilation strategy in order to preserve the communication pattern with the \( \text{Launcher} \). We describe in more detail how this communication is performed in the interpreter in Section 4.3.4 and in \( \text{C program} \) in Section 4.4.2. In the next section, we describe the bytecode interpreter, which, along with the \( \text{Launcher} \), forms the \( \text{CEE} \) before the application of the compilation strategy.

### 4.3 Bytecode Interpreter Model

This section describes the bytecode interpreter that handles execution of an \( \text{SCJ bytecode program} \). Its model is composed of two processes: the model of the object manager, \( \text{ObjMan} \), and the model of the interpreter itself, \( \text{Interpreter} \). These are composed together in parallel with the \( \text{Launcher} \) to form the complete core execution environment, \( \text{CEE} \), as shown below. The synchronisation sets and channel hidings, omitted here, are consistent with the communication patterns shown in Table 4.1. The parameters of each of the components, including the classes, \( cs \), and bytecode instructions, \( bc \), which are explained later in this section, become parameters of \( \text{CEE} \).

\[
\text{CEE}(cs, bc, \text{sid, initOrder}) \triangleq \text{ObjMan}(cs) \parallel \text{Interpreter}(cs, bc) \parallel \text{Launcher}(\text{sid, initOrder})
\]

In Section 4.3.1 we first give an informal description of the bytecode instructions handled in our model and the ways in which their \( \text{SCJ semantics} \) differ from that of standard Java. In Section 4.3.2 we describe
our model of Java class information that is used by both *ObjMan* and *Interpreter*. The first component, *ObjMan*, is described in Section 4.3.3 and the second, *Interpreter*, in Section 4.3.4.

4.3.1 Bytecode Subset

We model a subset of Java bytecode sufficient to express a wide variety of SCJ programs and illustrate how further features may be added, but small enough to permit effective reasoning. The subset has been chosen by considering the bytecode generated from a simple SCJ program and removing instructions similar to those already in the subset. This ensures the model is not unnecessarily complicated with trivial or redundant instructions, so we can concentrate on the instructions that are most of interest in creating the compilation strategy. The bytecode instructions in our subset are described in Table 4.2.

Java bytecode instructions operate over a state that records information on all loaded classes, a stack frame, and the object data residing in memory. Various pieces of class information are required for execution of bytecode instructions, but a constant pool, which stores all the constants and names required by the class, is the main information used.

The constant pool contains references to classes, methods and fields used by the bytecode instructions in the class, as well as constant values used in the code. The form of the constant pool is a large array. Indices into this array are used as parameters to instructions requiring information from the constant pool.

For example, the *getfield* and *putfield* instructions take constant pool index parameters pointing to a reference to a field whose value should be obtained or set. Other class information used at runtime includes information on fields and methods belonging to the class, which is required for creation of objects and invocation of methods.

The frame stack forms the second part of the JVM manipulated by bytecode instructions and consists of a series of frames that contain the runtime information for each invocation of a method. When a method is invoked, a new stack frame is created for it and pushed onto the frame stack, and when the method returns, the stack frame is popped from the stack.

Each stack frame contains an operand stack, which is used to store values manipulated by bytecode instructions, and an array of local variables. Most bytecode instructions manipulate the operand stack in some way, popping arguments from it, pushing results to it or performing specific operations upon it.

The local variables are used to store the arguments of a method and the results of computations performed on the operand stack. Operations are not performed directly on the local variables, so the only bytecode instructions that affect them are those for moving values between the operand stack and the local variables (*aload* and *astore* are examples of such instructions).

Some bytecode instructions also manipulate objects, which in our case reside in backing store memory. Such instructions include *new*, which creates objects, and *getfield*, which gets the value from a field of an object. In our choice of instructions for the subset, we mainly focus on manipulation of objects and method invocation, since those are core concepts of Java bytecode and require special handling by the compilation strategy.

The instruction *dup* is included as an example of a simple instruction that operates on the operand stack. It has been chosen for its frequent occurrence in object initialisation. Other instructions that do simple operand stack manipulation, including the arithmetic instructions, can be specified similarly.

We also include a few arithmetic instructions as an example of how integers are handled. Specifically, we include the integer addition operation, *iadd*, as an example of a binary operation, and the integer negation operation, *ineg*, as an example of an unary operation. We do not include operations for floating point values since the operations upon them are not substantially different from those on integers at the level of modelling and compilation. The model can be easily extended to include more integer operations.

Instructions that create object references (the *new* and *aconst_null* instructions), pass them around (*aload, astore, areturn*, etc.), and permit field accesses (*getfield* and *putfield*) are also included to allow the full range of object manipulations. We also provide instructions for *static* field accesses (*getstatic* and *putstatic*) since they are of use in sharing data between different parts of the program.
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>aconst_null</td>
<td>(none)</td>
<td>Pushes a null object reference onto the operand stack.</td>
</tr>
<tr>
<td>aload</td>
<td>local variable index</td>
<td>Loads the value from a specified local variable and pushes it onto the operand stack.</td>
</tr>
<tr>
<td>areturn</td>
<td>(none)</td>
<td>Returns from the current method, pushing the value on top of the current method's operand stack onto the operand stack of the method returned to.</td>
</tr>
<tr>
<td>astore</td>
<td>local variable index</td>
<td>Pops a value from the operand stack and stores it in the specified local variable.</td>
</tr>
<tr>
<td>dup</td>
<td>(none)</td>
<td>Duplicates the value on top of the operand stack.</td>
</tr>
<tr>
<td>getfield</td>
<td>constant pool index</td>
<td>Pops an object reference from the operand stack, gets the value of the field specified by the identifier at the given constant pool index for the referenced object, and pushes it onto the operand stack.</td>
</tr>
<tr>
<td>getstatic</td>
<td>constant pool index</td>
<td>Gets the value of the static field specified by the field and class identifiers at the given constant pool index, and pushes it onto the operand stack.</td>
</tr>
<tr>
<td>goto</td>
<td>program address</td>
<td>Unconditionally branches to the given program address.</td>
</tr>
<tr>
<td>iadd</td>
<td>(none)</td>
<td>Pops two integer values from the operand stack, adds them, and pushes the result onto the operand stack.</td>
</tr>
<tr>
<td>iconst</td>
<td>integer value</td>
<td>Pushes the given integer value onto the operand stack of the current method.</td>
</tr>
<tr>
<td>if_icmple</td>
<td>program address</td>
<td>Pops two integer values from the operand stack, and branches to the given program address if the second value popped is less than or equal to the first value.</td>
</tr>
<tr>
<td>ineg</td>
<td>(none)</td>
<td>Pops an integer value from the operand stack, negates it, and pushes the negated value onto the operand stack.</td>
</tr>
<tr>
<td>invokespecial</td>
<td>constant pool index</td>
<td>Gets the method and class identifier at the given constant pool index and invokes the specified method of the specified class, popping the method's arguments, plus a this object reference, from the operand stack.</td>
</tr>
<tr>
<td>invokestatic</td>
<td>constant pool index</td>
<td>Gets the method and class identifier at the given constant pool index and invokes the specified method of the specified class, popping the method's arguments from the operand stack.</td>
</tr>
<tr>
<td>invokevirtual</td>
<td>constant pool index</td>
<td>Gets the method and class identifier at the given constant pool index, pops the arguments of the specified method, including a this object reference, from the operand stack, and invokes the specified method of the class of referenced object.</td>
</tr>
<tr>
<td>new</td>
<td>constant pool index</td>
<td>Allocates a new object of the class specified by the identifier at the given constant pool index and pushes a reference to the new object onto the operand stack.</td>
</tr>
<tr>
<td>putfield</td>
<td>constant pool index</td>
<td>Pops an object reference and value from the operand stack and stores the value in the field specified by the identifier at the given constant pool index for the referenced object.</td>
</tr>
<tr>
<td>putstatic</td>
<td>constant pool index</td>
<td>Pops a value from the operand stack and stores the value in the static field specified by the field and class identifiers at the given constant pool index.</td>
</tr>
<tr>
<td>return</td>
<td>(none)</td>
<td>Returns from a method with no return value.</td>
</tr>
</tbody>
</table>

Table 4.2: The instructions in our bytecode subset
However, arrays are not included as they require additional instructions and can be emulated, albeit inefficiently, with the instructions given here.

Both the `invokevirtual` and `invokespecial` instructions, which invoke methods on objects, are included. The `invokevirtual` instruction looks up the method to invoke in the method table for the class of the object that the method is invoked on. The `invokespecial` instruction, on the other hand, uses the class identifier supplied in the method reference pointed to by the parameter of the instruction when looking up the method. The `invokestatic` instruction, for invoking static methods of classes, is similar to `invokespecial`, but does not supply a `this` object parameter, whereas `invokevirtual` and `invokespecial` pop this from the stack as an extra argument.

The `goto` and `if_icmple` instructions are provided as examples of control flow instructions, with `goto` representing an unconditional branch and `if_icmple` representing a conditional branch. Other forms of conditional branch may be implemented in a similar fashion to `if_icmple`, but we do not include those in our subset since `if_icmple` is sufficient to represent most control flow structures. Although `goto` could be represented as a special case of `if_icmple`, we include it as a separate instruction due to its frequent use in conjunction with `if_icmple` to implement loops.

We do not handle exceptions; errors in the SCJVM are instead handled by simply aborting execution. SCJ programs can be statically verified to prove that exceptions will not be thrown [45, 63]. Furthermore, reliance on exceptions to handle errors has been discouraged by an empirical study due to the potential for errors in exception handling [90]. The bytecode instructions that relate to throwing and catching exceptions are, therefore, not included in our bytecode subset.

As a simplifying assumption, we consider that all values consist of only a single virtual machine word. This means that `long` and `double` values are not handled. Any SCJ API methods that take `long` or `double` arguments are viewed as taking `int` or `float` instead. The reason for this assumption is that handling of two word values makes little difference at the level of the formal model and our approach can be easily extended to deal with more types.

Further, we do not make a distinction between the different virtual machine types in our bytecode instructions. This is justified as the bytecode instructions simply handle values as 32-bit words, with the type information only used for typechecking during bytecode validation. The code passed into the core execution environment is assumed to have already passed, which may be done by a separate component [24, 46, 51, 95]. Since many of the instructions behave the same for different types, we only include those instructions that handle values as object references. We would introduce a lot of duplication in the model if, for example, both the `areturn` and `ireturn` instructions were to be included.

Because we are considering bytecode arising from an SCJ program, some requirements of SCJ permit further simplifications to our bytecode subset. The `invokedynamic` instruction performs method invocation with runtime typechecking, mainly for the purpose of implementing dynamically-typed languages targeting the JVM (though it is also used to implement the lambda expressions introduced in Java 8). It is not included in our subset as it does not allow static typechecking and so should not be used for SCJ.

The requirement for all classes to be loaded at startup greatly simplifies the semantics of several instructions, since dynamic class loading does not need to be considered. It also means that method lookup tables can be precomputed. This means that the semantics of the `invokevirtual` and `invokeinterface` instructions are the same, since they both invoke a method on an object, using the object’s class as the class for method lookup. They, therefore, do not both need to be included and so we have not included the `invokeinterface` instruction, since it exists only to optimise method lookup.

In terms of concurrency considerations, we are assuming our SCJVM to be single processor, and so we do not need to have more than one interpreter. As we see later, the interpreter’s threads are modelled using separate Circus processes, but execution only occurs on one at a time. We also assume that thread switches can only occur between bytecode instructions in the interpreter. This is justified since bytecode instructions should appear to be atomic. An implementation may be non-atomic as long as the externally visible sequence of events is the same as for the model with atomic instructions. This means that instructions requiring communication with other components of the SCJVM, such as `new`, which communicates with the memory manager, must be atomic since they affect shared state.
Having described our bytecode subset and the assumptions we are making, we now proceed to describe our model of Java classes in the next section.

4.3.2 Classes

In our model, information about the Java classes that form the program is recorded in a map, cs, that is provided as a parameter to CEE. The cs map associates ClassIDs with records of a schema type Class defined as the conjunction of three schemas. The first schema, ClassConstantPool contains components that represent the constant pool and indices into the constant pool. The second schema, ClassMethods, represents information on the methods in the class. The final schema, ClassFields, is our model for information on the fields in the class.

The components of ClassConstantPool are constantPool, the constant pool itself, and some indices into constantPool: this, referencing the current class, super, referencing the current class’ superclass, and interfaces, a set of indices referencing the interfaces implemented by the current class.

The entries of constantPool are indexed by a elements of a type CPIndex. In the JVM, the CPIndex values are positive integers, but no arithmetic or comparison is performed on constant pool indices in our model, so we do not represent that fact.

We distinguish one particular CPIndex value, a constant nullCPIndex, which represents an invalid index into constantPool. It is used as a placeholder in cases when no index is present. For example, the class Object has no superclass, so the index of the constant pool entry referencing its superclass is nullCPIndex.

Each of the entries in the constantPool is represented by an element of a free type CPEntry, the definition of which is shown below. It has three constructors: ClassRef, representing a reference to a ClassID, MethodRef, representing a reference to a method of a particular class by a ClassID and MethodID, and FieldRef, representing a reference to a field of a particular class by a ClassID and FieldID.

The first conjunct of the invariant of ClassConstantPool requires that nullCPIndex not be in the domain of constantPool, since nullCPIndex is not a valid index into constantPool. The second conjunct states that the indices this, super, and interfaces must be in the domain of constantPool, unless super is nullCPIndex (which is the case for the Object class). Finally, the third conjunct requires that the constantPool entries at this, super and interfaces are ClassRefs.

The components of ClassMethods, shown below, are maps from MethodID values to information about each method. The first two, methodEntry and methodEnd, map to ProgramAddress values, which are indices into a separate bytecode array representing the start and end points of the method. The last two
components, methodLocals and methodStackSize, map to natural numbers giving the required number of local variables and operand stack slots for the method. These values are used during the compilation strategy to declare C variables to store the local variables and operand stack values.

<table>
<thead>
<tr>
<th>ClassMethods</th>
</tr>
</thead>
<tbody>
<tr>
<td>methodEntry, methodEnd : MethodID ↦ ProgramAddress</td>
</tr>
<tr>
<td>methodLocals, methodStackSize : MethodID ↦ ℕ</td>
</tr>
</tbody>
</table>

∀ m : dom methodEntry = dom methodEnd = dom methodLocals = dom methodStackSize
∀ m : dom methodLocals • methodArguments m ≤ methodLocals m

In addition to the components of ClassMethods, we declare a global function methodArguments from MethodIDs to natural numbers, which gives the number of arguments that each method takes. This is a global function since each MethodID encodes the type of the method, so the number of arguments for a method can always be determined from its identifier. The methodArguments function is also total for this reason. We use methodArguments in the invariant of ClassMethods, and also in the Interpreter and compilation strategy when handling method calls.

The first conjunct of the invariant of ClassMethods requires that all the component maps have the same domain, so that all the information must be supplied for any method present in the class. The second conjunct requires that the methodEntry for each method be before its methodEnd. Finally, the third conjunct requires that methodLocals be large enough for each method to contain its methodArguments, since each argument of a method is stored in a local variable.

The final components of our model for class information are given in the ClassFields schema below. It contains two sets of FieldIDs, fields and staticFields, which are the identifiers of the class’ object fields and static fields respectively. The static and non-static fields need to be distinguished so that we know where each needs to be stored: static variables have only one copy for each class, whereas non-static fields are stored separately for each instance of a class. The fields and staticFields sets are required to be disjoint since no field can be both static and non-static.

<table>
<thead>
<tr>
<th>ClassFields</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields, staticFields : ℕ FieldID</td>
</tr>
<tr>
<td>fields ∩ staticFields = ⊘</td>
</tr>
</tbody>
</table>

The three schemas containing the different parts of the class information are conjoined together to form Class, as shown below.

Class == ClassConstantPool ∧ ClassMethods ∧ ClassFields

In addition to defining Class, we also define functions for extracting information from the constantPool for a given Class, in order to make specifying things about them easier. Since the functions are just abbreviations of data access operations, we omit them here. We recall that the definitions omitted here are given in Appendix B.

We also require a way of expressing the fact that one class is a subclass of another. We say that a Class binding, c₁, is a direct subclass of another class, c₂, written c₁ ⊆ₜ c₂, if the this identifier of c₂ is the super identifier of c₁ or one of its interfaces identifiers.

We also define a relation, subclassRel, between class identifiers cid₁ and cid₂ in terms of the ⊆ₜ relation. This requires a map from ClassID to bindings of Class, which is provided as a parameter to subclassRel. Given such a map, cs, we define (cid₁, cid₂) ∈ subclassRel cs to hold if, in cs, cid₁ and cid₂ refer to Class bindings such that cs cid₁ ⊆ₜ cs cid₂ holds. We expand subclassRel to refer to its reflexive transitive closure so that it includes indirect subclass relationships and classes being subclasses of themselves. We omit the formal definitions of ⊆ₜ and subclassRel here.

78
The $cs$ map provided as a parameter to $CEE$ is used as the parameter to $subclassRel$ in each of the processes that uses it. In order for the $CEE$ to execute the program, this $cs$ parameter must represent a valid SCJ program, with all the necessary classes present. If this holds, then $subclassRel$ represents the usual notion of when a object of a Java class is assignable to a variable of a given class.

We next describe the object manager process, $ObjMan$, which uses the $Class$ type and the $cs$ map.

### 4.3.3 Object Manager

The object manager, which is represented by the process $ObjMan$, manages the objects of the SCJ program executed by the core execution environment. This component is necessary because the SCJVM memory manager is agnostic to the structure of objects, which depends on the contents of the classes supplied as part of an SCJ program. Besides managing the creation and manipulation of objects, the object manager tracks the current allocation context for each thread. It ensures objects are allocated in the correct area, since the SCJVM memory manager is also agnostic to the existence of threads.

The start of the definition of $ObjMan$ is shown below. $ObjMan$ takes a single parameter, $cs$, which is a map from $ClassIDs$ to $Class$ records containing the class information for each of the classes in the program. This information is used in determining the structure of the objects for each class.

\[
\text{process } ObjMan \equiv cs : ClassID \rightarrow Class \cdot \text{begin}
\]

Since the actual arrangement of an object in memory is an implementation consideration, the amount of memory required to store each object is implementation-defined. It is represented here by a global function $sizeOfObject$, declared below, which maps the information in each $Class$ to the amount of memory required for objects of the class it represents.

\[
\text{sizeOfObject} : Class \rightarrow \mathbb{N}
\]

The objects that $ObjMan$ operates on are described by records of the schema type $Object$ shown below. It contains a map, $fields$, which associates the $FieldID$ for each field of an object with a $Word$ value stored in that field. A copy of the $Class$ information for the object’s class is also recorded in $class$. The invariant of $Object$ requires that the domain of $fields$ be the same as the fields given in $class$.

\[
\begin{array}{l}
\text{Object} \\
\text{fields} : FieldID \rightarrow \text{Word} \\
\text{class} : \text{Class} \\
\text{dom fields} = \text{class.fields}
\end{array}
\]

The state of $ObjMan$ is described by the schema $ObjManState$, shown below. Its first component is a map, $objects$, from $ObjectIDs$ to $Object$ records, storing the actual object data for the program. Its second component, $backingStoreMap$, associates backing stores with the identifiers of objects stored within them. The third component, $backingStoreStacks$, associates to each $ThreadID$ a sequence of $BackingStoreID$ values representing the backing stores entered by the thread. This is required to contain at least one value, since each thread that is ready to execute must have a backing store, and those that are not should not be in the domain of $backingStoreStacks$. $ObjManState$ also stores $rootBS$, a copy of the root backing store identifier. The final components of $ObjManState$ are used to store the static fields of classes. The $staticClassFieldsID$ component is of a free type $StaticFieldsStructID$, and may be either $Uninitialised$ or $Initialised$ with an $ObjectID$ representing the space allocated for the static class fields. The values of the fields themselves are stored in the $staticClassFields$ map.
The first conjunct of the invariant requires that the domain of `objects`, plus `staticClassFieldsID` if it is `Initialised`, is partitioned by the `ObjectID` sets given by `backingStoreMap`, so that every object is allocated in exactly one backing store. This uses a function `toSet` to convert `staticClassFieldsID` to a set of `ObjectID`s. The second conjunct requires that the domain of `backingStoreMap` be the identifiers of backing stores in `backingStoreStacks`, since a backing store that is not in use is cleared and so should not have any objects in it. The third conjunct requires that `rootBS` always be in the domain of `backingStoreMap`, since that represents immortal memory, which is never cleared. Combined with the second conjunct, this implies that at least one thread must have entered the `rootBS` at all times. The fourth conjunct ensures that `staticClassFields` does not contain any fields until `staticClassFieldsID` is `Initialised`. Finally, the fifth conjunct requires that `staticClassFieldsID` must be in the root backing store.

The `ObjManState` is initialised as described in `ObjManInit` below. This operation takes the identifier of the root backing store as an input, `rootBS?`. The `objects` map is initialised to the empty set, since there are initially no objects in existence. The `backingStoreMap` initially contains only `rootBS?`, which is the only backing store initially in existence, associated with an empty set of object identifiers. The `backingStoreStacks` map is initialised to contain the `main` and `idle` thread, both with `rootBS?` as their only backing store entered. The `rootBS` identifier is set to be the same as the `rootBS?` input. The `staticClassFieldsID` is set to `Uninitialised`, with `staticClassFields` empty.

The `ObjMan` process then proceeds as described in its main action, shown below. It begins in `Init` by communicating with the SCJVM memory manager to obtain the identifier of the root backing store and then initialising the state as described in `ObjManInit`. The space for static fields is then allocated in `AllocateStaticFields`, initialising `staticClassFieldsID` and populating `staticClassFields` with each field mapped to `null`. Finally, in `Loop`, the process repeatedly offers each of its services in external choice.

- `Init`; `AllocateStaticFields`; `Loop`

After `ObjMan` is initialised and the space for the static fields has been created, `ObjMan` offers services to the other components of the CEE, in the `Loop` action shown below. The services offered by `Loop` include `NewObject`, which creates an object of a given class. The `GetField` and `PutField` actions allow for obtaining and setting the value of an object’s field. Similarly, `GetStatic` and `PutStatic` allow for
obtaining and setting the value of a class’ static fields, applying the same data operations as \texttt{GetField}
and \texttt{PutField} to the object in \texttt{classObjectMap}. \texttt{GetClassIDOf} obtains the \texttt{ClassID} for the class of an
object, by extracting the \texttt{this} identifier from the \texttt{Class} information for the object.

Management of allocation contexts is provided by the remaining services. The first is \texttt{EnterBackingStore},
which enters a backing store for a given thread by pushing it onto the stack in \texttt{backingStoreStacks}
for that thread, and adding it to \texttt{backingStoreMap} if it is not already in its domain. The second is
the corresponding operation \texttt{ExitBackingStores} for exiting the current allocation context of a given
thread, which means popping it from the thread’s stack in \texttt{backingStoreStacks}, and clearing and removing
the backing store from \texttt{backingStoreMap} if no threads are still using it. The \texttt{AddThreadMemory} and
\texttt{RemoveThreadMemory} services allow for adding a thread to \texttt{backingStoreStacks} when it starts executing,
and removing it when it finishes executing. Finally, the \texttt{GetCurrentAC} action obtains the current
allocation context for a given thread, which is the backing store on top of its stack in \texttt{backingStoreStacks}.

The compilation refines the structure of objects. This means that field access operations (\texttt{GetField},
\texttt{PutField}) and object allocation (\texttt{NewObject}) are affected. \texttt{GetClassIDOf} is also affected since an object’s
class identifier is stored as part of its structure. We also refine the static fields data structure, requiring
the operations upon it (\texttt{AllocateStaticFields}, \texttt{GetStatic}, \texttt{PutStatic}) to be changed. The management of
allocation contexts is unaffected by the compilation, but is required for managing allocation of objects,
so that they can be allocated in the correct backing store.

The definition of the \texttt{NewObject} action is shown below. It allocates space for an object by communicating
with the memory manager in much the same way as \texttt{AllocateStaticFields}, except that the thread and
class identifiers for the object are communicated via the \texttt{newObject} channel, and the size required for the
object is obtained from the class information via a \texttt{sizeOfObject} function. After a successful allocation,
the object is added to the \texttt{objects} map with its fields initialised to \texttt{null}, in \texttt{ObjManObjectInit}, and the
object’s identifier is returned via \texttt{newObjectRet}.

\[
\text{NewObject} \equiv \text{var objectID : ObjectID \bullet}
\begin{align*}
\text{newObject?thread?classID \rightarrow} \\
\text{MMallocateMemory\!(last\!(backingStoreStacks\! thread))\!(sizeOfObject\!( cs classID)) \rightarrow} \\
\text{MMallocateMemoryRet\!objectID \rightarrow} \\
\text{(MMreport?r : (r = MMokay) \rightarrow (\exists\ class? == cs classID \bullet ObjManObjectInit) ;} \\
\text{newObjectRet\!objectID \rightarrow Skip) \bullet} \\
\text{MMreport?r : (r \neq MMokay) \rightarrow Chaos)}
\end{align*}
\]

We omit the definitions of the other actions of \texttt{Loop} here. The full model of the object manager can be
found in Appendix B.3.

Next, we discuss the \texttt{Interpreter} process, which is the final component of CEE and handles the execution
of the bytecode instructions themselves.

### 4.3.4 Interpreter

The \texttt{Interpreter} process is the final component of CEE that we present. It handles the execution
of bytecode instructions. The bytecode instructions it handles are those in the subset described in
Section 4.3.1 which in our model are represented by the free type \texttt{Bytecode}, sketched below. \texttt{Bytecode}
has a constructor for each bytecode instruction, with any parameter to the instruction represented as a
parameter of the constructor. The full definitions omitted here are in Appendix B.5.

\[
\text{Bytecode ::= aconst\_null | dip | areturn | return | iadd | ineg}
| new\!\{}\text{CPIndex}\!\} | iconst\!\{}\text{N}\!\} | aload\!\{}\text{N}\!\} | astore\!\{}\text{N}\!\} | \ldots
\]

81
The bytecode instructions are arranged in a map, \( bc \), from \( \text{ProgramAddress} \) values (which are modeled by natural numbers) to \( \text{Bytecode} \) values. The \( bc \) map is passed as a parameter to the \( \text{Interpreter} \) process, along with the \( cs \) map described in Section 4.3.2 as can be seen from its definition shown below. The overall structure of \( \text{Interpreter} \) is a parallel composition of \( \text{Thr} \) processes representing the individual interpreter threads, with one process for each \( \text{ThreadID} \) except for \( \text{idle} \). The \( bc \) and \( cs \) parameters are passed to each \( \text{Thr} \) process, along with its \( \text{ThreadID} \).

\[
\text{process } \text{Interpreter} ::= \\
\text{bc : ProgramAddress } \mapsto \text{Bytecode}; \text{cs : ClassID } \mapsto \text{Class} \bullet \\
\lor t : \text{ThreadID} \setminus \{ \text{idle} \} [\text{ThrChans}(t)] \bullet \text{Thr}(bc, cs, t)
\]

Each \( \text{Thr}(bc, cs, t) \) process synchronises on a set \( \text{ThrChans}(t) \), containing the \( \text{CEEswitchThread}.t.12 \) and \( \text{CEEswitchThread}.t.12.t \) events for all thread identifiers \( t2 \). This ensures thread switches can be handled since the two threads involved in the thread switch (the thread switched from and the thread switched to) synchronise on the thread switch request. This model of the interpreter threads captures the fact that they are conceptually running in parallel, each with their own state, and we do not mandate a specific thread switch mechanism.

**State**

The state of each \( \text{Thr} \) process contains the stack for the thread, which consists of a series of stack frames, one for each method on the call stack. The contents of each stack frame are specified by the schema \( \text{StackFrame} \). Its first component, \( \text{localVariables} \), is a sequence of \( \text{Word} \) values representing the local variable array for the method. Its second component, \( \text{operandStack} \) represents the data stack upon which each bytecode instruction operates. The third component, \( \text{storedPC} \), is used for storing the program counter value as a return address when another method is invoked. The fourth component, \( \text{frameClass} \), is a copy of the \( \text{Class} \) information for the class of the stack frame’s method, so that the constant pool for the class is available to the operations of \( \text{Thr} \). The fifth component, \( \text{stackSize} \), stores the maximum size of the \( \text{operandStack} \) for the thread. The final component of \( \text{StackFrame} \), \( \text{baseFrame} \) is a boolean value indicating whether or not the frame was created in response to an external request, so that it forms the base of a substack of frames. This is used when determining whether to send a return value to the \( \text{executeMethodRet} \) channel or push it onto the previous method’s \( \text{operandStack} \).

\[
\text{StackFrame}
\begin{align*}
\text{localVariables} & : \text{seq } \text{Word} \\
\text{operandStack} & : \text{seq } \text{Word} \\
\text{storedPC} & : \text{ProgramAddress} \\
\text{frameClass} & : \text{Class} \\
\text{stackSize} & : \mathbb{N} \\
\text{baseFrame} & : \mathbb{B}
\end{align*}
# \text{operandStack} \leq \text{stackSize}
\]

The invariant of \( \text{StackFrame} \) just requires the \( \text{operandStack} \) to not be larger than \( \text{stackSize} \).

The state of \( \text{Thr} \) is given by the schema \( \text{InterpreterState} \), below. Its first component, \( \text{frameStackID} \), is the identifier of the frame stack for the thread. It is of a type \( \text{Stack} \), which may be \( \text{Uninitialised} \) with no value, or \( \text{Initialised} \) with a \( \text{StackID} \) value for the stack. The second component, \( \text{frameStack} \), represents the stack itself, which is a sequence of \( \text{StackFrame} \) bindings, with one for each method entered. The third component, \( \text{pc} \), is the program counter for the thread. Finally, the fourth component, \( \text{currentClass} \), is a copy of the class information for the current method.

82
Figure 4.3: The overall control flow of Thr

\[
\begin{align*}
\text{Init} & \rightarrow \text{MainThread} \quad \text{Running} \quad \text{Blocked} \\
\text{NotStarted} & \rightarrow \text{Blocked} \quad \text{Running}
\end{align*}
\]

\section*{InterpreterState}

\begin{itemize}
\item \text{frameStackID : Stack}
\item \text{frameStack : seq StackFrame}
\item \text{pc : ProgramAddress}
\item \text{currentClass : Class}
\end{itemize}

\begin{align*}
\text{frameStackID} &= \text{Uninitialised} \Rightarrow \text{frameStack} = \emptyset \\
\text{frameStack} \neq \emptyset \Rightarrow \text{currentClass} = (\text{last frameStack}).\text{frameClass} \\
\text{frameStack} \neq \emptyset \Rightarrow (\text{head frameStack}).\text{baseFrame} = \text{True} \\
\exists m : \text{MethodID} \mid m \in \text{dom currentClass}.\text{methodEntry} \quad \bullet \\
\text{pc} \in \text{currentClass}.\text{methodEnd m} \quad \text{.. currentClass}.\text{methodEnd m}
\end{align*}

The first conjunct of the invariant requires that there be no StackFrames on the frameStack when frameStackID is Uninitialised. The second conjunct defines currentClass as the frameClass of the last StackFrame on the frameStack. This is only required when frameStack is nonempty, since currentClass is unused when frameStack is empty. The third conjunct states that, if the frameStack is nonempty, the first StackFrame must have baseFrame set to True, since it is always the start of a substack of frames. Finally, the fourth conjunct states that pc must be in the ProgramAddress range for a method of currentClass, to ensure the bytecodes are associated with the correct class information.

The state is initialised as described in a schema InterpreterInit. The initialisation just sets frameStackID to Uninitialised and the frameStack to empty. The other state components take arbitrary values and are initialised when the first StackFrame is created, since they are unused until then.

\section*{Behaviour}

The main action of Thr is shown below. After the initialisation, it behaves as MainThread or NotStarted, depending on whether the thread represented by the Thr process is the main thread or not. MainThread and NotStarted make use of the same actions for executing bytecode instructions, but they occur in different orders. The control flow of the Thr process is shown in Figure 4.3.

\begin{align*}
\bullet \left( \text{InterpreterInit} \right) & ; \\
& \left( \begin{array}{l}
\text{thread = main} \land \text{MainThread} \\
\text{thread \neq main} \land \text{NotStarted}
\end{array} \right)
\end{align*}

The MainThread action is shown below. It begins by accepting a StackID from the Launcher on the initMainThread channel, which is used to initialise the frameStackID state component. It then offers a choice of executing a method on the main thread in response to a request from the Launcher, or switching to another thread. A request to start execution of a method is handled in the StartInterpreter action, which creates the StackFrame for the method. The process then behaves as the Running action, executing bytecode instructions until the method has finished, after which the MainThread action recurses to offer the choice of method execution and thread switch again. If an instruction to switch to another thread from the scheduler is received on the CEEswitchThread channel, then it is only accepted if the thread
switched from is the thread represented by the process. If it is accepted, then the process behaves as Blocked, waiting for a request to switch back to the thread, after which it recurses back to offer the choice of behaviours again.

\[
\begin{align*}
\text{MainThread} & \equiv \text{initMainThread?stack} \rightarrow \text{frameStackID} := \text{Initialised stack} \land \mu X \bullet \left( \begin{array}{l}
\text{StartInterpreter} \land \text{Running} \land X \quad \square \\
\text{CEEswitchThread?from?to} : (\text{from} = \text{thread}) \rightarrow \text{Blocked} \land X
\end{array} \right)
\end{align*}
\]

The StartInterpreter action, used by MainThread also handles special methods that execute nested methods. Its definition is shown below. It accepts communication on the executeMethod channel, requiring the ThreadID communicated to be the same as that of the current thread, thread, and storing the other values communicated as classID, methodID and methodArgs. A data operation ResolveMethod is then used to determine the appropriate class information for the method, since the method may actually be defined in a superclass of the provided classID. ResolveMethod follows the method resolution rules of the JVM specification, first checking if the class corresponding to classID defines the method, then checking if one of its superclasses defines the method, and finally looking for the method definition among its superinterfaces. The Class information resulting from this is stored in class and used to create a new StackFrame on the frameStack in InterpreterNewStackFrame. The baseFrame value for this stack frame is set to True, since it is being created in response to an external request.

\[
\begin{align*}
\text{StartInterpreter} & \equiv \begin{array}{l}
\text{var classID : ClassID; methodID : MethodID; methodArgs : seq Word; class : Class} \\
\text{executeMethod? : (t = thread)?c?m?a \rightarrow classID, methodID, methodArgs := c, m, a;}
\end{array} \\
\begin{cases}
\text{ResolveMethod}[cs/cs?] ; (\exists \text{baseFrame} \equiv \text{True} \land \text{InterpreterNewStackFrame})
\end{cases}
\end{align*}
\]

For threads other than main, the behaviour is described by the NotStarted action below. It accepts a request to start the thread represented by the process from the scheduler on the CEEstartThread channel. The identifier, bsid, of the thread’s backing store is then passed to ObjMan via the addThreadMemory channel and the stack identifier is used to initialise frameStackID. The remaining information is stored in classID, methodID and methodArgs, and the Class information is determined and a new StackFrame is created, similarly to what happens in StartInterpreter. The process then behaves as the Blocked action, waiting for an instruction to switch to that thread, after which it behaves as the Running action, executing bytecode instructions. When the execution of the thread’s method has finished, the scheduler is signalled on the CEEremoveThread channel. This causes the scheduler to switch to a different thread, so a thread switch is accepted on the CEEswitchThread channel. After that, the action recurses to allow the thread to be restarted.

\[
\begin{align*}
\text{NotStarted} & \equiv \begin{array}{l}
\text{var classID : ClassID; methodID : MethodID; methodArgs : seq Word; class : Class} \\
\text{CEEstartThread?toStart?bsid?stack?cid?mid?args : (toStart = thread) \rightarrow}
\end{array} \\
\begin{cases}
\text{addThreadMemory!thread!bsid} \rightarrow \\
\text{frameStackID, classID, methodID, methodArgs := Initialised stack, cid, mid, args;}
\end{cases} \\
\begin{cases}
\text{ResolveMethod}[cs/cs?] ; (\exists \text{baseFrame} \equiv \text{True} \land \text{InterpreterNewStackFrame}) ;
\end{cases} \\
\text{Blocked \land Running \land CEEremoveThread!thread} \rightarrow \\
\text{CEEswitchThread?from?to} : (\text{from} = \text{thread}) \rightarrow \text{NotStarted}
\end{align*}
\]

The Blocked and Running actions define the behaviour of threads after they have been started. The Blocked action simply waits for a signal on the CEEswitchThread channel to switch to thread, after which it terminates to allow execution in a different action to continue.

\[
\begin{align*}
\text{Blocked} & \equiv \text{CEEswitchThread?from?to} : (to = \text{thread}) \rightarrow \text{Skip}
\end{align*}
\]

The Running action, shown below, executes the bytecode instructions of a program. It has the form of a loop that repeatedly executes until frameStack is empty. Within the loop, it handles the bytecode
instruction at the current pc value in HandleInstruction and then it polls for thread switches in Poll.

\[
\text{Running} \triangleq \\
\text{if } \text{frameStack} = \emptyset \longrightarrow \text{Skip} \\
\text{else if } \text{frameStack} \neq \emptyset \longrightarrow \text{HandleInstruction} ; \text{Poll} ; \text{Running} \\
\text{fi}
\]

The behaviour of polling for thread switches in Poll permits thread switches inbetween bytecode instructions. Implementations that allow thread switches at other points are valid if they retain the same sequence of externally visible events, meaning only instructions involving communication with other parts of the model need be atomic. Poll simply offers communication from the scheduler on the CEEswitchThread and CEEproceed channels, switching to Blocked upon receiving a signal on CEEswitchThread, and terminating on receiving a signal on CEEproceed.

The HandleInstruction action, shown in part below, offers a choice of actions for handling the bytecode instructions. There is one action for each of the instructions, with the action’s name formed from the bytecode mnemonic prefixed with Handle (e.g. HandleAload for the aload instruction).

\[
\text{HandleInstruction} \triangleq \\
\text{HandleAconst} \text{null} \circ \text{HandleDup} \circ \text{HandleAload} \circ \text{HandleAstore} \circ \text{HandleIadd} \circ \cdots
\]

The Handle actions define the semantics for the instructions and, as such are involved in the compilation strategy. Many of these actions for handling bytecode instructions have a similar form.

Bytecode Semantics

The simplest Handle actions consist of a guard requiring the bc value at the current pc to be a particular bytecode instruction, followed by a data operation specified by a Z schema updating InterpreterState. This is illustrated in the definition of HandleAconst_null below, which uses the InterpreterAconst_null schema.

\[
\text{HandleAconst_null} \triangleq \text{(bc pc = aconst_null)} \& \left( \text{InterpreterAconst_null} \right)
\]

The Circus actions and Z schemas for each bytecode instruction are listed in Table 4.3. We omit the definitions of the Z schemas in our description here. Their contents are in line with the state updates for the bytecode instructions presented in Table 4.2.

The HandleDup, HandleIadd and HandleIneg actions follow the simple form exemplified above. Some instructions have parameters that must be extracted so that they can be passed to the data operation for the instruction. This can be seen in the definition of the HandleAload action, shown below, in which the inverse of the aload constructor is used to extract its parameter into a variableIndex variable that is used by the InterpreterAload schema.

\[
\text{HandleAload} \triangleq \text{(bc pc \in ran aload)} \& \\
\text{var variableIndex : N \& variableIndex := (aload ~) (bc pc) ; (InterpreterAload)}
\]

The HandleAstore, HandleGoto, HandleIconst and HandleIf_iemple actions all follow a similar form, extracting the parameter of the bytecode instruction into a separate variable.

The actions to handle the return instructions (areturn and return) require additional communication to pass the return value to the Launcher when returning from a method that has been started by the Launcher. This is performed by an additional action, CheckLauncherReturn, that is called after return from the schema action. This can be seen in the definition of HandleAreturn below, where CheckLauncherReturn is passed the two output values from the InterpreterAreturn function. The first output, returnValue, holds the return value for the returning method, while the second, fromBaseFrame, is a boolean value indicating if the StackFrame for the method had its baseFrame value set to True.

\[
\text{HandleAreturn} \triangleq \text{var returnValue : Word; fromBaseFrame : } \mathbb{B} \& \\
\text{(bc pc = areturn) \& (InterpreterAreturn)} ; \\
\text{CheckLauncherReturn(returnValue, fromBaseFrame)}
\]
<table>
<thead>
<tr>
<th>Bytecode instruction</th>
<th>Circus action</th>
<th>Z schema</th>
</tr>
</thead>
<tbody>
<tr>
<td>aconst_null</td>
<td>HandleAconst_null</td>
<td>InterpreterAconst_null</td>
</tr>
<tr>
<td>aload</td>
<td>HandleAload</td>
<td>InterpreterAload</td>
</tr>
<tr>
<td>areturn</td>
<td>HandleAreturn</td>
<td>InterpreterAreturn</td>
</tr>
<tr>
<td>astore</td>
<td>HandleAstore</td>
<td>InterpreterAstore</td>
</tr>
<tr>
<td>dup</td>
<td>HandleDup</td>
<td>InterpreterDup</td>
</tr>
<tr>
<td>getfield</td>
<td>HandleGetfield</td>
<td>InterpreterPush</td>
</tr>
<tr>
<td>getstatic</td>
<td>HandleGetstatic</td>
<td>InterpreterPush</td>
</tr>
<tr>
<td>goto</td>
<td>HandleGoto</td>
<td>InterpreterGoto</td>
</tr>
<tr>
<td>iadd</td>
<td>HandleIadd</td>
<td>InterpreterIadd</td>
</tr>
<tr>
<td>iconst</td>
<td>HandleIconst</td>
<td>InterpreterIconst</td>
</tr>
<tr>
<td>if_icmple</td>
<td>HandleIf_icmple</td>
<td>InterpreterIf_icmple</td>
</tr>
<tr>
<td>ineq</td>
<td>HandleIneg</td>
<td>InterpreterIneg</td>
</tr>
<tr>
<td>invokespecial</td>
<td>HandleInvokespecial</td>
<td>InterpreterStackFrameInvoke, InterpreterNewStackFrame</td>
</tr>
<tr>
<td>invokestatic</td>
<td>HandleInvokestatic</td>
<td>InterpreterStackFrameInvoke, InterpreterNewStackFrame</td>
</tr>
<tr>
<td>invokevirtual</td>
<td>HandleInvokevirtual</td>
<td>InterpreterStackFrameInvoke, InterpreterNewStackFrame</td>
</tr>
<tr>
<td>new</td>
<td>HandleNew</td>
<td>InterpreterPush</td>
</tr>
<tr>
<td>putfield</td>
<td>HandlePutfield</td>
<td>InterpreterPop2</td>
</tr>
<tr>
<td>putstatic</td>
<td>HandlePutstatic</td>
<td>InterpreterPop</td>
</tr>
<tr>
<td>return</td>
<td>HandleReturn</td>
<td>InterpreterReturn</td>
</tr>
</tbody>
</table>

Table 4.3: The relationship between the bytecode instructions in our subset and the Circus actions and Z schemas defining them.

The form of the HandleReturn action is similar, but since InterpreterReturn does not output a return value, returnValue takes an arbitrary value.

Within the CheckLauncherReturn action, the definition of which is shown below, the boolean value fromBaseFrame is used to determine whether the return value should be communicated to the Launcher. If fromBaseFrame is set to True, then that normally indicates that the return value needs to be passed to the launcher. The exception to this is if the return is from the first frame of a thread other than main, which is created in response to a CEEstartThread signal from the scheduler and so does not require a return value to be sent to the launcher. Thus, the condition checked is that fromBaseFrame is True, and either frameStack is empty or the current thread is main. If that is true, then returnValue is sent to the Launcher on the executeMethodRet channel and a signal is awaited on the continueExecution channel before continuing. If the condition is not true then the action terminates.

\[
\text{CheckLauncherReturn} \triangleq \text{val returnValue : Word; val fromBaseFrame : B} \bullet
\]

\[
\begin{align*}
\text{if fromBaseFrame} &= \text{True} \land (\text{frameStack} \neq \emptyset \lor \text{thread} = \text{main}) \rightarrow \\
&\quad \text{executeMethodRet[thread?returnValue } \rightarrow \text{continueExecution?t : (t = thread) } \rightarrow \text{Skip} \\
&\quad \text{fromBaseFrame} = \text{False} \lor (\text{frameStack} = \emptyset \land \text{thread} \neq \text{main}) \rightarrow \text{Skip} \\
\end{align*}
\]

For the instructions that create objects and access their fields (\texttt{new, getField, putfield, getstatic} and \texttt{putstatic}), communication with ObjMan is needed. This can be seen in the definition of HandleGetfield, shown below, where the object identifier, oid, is popped from the operandStack of the current StackFrame using the data operation InterpreterPop, and the field identifier is extracted from the parameter to the bytecode instruction. Note that pc is hidden from the InterpreterPop, since this action uses two data operations promoted from StackFrame operations and only one of them need update pc. This information is passed to ObjMan via the getField channel, which then handles the operation. The field’s value is returned via the getFieldValue channel and pushed onto the operandStack of the current StackFrame by
the data operation InterpreterPush.

\[
\text{HandleGetfield} \triangleq (bc, pc) \in \text{ran getfield} \land (\text{getfield}^{-})((bc, pc) \in \text{fieldRefIndices currentClass}) \\
\text{var oid} : \text{ObjectID} \bullet (\text{InterpreterPop}\{oid!\} / \text{value}! \in ((bc, pc'), (pc, pc'))); \\
\text{getField}\{oid!\} / \text{fieldOf currentClass}((\text{getfield}^{-})((bc, pc))) \\
\text{\rightarrow getFieldRet! value} \rightarrow (\text{InterpreterPush})
\]

The \text{HandleNew}, \text{HandlePutfield}, \text{HandleGetstatic} and \text{HandlePutstatic} actions are similar, getting information from the \text{operandStack} using the \text{InterpreterPop}, communicating with \text{ObjMan} to handle the instruction, and using \text{InterpreterPush} to push returned information onto the \text{operandStack}.

Finally, the method invocation instructions (\text{invokespecial}, \text{invokestatic} and \text{invokevirtual}), require special handling by the virtual machine. Since the different method invocation instructions differ only in how the class for the method is determined and whether a this object identifier is passed among the method’s arguments, the invocation of the method after this has been determined is handled by a common \text{Invoke} action. This can be seen in the definition of the \text{HandleInvokeSpecial} action below. The method identifier \text{mid} is extracted from the instruction’s parameter, and the data operation \text{InterpreterStackFrameInvoke} is used to store the return \text{pc} address and pop the arguments of the method in \text{poppedArgs}. The number of arguments popped, \text{argsToPop}?, is the \text{methodArguments} value for \text{mid}, plus one for the this identifier passed to the method. The class identifier is then extracted from the instruction’s parameter and passed into \text{Invoke} along with \text{mid} and \text{poppedArgs}.

\[
\text{HandleInvokeSpecial} \triangleq \text{var cid} : \text{ClassID}; \text{mid} : \text{MethodID}; \text{poppedArgs} : \text{seq Word} \bullet \\
(bc, pc) \in \text{ran invokemethod} \land ((\text{invokemethod}^{-})((bc, pc)) \in \text{methodRefIndices currentClass}) \& \\
\text{mid} := \text{methodOf currentClass}((\text{invokemethod}^{-})((bc, pc)));
(\exists \text{argsToPop}? \rightleftharpoons \text{argsToPop} \rightarrow \text{methodArguments} \text{mid} + 1 \bullet \text{InterpreterStackFrameInvoke}); \\
\text{Invoke}\{\text{classOf currentClass}((\text{invokemethod}^{-})((bc, pc)), \text{mid}, \text{poppedArgs})
\]

The \text{HandleInvokeStatic} and \text{HandleInvokeVirtual} actions are similar, except that \text{argsToPop}? does not include the extra this argument in \text{HandleInvokeStatic}, and \text{HandleInvokeVirtual} obtains the class identifier from the this identifier via the \text{getClassIDOf} channel rather than from the instruction’s parameter.

The \text{Invoke} action, shown in part below, has the form of an external choice over actions for each of the special methods supported by the SCJVM, plus an \text{InvokeOther} action for handling non-special methods implemented in bytecode. The name of the action for each special method is formed from the name of the special method prefixed with \text{Invoke} (e.g. \text{InvokeResumeThread} for the \text{resumeThread()} method). The parameters passed to the \text{Invoke} action, which are the class identifier, \text{classID}, the method identifier, \text{method}, and the method argument list, \text{args}, are passed on to each of the actions in the external choice.

\[
\text{Invoke} \triangleq \text{val classID} : \text{ClassID}; \text{val method} : \text{MethodID}; \text{val args} : \text{seq Word} \bullet \\
\text{InvokeResumeThread}(\text{classID}, \text{method}, \text{args}) \Box \text{InvokeSuspend}(\text{classID}, \text{method}) \Box \cdots
\Box \text{InvokeOther}(\text{classID}, \text{method}, \text{args})
\]

Within the special method actions, there is a guard ensuring the action is taken when the class and method identifiers are those for the method. The method is then handled by communication on the appropriate channels. This is illustrated by the definition of the \text{InvokeResumeThread} action, shown below. The class identifier parameter, \text{classID}, is required to refer to a subclass of some class \text{resumeThreadClass}, while the method identifier, \text{method}, must be \text{resumeThreadID}. The class and method identifiers used in the special method actions are a mixture of identifiers from the SCJ API and implementation-defined identifiers provided to expose SCJVM services to bytecode programs. The argument to the method, stored as the first element of the \text{methodArgs} parameter, is converted to a \text{ThreadId} and passed to the \text{Launcher} via the \text{resumeThread} channel. A return signal is then awaited on the \text{resumeThreadRet} channel before continuing.

\[
\text{InvokeResumeThread} \triangleq \\
\text{val classID} : \text{ClassID}; \text{val method} : \text{MethodID}; \text{val methodArgs} : \text{seq Word} \bullet \\
((\text{classID}, \text{resumeThreadClass}) \in \text{subclassRel class} \land \text{method} = \text{resumeThreadID}) \& \\
\text{resumeThread}!(\text{WordToThreadID}(\text{methodArgs} 1)) \rightarrow \text{resumeThreadRet} \rightarrow \text{Skip}
\]
Some of the special methods start other methods as part of their execution, which requires additional handling. An example is `enterPrivateMemory()` from the SCJ API, which executes the `run()` method of a `Runnable` object after entering a private memory area. This is handled in the action `InvokeEnterPrivateMemory`, shown below. It begins in a similar way to `InvokeResumeThread`, with a guard on the `classID` and `method` passed into the action. The method is then handled by communicating with the `Launcher` on the `enterPrivateMemory` channel, passing the arguments from `methodArgs`. The execution of the nested method is then handled as in `StartInterpreter`, waiting for the request to execute the nested method on the `executeMethod` channel and creating a new `StackFrame` for it.

\[
\text{InvokeEnterPrivateMemory} \triangleq \\
\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet \\
\left(\left(\text{classID, managedMemoryClass} \in \text{subclassRel cs } \land \text{method} = \text{enterPrivateMemoryID}\right) \land \text{enterPrivateMemory!thread!(methodArgs 1)!(methodArgs 2)} \rightarrow \text{StartInterpreter}\right)
\]

In addition to the special methods handled in the `Launcher`, we also supply `read()` and `write()` methods for reading from and writing to some standard input and output device. These methods are handled using the `input` and `output` channels that communicate the values from and to the environment of the SCJVM. This is shown in the definition of the `InvokeRead` action below, which accepts the input on the `input` channel and pushes it onto the stack as the return value for the method.

\[
\text{InvokeRead} \triangleq \\
\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet \\
\left(\left(\text{classID, readClass} \in \text{subclassRel cs } \land \text{method} = \text{readID}\right) \land \text{input?value } \rightarrow \left(\text{InterpreterPush } \left(\text{pc, pc}^\prime\right)\right)\right)
\]

The `InvokeWrite` action is similar, writing the method argument to the `output` channel.

The `InvokeOther` action, shown in part below, describes the handling of non-special methods. It begins with a guard that is the conjunction of the negation of the guards for the invocation actions for the special methods. The actions starts execution of the method in the interpreter by finding its `Class` information with `ResolveMethod` and creating a new `StackFrame` with `InterpreterNewStackFrame`. The `baseFrame` value is set to `False` here, since the stack frame is created due to execution of a bytecode instruction in the interpreter, rather than in response to an external request.

\[
\text{InvokeOther} \triangleq \text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet \\
\left(\left(\text{classID, resumeThreadClass} \notin \text{subclassRel cs } \lor \text{methodID} \neq \text{resumeThreadID}\right) \land \ldots \land \left(\text{classID, writeClass} \notin \text{subclassRel cs } \lor \text{methodID} \neq \text{writeID}\right)\right) \land \\
\text{var class : Class } \bullet \\
\left(\text{ResolveMethod[cs/cs?!]}\right) \lor \left(\exists \text{baseFrame? } = \text{False } \bullet \text{InterpreterNewStackFrame}\right)
\]

This concludes our description of the handling of bytecode instructions, and of our description of the CEE before the application of the compilation strategy. In the next section we describe the model of the C code that is used for the output of the compilation strategy.

### 4.4 C Code Model

As mentioned previously, the CEE after compilation to C has a similar structure to the CEE before compilation, but the object manager is replaced with a struct manager and the interpreter is replaced with the C program. The struct manager is represented by a process `StructMancs`, and the C program by a process `CProgbc.cs`. These are placed in parallel composition with the `Launcher` process described in Section 4.2 to form a `CCEEbc.cs` process representing the CEE for a C program, as shown below.

\[
\text{CCEEbc.cs}(\text{sid, initOrder}) \triangleq \text{StructMancs} \parallel \text{CProgbc.cs} \parallel \text{Launcher}(\text{sid, initOrder})
\]
The subscripts here indicate that the processes depend on the $bc$ and $cs$ constants used as inputs to the compilation strategy. However, $bc$ and $cs$ are not true parameters of the processes and so are not referenced within the processes. We note that the $sid$ and $initOrder$ parameters to $Launcher$ remain as parameters here, since $Launcher$ is not transformed during the compilation strategy.

The channels used for communication between these processes are the same as those in Table 4.3 except that the $getField$, $getFieldRet$ and $putField$ channels are replaced with $getObject$, $getObjectRet$ and $putObject$ channels. The new channels are discussed in Section 4.4.2, which describes the struct manager. After that, in Section 4.4.3, the $CProg_{bc,cs}$ process is described.

### 4.4.1 Struct Manager

$StructMan_{cs}$ 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. For each class identifier $<classID_1>,...,<classID_n>$, we define a schema $<classID_k>Obj$ for $k \in \{1,...,n\}$, representing the objects of that class. They begin with a $classID$ component containing the class identifier of the object, so that polymorphic method calls can be made by choice over the object’s class. There is then a component for each of the fields $<fieldID_1>,...,<fieldID_{m_k}>$ of the class, with each component having the type $Word$.

$$
<classID_k>Obj
\begin{align*}
&\text{classID} : \text{ClassID} \\
&<fieldID_{k,1}> : \text{Word} \\
&\vdots \\
&<fieldID_{k,m_k}> : \text{Word}
\end{align*}
$$

The schema types for each type of object are combined into a single free type $ObjectStruct$. The constructor for each $<classID_k>$ is called $<classID_k>Con$, with a single parameter of type $<classID_k>Obj$.

$$ObjectStruct ::= <classID_1>Con\langle<classID_1>Obj\rangle \mid \ldots \mid <classID_n>Con\langle<classID_n>Obj\rangle$$

For each object type, we define a natural number constant $sizeof<classID_k>Obj$ that represents the result of applying C’s $sizeof$ operator to the struct represented by the corresponding $<classID_k>Obj$ type. We also define a function $classIDO$ for obtaining the value of the common $classID$ field from an $ObjectStruct$ value. Additionally, we define a $cast<classID_k>$ function for each $<classID_k>$, which maps an $ObjectStruct$ value to a $<classID_k>Obj$ value. This works not only for values in the range of the $<classID_k>Con$ constructor, but also for any class that is a subclass of $<classID_k>$, with the common fields copied across. Thus, $cast<classID_k>$ represents casting of C structs, where a struct can be truncated by casting to a struct whose fields are a prefix of it. Finally, we also define a function $update<classID_k>_\langle<fieldID_1>,...,<fieldID_{m_k}>\rangle$, for each class $<classID_k>$ and field identifier $<fieldID_{n_k}>$, which takes an $ObjectStruct$ and updates the field with a given value. This is a combined cast and update.

For the static class fields, the static fields from each class $<classID_1>,...,<classID_n>$ are collected together in a schema $StaticFields$, as shown below.

$$StaticFields
\begin{align*}
&<classID_1>_\langle<staticfieldID_{1,1},...<staticfieldID_{1,\ell_1}> : \text{Word} \\
&\vdots \\
&<classID_n>_\langle<staticfieldID_{n,1},...,<staticfieldID_{n,\ell_n}> : \text{Word}
\end{align*}$$

89
We define a constant `sizeofStaticFields` giving the space required for the struct represented by `StaticFields`.

Channels `getObject`, `getObjectRet`, and `putObject` are used to pass `ObjectStruct` values to and from `StructMan_cs`. The `StaticFields` type is communicated using the `getStaticFields` and `putStaticFields` channels.

The state of `StructMan_cs` is given by the schema `StructManState`. It is similar to `ObjManState`, but the `objects` map relates object identifiers to `ObjectStruct` values, and `staticClassFields` is a map from `ObjectID` to the `StaticFields` type.

The structure of the `StructMan_cs` process is much the same as for the `ObjMan` process, with the state initialised in the same way. However, `AllocateStaticFields` is changed to handle the `StaticFields` type. The definition of `AllocateStaticFields` in `StructMan_cs` is as shown below. It is similar to the `ObjMan` definition of `AllocateStaticFields`, but it does not collect the required field identifiers since that is already implicitly done in the definition of `StaticFields`. The required size for the allocated space is now `sizeofStaticFields`.

The initialisation of the static fields, in `InitStaticFields` is much the same, but now specialised to refer to `StaticFields`, setting all its components to `null`.

\[
\text{AllocateStaticFields} \equiv \\
\quad \text{MMallocateMemory}!(\text{last (backingStoreStacks main)})!(\text{sizeofStaticFields}) \rightarrow \\
\quad \text{MMallocateMemoryRet}?\text{objectID} \rightarrow \\
\quad \left(\text{MMreport}?r : (r = \text{Mokay}) \rightarrow (\text{InitStaticFields}) \right) \\
\quad \Box \text{MMreport}?r : (r = \text{Mokay}) \rightarrow \text{Chaos}
\]

Also, the actions `GetField`, `PutField`, `GetStatic` and `PutStatic` are replaced with new actions `GetObject`, `PutObject`, `GetStaticFields` and `PutStaticFields`. These are fairly simple, with `GetObject` and `PutObject` retrieving and storing `ObjectStruct` values in the `objects` map. These are exposed on the `getObject` and `getObjectRet` channels for `GetObject`, and the `putObject` channel for `PutObject`. Similarly, `GetStaticFields` and `PutStaticFields` retrieve and store the `StaticFields` value associated with `staticClassFieldsID` from the `staticClassFields` map. These are performed on the `getStaticFields` and `putStaticFields` channels respectively. We omit the definitions of these actions here; their definitions are given in Appendix B.8, which shows the general form of the `StructMan_cs` process.

The `NewObject` action is different in `StructMan_cs` to that in `ObjMan`. It uses the same channels (`newObject` and `newObjectRet`), but must create the correct `ObjectStruct` value for the provided class. It has the form shown below. The `thread` and `classID` identifiers are received from the `newObject` channel, like in `ObjMan`. A choice is then made over the `classID`, matching it against each class identifier supported by `StructMan_cs`. If `classID` matches a class identifier `<classID_k>`, then space for the object is allocated via communication with the memory manager, as in `AllocateStaticFields`, then the object is stored in `objects` and initialised. The allocation is performed in a separate action, `AllocateObject`, as it is similar for each class. The size of the object is given by the `sizeof <classID_k>` identifier for `<classID_k>`, and the returned object identifier is stored in `objectID`. The storing and initialisation of the object is then done in a schema action `StructMan <classID_k> ObjInit`, which sets all the object’s fields to `null` and puts it in `objects`, stored within `<classID_k> Con`. Finally, `objectID` is returned via the `newObjectRet` channel, as in `ObjMan`. The possibility of divergence if the memory manager reports an error is handled.
Finally, GetClassIDOf is changed to extract the class identifier from an ObjectStruct value using the classIDOf function. This is not used in the C code, which, as we will see in the next section, uses classIDOf directly, but is supplied for use by the Launcher, which is unchanged by the strategy. We omit the definition of GetClassIDOf here.

In the next section we describe the structure of the C code of the program and our shallow embedding of C in Circus, which makes use of some of the types and channels discussed in this section.

### 4.4.2 Shallow Embedding of C in Circus

The C code output by our compilation strategy is represented by a Circus process $CProg_{bc,cs}$, which is determined by the bytecode instructions, $bc$, and the class information, $cs$. This process has a similar structure to that of Interpreter: a parallel composition of $CThr_{bc,cs}(t)$ processes representing C threads, one for each thread identifier $t$ except the idle thread, as shown in the definition of $CProg_{bc,cs}$ below.

$$\text{process } CProg_{bc,cs} \equiv t : \text{ThreadID} \setminus \{\text{idle}\} \left[ \text{ThrChans}(t) \right] \bullet CThr_{bc,cs}(t)$$

The $CThr_{bc,cs}$ process has a similar structure to the Thr process presented in Section 4.3.4. However, the pc and frameStack components are eliminated from the state during compilation. Thus, the state of $CThr_{bc,cs}$, shown in $CThrState$ below, just contains stackID, which has the same type as frameStackID in InterpreterState and so may be Uninitialised or Initialised with a StackID for the C thread’s stack. Since there is no explicit stack in C, we only need to ensure there is a stack identifier in stackID so space for the thread’s stack has been allocated and the stackID is not manipulated beyond that.

$$\begin{align*}
CThrState \\
stackID : \text{Stack}
\end{align*}$$

The stackID is initially set to Uninitialised.

The Running action and creation of stack frames are also replaced with an ExecuteMethod action that executes the C function corresponding to a given method identifier. The main action of $CThr_{bc,cs}$ thus has the same structure as that of Interpreter, with a choice of MainThread for the main thread and NotStarted for non-main threads. However, MainThread now has the structure shown below. This is similar to the definition of MainThread in Thr, but the information received from the executeMethod channel is passed into the ExecuteMethod action to select the correct C function to execute. After method execution has finished, the return value, retVal, is obtained from ExecuteMethod and communicated on
the `executeMethodRet` channel.

\[\text{MainThread} \triangleq \text{initMainThread?stack} \rightarrow \text{stackID} := \text{Initialised stack} ; \mu X \bullet\]
\[
\begin{align*}
&\text{var retVal : Word} \bullet \text{executeMethod?t : (t = thread)?cid?mid?args} \rightarrow \\
&\text{ExecuteMethod(cid, mid, args, retVal)}; \\
&\text{executeMethodRet!retVal} \rightarrow \text{continueExecution} \rightarrow X
\end{align*}
\]

Similarly, the `NotStarted` action in `CThrnc.cs` has the form shown below, with `ExecuteMethod` after `Blocked`, replacing `Running`. The return value, `retVal`, is discarded here since the methods that are executed at the top level on the non-main threads should not return a value so `retVal` will be arbitrary. The thread’s removal is also handled by `CEEremoveThread` so no communication on `executeMethodRet` follows the end of the method’s execution.

`NotStarted \triangleq \text{var classID : ClassID; methodID : MethodID; methodArgs : seq Word; class : Class } \bullet$
\[
\begin{align*}
&\text{CEEstartThread?toStart?bsid?stack?cid?mid?args} : (\text{toStart = thread}) \rightarrow \\
&\text{addThreadMemory!thread!bsid} \rightarrow \\
&\text{frameStackID, classID, methodID, methodArgs := Initialised stack, cid, mid, args;} \\
&\text{Blocked ; var retVal : Word} \bullet \text{ExecuteMethod(classID, methodID, methodArgs, retVal)}; \\
&\text{CEEremoveThread!thread} \rightarrow \text{CEEswitchThread?from?to} : (\text{from = thread}) \rightarrow \text{NotStarted}
\end{align*}
\]

The `ExecuteMethod` action has the form shown below. It takes as parameters the class identifier, `cid`, method identifier, `mid`, and arguments list, `args`, for the method to be executed. It then chooses the appropriate action corresponding to the supplied `cid` and `mid`, and passes the appropriate number of arguments from `args` to the action. The return value of each of the actions, if they return one, is captured in `retVal` to be returned to `MainThread` or `Not Started`.

`ExecuteMethod \triangleq \text{val cid : ClassID; val mid : MethodID; val args : seq Word; res retVal : Word } \bullet$
\[
\begin{align*}
&\text{if (cid, mid) } = (\text{<classID}_1>, \text{<methodID}_1>) \rightarrow \\
&\text{<classID}_1>\langle\text{<methodID}_1>(\text{args}_1, \ldots, \text{args} (\text{methodArgs} <\text{methodID}_1>), \text{retVal}) \\
&\ldots \\
&\text{[ (cid, mid) } = (\text{<classID}_n>, \text{<methodID}_m>) \rightarrow \\
&\text{<classID}_n>\langle\text{<methodID}_m>(\text{args}_1, \ldots, \text{args} (\text{methodArgs} <\text{methodID}_m>), \text{retVal}) \\
&\text{if}
\end{align*}
\]

The actions used by `ExecuteMethod` represent C functions containing the behaviour of the compiled methods. The name of each action is made up of the class and method identifier for the method, separated by an underscore. Within the action, the constructs of C are represented by constructs of `Circus`. The representation of these constructs is summarised in Table 4.4.

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 [71] and present in the code generated by icecap. Conditionals in C correspond to `Circus` alternation blocks, similar to those in Dijkstra’s guarded command language [27]. We handle loops in C using recursion in `Circus`, with alternation used to handle loop conditions.

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 (via the object/struct manager), we represent function arguments using value parameters of the `Circus` action.

If a function has a return value, it is represented with a result parameter of the `Circus` action, usually named `retVal`, with an assignment to that parameter at the end of the action representing return
statements. We assume there is a single return statement at the end of the function, so the return can simply be represented by the termination of the action if there is no return value. 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 [71], 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).

Local variables of the function are represented using Circus variable blocks. These are placed near the start of the action, after the parameter declarations. While Circus variable blocks could also be used to represent variables declared in the middle of functions, that is not necessary for our work. Restricting ourselves to variables at the start of functions ensures the code or strategy generates is compatible with older versions of C.

### 4.5 Final Considerations

In this chapter we have presented our model of the core execution environment (CEE) of an SCJVM and specified the subset of Java bytecode covered in our model. Our bytecode subset consists of 14 instructions, which focus on method invocation and the manipulation of objects, since those are core concepts of Java. We have omitted instructions for exception handling, since that would complicate the model while adding little power. Our subset is sufficiently small to permit reasoning, but large enough to express a variety of SCJ programs.

Our CEE model is divided in three components, with a Circus process representing each component. The first component is the memory, which manages objects and the entering of backing stores, since the memory manager discussed in the previous chapter has no knowledge of the structure of objects. The second component of the CEE model is the interpreter, which describes the semantics of each of the bytecode instructions in our subset and provides for executing methods. The third and final component is the interpreter, which manages the SCJ mission model and coordinates execution.

One interesting point about our model is the handling of special methods in the interpreter and launcher. This is necessary for several reasons: to allow methods running in the interpreter to access the SCJVM services defined in the previous chapter, to allow mission setup methods to interact with the launcher, and to permit entering of memory areas by interaction with the CEE memory component. The handling of special methods works by having the interpreter check upon invocation of a method whether it requires special handling. If it does require special handling, it is passed to the launcher to be handled. The launcher then performs the required handling of the method, communicating with the SCJVM services and the memory as required.

This model forms the first part of our compilation strategy, which is the specification of the source language. That is mostly included in the interpreter section as the semantics of the bytecode instructions, though handling of special methods passed to the launcher and the representation of classes and objects must also be considered in the compilation strategy. There are also other possible uses for the model presented in this chapter. Since it is a model of an interpreting SCJVM, it could be used as a specification for an implementation of an interpreting SCJVM. Such an SCJVM could also incorporate the compilation strategy to provide a choice between interpreted and compiled code, as in the icecap HVM. Additionally, since error handling in our model is done via aborting execution, an identification of the conditions required for the model to be divergence-free would produce requirements that can be used for bytecode verification.
<table>
<thead>
<tr>
<th>Construct</th>
<th>C code</th>
<th>Circus equivalent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Function definition</td>
<td><code>void foo () { ... }</code></td>
<td><code>Foo ≡ ...</code></td>
</tr>
<tr>
<td>Function definition with argument</td>
<td><code>void bar(int32_t x) { ... }</code></td>
<td><code>Bar ≡ val x : Word</code></td>
</tr>
<tr>
<td>Function definition with return value</td>
<td><code>int32_t baz() { ... }</code></td>
<td><code>Baz ≡ res retval : Word</code></td>
</tr>
<tr>
<td>Function definition with parameter and return value</td>
<td><code>int32_t quux(int32_t x) { ... }</code></td>
<td><code>Quux ≡ val x : Word; res retval : Word</code></td>
</tr>
<tr>
<td>Function call</td>
<td><code>foo ();</code></td>
<td><code>Foo</code></td>
</tr>
<tr>
<td>Function call with argument</td>
<td><code>bar(x);</code></td>
<td><code>Bar(x)</code></td>
</tr>
<tr>
<td>Function call with return value</td>
<td><code>x = baz();</code></td>
<td><code>Baz(x)</code></td>
</tr>
<tr>
<td>Function call with argument and return value</td>
<td><code>y = quux(x);</code></td>
<td><code>Quux(x, y)</code></td>
</tr>
<tr>
<td>Return statement</td>
<td><code>return ;</code></td>
<td><code>Skip</code></td>
</tr>
<tr>
<td>Return statement with value</td>
<td><code>return x;</code></td>
<td><code>retval := x</code></td>
</tr>
<tr>
<td>Assignment</td>
<td><code>x = e;</code></td>
<td><code>x := e</code></td>
</tr>
<tr>
<td>Variable declaration</td>
<td><code>int32_t x;</code></td>
<td><code>var x : Word</code></td>
</tr>
<tr>
<td>Variable declaration and initialisation</td>
<td><code>int32_t x = e;</code></td>
<td><code>var x : Word; x := e</code></td>
</tr>
<tr>
<td>If statement</td>
<td><code>if (b) { ... }</code></td>
<td><code>if b → ... Skip</code></td>
</tr>
<tr>
<td>If-else statement</td>
<td><code>if (b) { ... } else { ... }</code></td>
<td><code>if b → ...</code></td>
</tr>
<tr>
<td>Infinite loop</td>
<td><code>while (1) { ... }</code></td>
<td><code>while b → ...</code></td>
</tr>
<tr>
<td>While loop</td>
<td><code>while (b) { ... }</code></td>
<td><code>while b → ...</code></td>
</tr>
<tr>
<td>Do-while loop</td>
<td><code>do {...} while (b);</code></td>
<td><code>do {...} while (b)</code></td>
</tr>
</tbody>
</table>
| Field read                                     | `y = ((X *) x)->f;`                         | `getObject x →
getObjectRet?struct →
y := (castX struct).f` |
| Field read                                     | `((X *) x)->f = y;`                         | `getObject x →
putObject(updateX_f struct y) → Skip` |

Table 4.4: The Circus representations of C constructs in our shallow embedding
Chapter 5

Compilation Strategy

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

**Theorem 5.0.1** (Compilation Strategy). Given $bc$, $cs$ and $sid$, there are processes $StructMan_{cs}$ and $CProg_{bc,cs}$ such that,

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

$StructMan_{cs}$ manages objects represented by C structs that incorporate the class information from $cs$, refining the process $ObjMan$, which handles abstract objects. $StructMan_{cs}$ has 2 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 $ObjectStruct$. $StructMan_{cs}$ contains a map from memory addresses managed by the SCJVM to the $ObjectStruct$ 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.

The compilation strategy is split into three stages, described in the following section. Each stage has a theorem describing it, for which the strategy acts as a proof. The proof of Theorem 5.0.1 is obtained by an application of the theorems for each stage.

### 5.1 Overview

Each stage of the compilation strategy handles a different part of the $Interpreter$ state: 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 then 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.

We describe each of these stages in a separate section. The first stage, which we call Elimination of Program Counter, is described in Section 5.2. The second stage, called Elimination of Frame Stack, is described in Section 5.3. Finally, third of the strategy, which is called Data Refinement of Objects, is described in Section 5.4.

5.2 Elimination of Program Counter

This stage eliminates $pc$ from the state of each thread’s process, $Thr(bc, cs, t)$, introducing the control flow constructs of C in the process. It may be summarised by the following theorem.

Theorem 5.2.1 (Elimination of Program Counter).

$$Thr(bc, cs, t) \subseteq ThrCF_{bc, cs}(cs, t)$$

In this stage we act mainly upon the Running action of $Thr$; its loop is unrolled to introduce the control flow that follows each bytecode instruction. The aim is to get each method’s bytecode instructions into a form in which the control flow, but not the data operations, are described using C constructs and, moreover, each path of execution (including every branch of the conditionals) ends in a return instruction or a loop. We refer to a method in this form as a complete method.

It is important to observe that it is possible to transform the bytecode instructions of every method so that they become complete. If we consider the control flow of a method beginning from that method’s entry point, each bytecode instruction reached must either be a return instruction, or followed by another bytecode. If another bytecode follows the bytecode’s execution, then it must be either a bytecode already considered, resulting in a loop, or one not already considered. Since there are finitely many bytecode instructions in a method, a loop or return must eventually be reached. Failure to do so would lead to an instruction beyond the end of the method, which is forbidden by the structural restrictions on Java bytecode [57]. We assume bytecode input to our strategy will have undergone bytecode verification so this cannot happen.

When a method is complete, it can be defined by a separate Circus action. When the code for all the methods has been split in this way, the choice of bytecode instruction using the program counter value can be removed and replaced with a choice over method identifiers. Thus dependency on the program counter can be completely removed, allowing it to be eliminated from the state of $Thr$.

The overall strategy for transforming $Thr$ in this stage and achieving this elimination is described by Algorithm 1. It begins at line 1 by expanding the Circus definitions of the bytecode instructions from the $bc$ map into the Running action, pulling out the program counter updates so that they can be more easily manipulated by the strategy. In line 2 instructions that are forward gotos or are simply followed by execution of the bytecode at the next $pc$ value are sequenced with the instructions following them. After that, for each method, its loops and conditionals are introduced in line 4. Afterwards, any complete methods are separated out, in line 5, and any method calls involving completed methods are resolved by sequencing the method call with the Circus action representing the method, in line 6.

This is repeated until all methods have been separated out, as indicated by the while loop in line 3. The $MainThread$ and $NotStarted$ actions are then refined in line 8 to provide a choice over method identifiers, rather than $pc$ values, thus removing all uses of $pc$ from the interpreter. The $pc$ component is then removed from the state in line 9 of the algorithm.

Each of the procedures used in Algorithm 1 is defined in a separate section in the sequel. Beforehand, we give a more detailed overview of the strategy.
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

```java
import java.io.InputStream;
import java.io.OutputStream;
import javax.realtime.AperiodicParameters;
import javax.realtime.ConfigurationParameters;
import javax.realtime.PriorityParameters;
import javax.safetycritical.AperiodicEventHandler;
import javax.safetycritical.StorageParameters;
import javax.safetycritical.io.ConsoleConnection;

public class TPK extends AperiodicEventHandler {

    public TPK(PriorityParameters priority,
                AperiodicParameters release,
                StorageParameters storage,
                ConfigurationParameters config) {
        super(priority, release, storage, config);
    }

    public void handleAsyncEvent() {
        ConsoleConnection console = new ConsoleConnection(null);
        InputStream input = console.openInputStream();
        OutputStream output = console.openOutputStream();
        for(int i = 0; i <= 10; i = i + 1) {
            int y = f(input.read());
            if (y > 400) {
                output.write(0);
            } else {
                output.write(y);
            }
        }
    }

    public static int f(int x){
        return x + x + x + 5;
    }
}
```

Figure 5.1: Our example program

5.2.1 Overview

We explain the strategy with an example, the Java code for which is shown in Figure 5.1. Our example is based on the Trabb Pardo-Knuth algorithm [48], used for comparison of programming languages, since it includes a variety of programming language constructs that provide a good test of the strategy. We have simplified the algorithm by removing the reading into an array, since our bytecode subset does not include array operations. Attempting to add arrays makes the example much longer, while not giving any interesting insight into our compilation strategy.
TPK : Class

TPK = {
  constantPool == {
    1 => ClassRef TPKClassID,
    3 => ClassRef AperiodicEventHandlerClassID,
    8 => MethodRef AperiodicEventHandlerClassID APEHinit,
    27 => ClassRef ConsoleConnectionClassID,
    29 => MethodRef ConsoleConnectionClassID CCinit,
    32 => MethodRef ConsoleConnectionClassID openInputStream,
    36 => MethodRef ConsoleConnectionClassID openOutputStream,
    40 => MethodRef InputStreamClassID read,
    41 => ClassRef InputStreamClassID,
    46 => MethodRef TPKClassID f,
    50 => MethodRef OutputStreamClassID write,
    51 => ClassRef OutputStreamClassID
  },
  this == 1,
  super == 3,
  interfaces == {},
  methodEntry == {
    f => 43,
    handleAsyncEvent => 7,
    APEHinit => 0,
  },
  methodEnd == {
    f => 50,
    handleAsyncEvent => 42,
    APEHinit => 6
  },
  methodLocals == {
    f => 1,
    handleAsyncEvent => 6,
    APEHinit => 5,
  },
  methodStackSize == {
    f => 2,
    handleAsyncEvent => 3,
    APEHinit => 5,
  },
  fields == {},
  staticFields == {}
}

cs : ClassID → Class

cs = {
  TPKClassID => TPK
}

bc : ProgramAddress → Bytecode

bc = {
  0 => aload 0,
  1 => aload 1,
  2 => aload 2,
  3 => aload 3,
  4 => aload 4,
  5 => invokespecial 8,
  6 => return,
  7 => new 27,
  8 => dup,
  9 => aconst_null,
  10 => invokevirtual 29,
  11 => astore 1,
  12 => aload 1,
  13 => invokevirtual 32,
  14 => astore 2,
  15 => aload 1,
  16 => invokevirtual 36,
  17 => astore 3,
  18 =>  iconst 0,
  19 => astore 4,
  20 => goto 19,
  21 => aload 2,
  22 => invokevirtual 40,
  23 => invokestatic 46,
  24 => astore 5,
  25 => aload 5,
  26 =>  iconst 400,
  27 => if _temp 5,
  28 => aload 3,
  29 =>  iconst 0,
  30 => invokevirtual 50,
  31 => goto 4,
  32 =>  iconst 400,
  33 => invokevirtual 50,
  34 => invokevirtual 50,
  35 =>  iconst 400,
  36 =>  iconst 1,
  37 =>  iadd,
  38 => astore 4,
  39 =>  iconst 400,
  40 =>  iconst 10,
  41 => if _temp (~ 20),
  42 => return,
  43 =>  iconst 0,
  44 =>  iconst 0,
  45 =>  iadd,
  46 =>  iconst 0,
  47 =>  iadd,
  48 =>  iconst 5,
  49 =>  iadd,
  50 => areturn
}

Figure 5.2: The Circus code corresponding to our example program
Running ≡
\[\text{if} \ frameStack = \emptyset \rightarrow \text{Skip} \]
\[\text{if} \ frameStack \neq \emptyset \rightarrow \]
\[\text{if} \ pc = 0 \rightarrow \text{HandleAloadEPC}(0) ; \ pc := 1 \]
\[\text{if} \ pc = 1 \rightarrow \text{HandleAloadEPC}(1) ; \ pc := 2 \]
\[\text{if} \ pc = 2 \rightarrow \text{HandleAloadEPC}(2) ; \ pc := 3 \]
\[\text{if} \ pc = 3 \rightarrow \text{HandleAloadEPC}(3) ; \ pc := 4 \]
\[\text{if} \ pc = 4 \rightarrow \text{HandleAloadEPC}(4) ; \ pc := 5 \]
\[\text{if} \ pc = 5 \rightarrow \text{HandleInvokespecialEPC}(8) \]
\[\text{if} \ pc = 6 \rightarrow \text{HandleReturnEPC} \]
\[\text{if} \ pc = 7 \rightarrow \text{HandleNewEPC}(27) ; \ pc := 8 \]
\[\text{if} \ pc = 8 \rightarrow \text{HandleDupEPC} ; \ pc := 9 \]
\[\text{if} \ pc = 9 \rightarrow \text{HandleAconst_nullEPC} ; \ pc := 10 \]
\[\text{if} \ pc = 10 \rightarrow \text{HandleInvokespecialEPC}(29) \]
\[\text{if} \ pc = 11 \rightarrow \text{HandleAstoreEPC}(1) ; \ pc := 12 \]
\[\ldots \]
\textbf{fi} ; \textbf{Poll} ; \textbf{Running} \textbf{fi}

Figure 5.3: The Running action after bytecode expansion

We have also written the example as an SCJ program, with the algorithm as the body of an aperiodic event handler, TPK, one or more instances of which can be registered as part of a mission and released during mission execution. As already mentioned, each release of the handler causes its handleAsyncEvent() method to be executed. This method creates an instance of a ConsoleConnection, which is the only standard input/output connection required by SCJ. Instances of InputStream and OutputStream are then obtained from the ConsoleConnection.

After the input and output streams have been obtained, we enter a for loop in which an integer is read from the InputStream, a static method f() is applied to it, and the result is output if it is less than 400, otherwise 0 is output. The method f() takes an integer as input, multiplies it by 3 and adds 5 to it.

The TPK class is part of a larger program that includes many other classes, including a Safelet, a MissionSequencer, a Mission, and the classes that make up the SCJ API. Considering these classes in our example would make the example much larger and more complex, while not introducing any more interesting aspects for the strategy to consider. We, therefore, omit a presentation of these classes, though it should be noted that they are part of the complete example.

Throughout the strategy we assume the extra classes have gone through similar processing to that which we illustrate for the TPK class. This adds little complexity to the strategy since the bytecode instructions the strategy acts upon are placed in a contiguous array that is acted upon consistently for all classes, and the current class of a given bytecode instruction can always be determined from its address in the array.

The Java code must be run through a Java compiler to generate the corresponding bytecode, which then defines the bc and cs constants of our model. The bc and cs values for our example are shown in Figure 5.2.

Applying the bytecode expansion on line 1 of Algorithm 1 yields the Running action shown in Figure 5.3. This step expands the bytecode instruction definitions, by copying HandleInstruction into Running, and converting it to a choice of actions based on the value of the program counter, mirroring the contents of the bc map for each value.

The actions that make up HandleInstruction are also replaced with actions that incorporate instruction parameters from the bc map and have pc updates separated from stack updates so they can be more easily operated on in this stage of the strategy. This can be seen in Figure 5.3 where, in the pc = 0 case,
Running \(\triangleq\)  

\[
\begin{align*}
\text{if } & \text{frameStack } = \emptyset \rightarrow \text{Skip} \\
\text{if } & \text{frameStack } \neq \emptyset \rightarrow \\
\text{if } & \text{pc } = 0 \rightarrow \text{HandleAloadEPC}(0); \text{ pc } := 1; \text{ Poll } \text{; HandleAloadEPC}(1); \\
\text{ pc } := 2; \text{ Poll } \text{; HandleAloadEPC}(2); \text{ pc } := 3; \text{ Poll } \text{; HandleAloadEPC}(4); \\
\text{ pc } := 5; \text{ Poll } \text{; HandleInvokeSpecialEPC}(8) \\
\ldots \\
\text{pc } &= 6 \rightarrow \text{HandleReturnEPC} \\
\text{pc } &= 7 \rightarrow \text{HandleNewEPC}(27); \text{ pc } := 8; \text{ Poll } \text{; HandleDupEPC } \text{; pc } := 9; \text{ Poll } \text{; HandleInvokeSpecialEPC}(29) \\
\ldots \\
\text{pc } &= 11 \rightarrow \text{HandleAstoreEPC}(1); \text{ pc } := 12; \text{ Poll } \text{; HandleAloadEPC}(1); \\
\text{ pc } := 13; \text{ Poll } \text{; HandleInvokeVirtualEPC}(32) \\
\ldots \\
\text{pc } &= 14 \rightarrow \text{HandleAstoreEPC}(2); \text{ pc } := 15; \text{ Poll } \text{; HandleAloadEPC}(1); \\
\text{ pc } := 16; \text{ Poll } \text{; HandleInvokeVirtualEPC}(36) \\
\ldots \\
\text{pc } &= 17 \rightarrow \text{HandleAstoreEPC}(3); \text{ pc } := 18; \text{ Poll } \text{; HandleIconstEPC}(0); \\
\text{ pc } := 19; \text{ Poll } \text{; HandleAstoreEPC}(4); \text{ pc } := 20; \text{ Poll } \text{; pc } := 39 \\
\ldots \\
\text{pc } &= 39 \rightarrow \text{HandleAloadEPC}(4); \text{ pc } := 40; \text{ Poll } \text{; HandleIconstEPC}(10); \\
\text{ pc } := 41; \text{ Poll } \text{; var value1, value2 : Word } \bullet \text{ InterpreterPop2}; \\
\text{pc } := \text{if value1 } \leq \text{ value2 then 21 else 42} \\
\ldots \\
\text{pc } &= 42 \rightarrow \text{HandleReturnEPC} \\
\text{pc } &= 43 \rightarrow \text{HandleAloadEPC}(0); \text{ pc } := 44; \text{ Poll } \text{; HandleAloadEPC}(0); \\
\text{ pc } := 45; \text{ Poll } \text{; HandleIaddEPC}; \text{ pc } := 46; \text{ Poll } \text{; HandleAloadEPC}(0); \\
\text{pc } &= 47; \text{ Poll } \text{; HandleIaddEPC}; \text{ pc } := 48; \text{ Poll } \text{; HandleIaddEPC}; \\
\text{pc } &= 50; \text{ Poll } \text{; HandleAreturnEPC} \\
\ldots \\
\text{fi}; \text{ Poll } \text{; Running} \\
\text{fi}
\end{align*}
\]

Figure 5.4: The Running action after forward sequence introduction

aload 0 has been converted to HandleAloadEPC(0); pc := 1, with the parameter, 0, to the bytecode instruction becoming a parameter of the new instruction handling action HandleAloadEPC, and the update to pc placed after the data operation.

The reason for making parameters of the bytecode instructions into parameters of the handling actions is to remove the need to reference the bytecode instructions in the bc map, as that involves use of the pc value, which we seek to remove in this stage. This also has the benefit of fully incorporating bc into the Thr process, ensuring all the information required to introduce C code constructs is available directly in Circus, which makes stating compilation laws simpler. This is described in more detail in Section 5.2.2 where we define the ExpandBytecode procedure.

On line 2 of the algorithm, sequential composition is introduced for instructions that do not affect the sequential flow of the program. Such instructions are identified by considering the control flow graph of the program and locating nodes with a single outgoing edge going to target node exactly one incoming edge. The introduction of sequential composition is performed by unrolling the loop in Running to introduce the control flow following each of these instructions. This causes the instruction to be sequentially composed with the next instruction, with Poll in between to allow for thread switches between instructions. This is performed exhaustively to get the code in the form shown in Figure 5.4 where the choice over pc has sequences of instructions collected together at the point where they start.
up to the point at which a more complex control flow (such as a method call, conditional or a loop) occurs. The introduction of sequential composition is described in more detail in Section 5.2.3 where we define the `IntroduceSequentialComposition` procedure.

Handling the remaining constructs requires consideration of dependency between methods to ensure method calls can be resolved correctly. We say a method call is resolved when the method invocation bytecode has been placed in sequential composition with a call to a Circus action containing the body of the method being invoked, which is then followed by the sequence of instructions that occur after the invocation bytecode instruction in the calling method. After a method call has been resolved, it no longer breaks up the sequence of instructions it occurs in.

Since we have the bytecode instructions of all the methods needed, we can always resolve the call of a complete method, provided that method has already been split into its own Circus action. To ensure the method that a method call depends on is complete, we first perform loop and conditional introduction upon it. Since introducing loops and conditionals requires unbroken sequences of instructions that form the bodies of loops and branches of conditionals, introduction of loops and conditionals can only be performed on methods that have no unresolved method calls. For this reason, we perform method call resolution and loop and conditional introduction repeatedly until all method calls are resolved and the resulting complete methods have all been separated out. This is expressed in Algorithm 1 by the while loop on line 3.

Introduction of loops and conditionals to the body of a method with no unresolved method calls occurs on line 4 of the algorithm. To introduce loops and conditionals we consider the control flow graph of the method again, though it is now much simpler than the control flow graph used for sequence introduction since straight sequences of instructions have already been combined together. Patterns representing conditionals and loops are then identified using the control flow graph and the corresponding constructs are introduced. As loops and conditionals are introduced, nodes in the control flow graph are merged until the graph consists of a single node, which is the starting point of the method, containing the complete method body.

In our example, `handleAsyncEvent()` is the only method that needs loops and conditionals introducing but, since it also contains method calls that break up the body of a loop, we must wait until its method calls have been resolved before introducing loops and conditionals. The result of introducing loops and conditionals after method calls have been resolved in `handleAsyncEvent()` is shown in Figure 5.5. The process of introducing loops and conditionals is described in more detail in Section 5.2.4 where we define the `IntroduceLoopsAndConditionals` procedure.

After loops and conditionals have been introduced, methods that are then complete can be copied into separate actions. This occurs in line 5 of this algorithm. This is done with a simple application of the copy rule, replacing the actions at the entry points of the split methods with references to newly created method actions. This can be seen in Figure 5.6 where the `TPK_F` action has been created by splitting the sequence of actions for the `f()` method of `TPK` from the `pc = 43` case. As this step is relatively simple, we do not explain it in a separate section.

Calls to those methods can then be resolved, sequencing the method invocation instruction with a call to the Circus action representing its body and the instructions following the method call. This occurs on line 6 of the algorithm, and can be seen in Figure 5.6 which shows our example after method call resolution has been applied.

The target of each method call can be determined from the parameter to the method invocation instruction. This parameter is an index into the constant pool of the current class that points to a method reference for the method being called. The correct current class for each bytecode instruction is always known, since the information on the method entries and ends is contained in the class information, and there is a one-to-one mapping between classes and blocks of bytecode instructions that form methods. After the target of the method call has been determined, the invocation instruction can be sequenced with a call to the corresponding Circus action.

An example of this is the occurrence of `HandleInvokestaticEPC(46)` in the sequence of actions at `pc = 21`. As can be seen from Figure 5.2 the constant pool index 46 corresponds to the method identifier for the method `f()` of the TPK. The sequence of instructions corresponding to this method is in an action `TPK_F`,
Running \(\overset{\circ}{=}\)

\[
\begin{align*}
    \text{if } \text{frameStack} = \emptyset & \rightarrow \text{Skip} \\
    \text{frameStack} \neq \emptyset & \rightarrow \\
    \text{if } \text{pc} = 0 & \rightarrow \text{HandleAloadEPC}(0); \text{ pc } := 1; \ Poll; \ HandleAloadEPC(1); \\
    \text{pc} := 2; \ Poll; \ HandleAloadEPC(2); \text{ pc } := 3; \ Poll; \ HandleAloadEPC(3); \\
    \text{pc} := 4; \ Poll; \ HandleAloadEPC(4); \text{ pc } := 5; \ Poll; \\
    \text{HandleInvokeSpecialEPC}(8); \ Poll; \ AperiodicEventHandler_{APEHInit}; \\
    \text{Poll; HandleReturnEPC} \\
    \ldots
\end{align*}
\]

\[
\begin{align*}
    \text{pc} = 7 & \rightarrow \text{HandleNewEPC}(27); \text{ pc } := 8; \ Poll; \ HandleDupEPC; \text{ pc } := 9; \\
    \ldots
\end{align*}
\]

\[
\begin{align*}
    \text{Poll; pc } := 39; \ Poll; \ \mu Y \bullet \\
    \text{HandleAloadEPC}(4); \text{ pc } := 40; \ Poll; \ HandleIconstEPC(10); \text{ pc } := 41; \\
    \text{Poll; var value1, value2 : Word \bullet InterpreterPop2; pc } := \text{if value1 } \leq \text{ value2 then 21 else 42}; \ Poll; \\
    \text{if value1 } \leq \text{ value2 } & \rightarrow \text{HandleAloadEPC}(2); \text{ pc } := 22; \ Poll; \\
    \ldots
\end{align*}
\]

\[
\begin{align*}
    \text{HandleIconstEPC}(400); \text{ pc } := 27; \ Poll; \ \text{var value1, value2 : Word \bullet InterpreterPop2; pc } := \text{if value1 } \leq \text{ value2 then 32 else 28}; \ Poll; \\
    \text{if value1 } \leq \text{ value2 } & \rightarrow \text{HandleAloadEPC}(3); \text{ pc } := 33; \ Poll; \\
    \text{HandleAloadEPC}(5); \text{ pc } := 34; \ Poll; \ HandleInvokeVirtualEPC(50); \\
    \text{Poll; ConsoleOutput_Write \ [ value1 > value2 } & \rightarrow \text{HandleAloadEPC}(3); \text{ pc } := 29; \ Poll; \\
    \text{HandleIconstEPC}(0); \text{ pc } := 30; \ Poll; \ HandleInvokeVirtualEPC(50); \\
    \text{Poll; ConsoleOutput_Write fi; pc } := 35; \ Poll; \ HandleAloadEPC(4); \text{ pc } := 36; \ Poll; \\
    \text{HandleIconstEPC}(1); \text{ pc } := 37; \ Poll; \ HandleIaddEPC; \text{ pc } := 38; \\
    \text{Poll; HandleAstoreEPC(4); pc } := 39; \ Poll; \ Y \\
    \text{value1 } > \text{ value2 } & \rightarrow \text{HandleAloadEPC(4); pc } := 39; \ Poll; \ Y \\
    \text{value1 } > \text{ value2 } & \rightarrow \text{HandleReturnEPC fi} \\
    \ldots
\end{align*}
\]

\[
\begin{align*}
    \text{pc } := 43 & \rightarrow \text{TPK}_F \\
    \ldots
\end{align*}
\]

\[
\begin{align*}
    \text{fi; Poll; Running fi}
\end{align*}
\]

Figure 5.5: The Running action after loop and conditional introduction

created in the previous step, on line 5. This action is sequenced with the invocation instruction, with the Poll action inbetween (to allow thread switches before the first instruction of the called method). The instructions following the method call are then sequenced after it, with another Poll action (to allow thread switches following the return instruction). Method call resolution is described in more detail in Section 5.2.5, where we define the SEPARATECOMPLETEMETHODS and RESOLVEMETHODCALLS procedures.

As mentioned previously, these steps are then repeated, in the loop beginning at line 3 to introduce the loops and conditionals in methods that would otherwise have unresolved method calls in the middle of loops and conditionals. Afterwards, those methods can be separated out and this loop, conditional and method resolution repeated until every method has been separated out in this way.

We do not allow recursion to ensure this terminates, since this requires all the methods called by a given method to be resolved before the method itself can be resolved. This is a sensible restriction since recursion is not normally allowed in safety-critical applications because of the potential for unpredictable failure due to stack overflow.
The Running action of our example at the end of the loop, when all loops and conditionals have been introduced, all the methods have been separated out, and all method calls have been resolved, is shown in Figure 5.7. At this point, the choice over the pc value maps entry points of methods onto the actions representing those methods, with the other pc values now redundant.

The next step is then to eliminate these redundant paths and remove the dependency on pc to select the method action. This occurs at line 8 of the algorithm, in which the NotStarted andMainThread actions are refined to replace the Running action with an ExecuteMethod action that contains a choice of method action based on the method and class identifier of the method. This can be seen in Figure 5.8, which shows the ExecuteMethod action corresponding to our example, and the refined NotStarted andMainThread actions that reference it. We describe this refinement in more detail in Section 5.2.6, where we define the RefineMainActions procedure.

When all of the previous steps are completed, reliance on pc to determine control flow has been completely removed. The pc state component can then be removed in a simple data refinement that also removes all the assignments to pc, resulting in the HandleAsyncEvent action shown in Figure 5.9.

The remaining instruction handling actions then only affect the stack, the removal of which is the concern of the next stage of the compilation strategy. The data refinement to remove pc is applied at the end of the algorithm, on line 9, and is described in more detail in Section 5.2.7, where we define the RemovePCFromState procedure.

We now proceed to describe each of the steps of program counter elimination in more detail.
Running ⇑
\[\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \]
\[\text{if } \text{frameStack} \neq \emptyset \rightarrow \]
\[\text{if } \text{pc} = 0 \rightarrow \text{TPK\_APEHInit} \]
\[\ldots \]
\[\text{if } \text{pc} = 7 \rightarrow \text{TPK\_HandleAsyncEvent} \]
\[\ldots \]
\[\text{if } \text{pc} = 43 \rightarrow F \]
\[\ldots \]
\[\text{fi}; \text{Poll}; \text{Running} \]
\[\text{fi} \]

\text{TPK\_APEHInit} \equiv \text{HandlealoadEPC}(0); \text{pc} := 1; \text{Poll}; \text{HandlealoadEPC}(1); \text{pc} := 2; \text{Poll}; \text{HandlealoadEPC}(2); \text{pc} := 3; \text{Poll}; \text{HandlealoadEPC}(3); \text{pc} := 4; \text{Poll}; \text{HandlealoadEPC}(4); \text{pc} := 5; \text{Poll}; \text{HandleInvokeSpecialEPC}(8); \text{Poll}; \text{AperiodicEventHandler\_APEHInit}; \text{Poll}; \text{HandleReturnEPC} \]

\text{TPK\_HandleAsyncEvent} \equiv \text{HandleNewEPC}(27); \text{pc} := 8; \text{Poll}; \text{HandleDupEPC}; \ldots \]
\[\text{Poll}; \text{pc} := 39; \text{Poll}; \mu Y \bullet \]
\[\text{HandlealoadEPC}(4); \text{pc} := 40; \text{Poll}; \text{HandleIconstEPC}(10); \text{pc} := 41; \text{Poll}; \text{var value1, value2 : Word \bullet InterpreterPop2}; \]
\[\text{pc} := \text{if } \text{value1} \leq \text{value2} \text{ then 21 else 42}; \text{Poll}; \]
\[\text{if } \text{value1} \leq \text{value2} \rightarrow \text{HandlealoadEPC}(2); \text{pc} := 22; \text{Poll}; \]
\[\ldots \]
\[\text{HandleIconstEPC}(400); \text{pc} := 27; \text{Poll}; \text{var value1, value2 : Word \bullet InterpreterPop2}; \]
\[\text{pc} := \text{if } \text{value1} \leq \text{value2} \text{ then 32 else 28}; \text{Poll}; \]
\[\text{if } \text{value1} \leq \text{value2} \rightarrow \text{HandlealoadEPC}(3); \text{pc} := 33; \text{Poll}; \]
\[\text{HandlealoadEPC}(5); \text{pc} := 34; \text{Poll}; \text{HandleInvokevirtualEPC}(50); \text{Poll}; \text{ConsoleOutput\_Write} \]
\[\text{if } \text{value1} > \text{value2} \rightarrow \text{HandlealoadEPC}(3); \text{pc} := 29; \text{Poll}; \]
\[\text{HandleIconstEPC}(0); \text{pc} := 30; \text{Poll}; \text{HandleInvokevirtualEPC}(50); \text{Poll}; \text{ConsoleOutput\_Write} \]
\[\text{fi}; \text{pc} := 35; \text{Poll}; \text{HandlealoadEPC}(4); \text{pc} := 36; \text{Poll}; \]
\[\text{HandleIconstEPC}(1); \text{pc} := 37; \text{Poll}; \text{HandleAddEPC}; \text{pc} := 38; \text{Poll}; \text{HandleAstoreEPC}(4); \text{pc} := 39; \text{Poll}; Y \]
\[\text{value1} > \text{value2} \rightarrow \text{HandleReturnEPC} \]
\[\text{fi} \]

Figure 5.7: The Running action after all the methods are separated
ExecuteMethod \equiv \texttt{val} cid : \texttt{ClassID}; mid : \texttt{MethodID} \\
\texttt{if}(cid, mid) = (\texttt{TPKClassID}, \texttt{APEHInit}) \rightarrow \texttt{TPK}_A\texttt{PEHInit} \\
\texttt{if}(cid, mid) = (\texttt{TPKClassID}, \texttt{handleAsyncEvent}) \rightarrow \texttt{TPK}_A\texttt{HandleAsyncEvent} \\
\texttt{if}(cid, mid) = (\texttt{TPKClassID}, f) \rightarrow \texttt{TPK}_F \\
\texttt{fi}

MainThread \equiv \\
\texttt{(var} cid : \texttt{ClassID}; method : \texttt{MethodID}; methodArgs : \texttt{seq Word} \triangleright \texttt{interpreter}?c?m?a \rightarrow cid, method, methodArgs := c, m, a; \\
(\exists \texttt{class}? == cs cid; baseFrame? == True \triangleright \texttt{InterpreterNewStackFrame}); \\
\texttt{ExecuteMethod}(cid, method) ; \texttt{MainThread} \\
\square

CEEswitchThread?from?to : (from = main) \rightarrow \texttt{Blocked}; \texttt{MainThread}

NotStarted \equiv \texttt{var} cid : \texttt{ClassID}; method : \texttt{MethodID}; methodArgs : \texttt{seq Word} \triangleright \\
CEEstartThread?toStart?bsid?c?m?a : (toStart = thread) \rightarrow \\
\texttt{cid, method, methodArgs := c, m, a}; \\
(\exists \texttt{class}? == cs cid; baseFrame? == True \triangleright \texttt{InterpreterNewStackFrame}); \\
\texttt{Blocked}; \texttt{ExecuteMethod}(cid, method) ; \texttt{CEEremoveThread!thread} \rightarrow \texttt{NotStarted}

Figure 5.8: The \texttt{ExecuteMethod}, \texttt{NotStarted}, and \texttt{MainThread} actions after main action refinement
HandleAsyncEvent = HandleNewEPC(27); Poll; HandleDupEPC;
Poll; HandleAconst_nullEPC; Poll; HandleInvokespecialEPC(29); Poll; CCInit; Poll; HandleAstoreEPC(1); Poll;
HandleAloadEPC(1); Poll; HandleInvokevirtualEPC(32); Poll;
OpenInputStream; Poll; HandleAstoreEPC(2); Poll;
HandleAloadEPC(1); Poll; HandleInvokevirtualEPC(36); Poll;
OpenOutputStream; Poll; HandleAstoreEPC(3); Poll;
HandleIconstEPC(0); Poll; HandleAstoreEPC(4);
Poll; Poll; HandleAloadEPC(4); Poll;
HandleIconstEPC(10); Poll; \textbf{var} value1, value2 : Word ●
InterpreterPop2; Poll; µ Y ●
\textbf{if} value1 ≤ value2 \rightarrow \text{HandleAloadEPC}(2); Poll;
HandleInvokevirtualEPC(40); Poll; Read; Poll;
HandleInvokestaticEPC(46); Poll; F; Poll;
HandleAstoreEPC(5); Poll; HandleAloadEPC(5);
HandleIconstEPC(400); Poll; \textbf{var} value1, value2 : Word ●
InterpreterPop2; Poll;
\textbf{if} value1 ≤ value2 \rightarrow \text{HandleAloadEPC}(3); Poll;
HandleAloadEPC(5); Poll; HandleInvokevirtualEPC(50);
Poll; Write; Poll; HandleAloadEPC(4); Poll;
HandleIconstEPC(1); Poll; HandleIaddEPC; Poll;
HandleAstoreEPC(4); Poll; HandleAloadEPC(4); Poll;
HandleIconstEPC(10); Poll; \textbf{var} value1, value2 : Word ●
InterpreterPop2; Poll;
\textbf{if} value1 ≤ value2 \rightarrow \text{Y}

\[ \text{value1} > \text{value2} \rightarrow \text{HandleReturnEPC} \]
\textbf{fi}

\[ \text{value1} > \text{value2} \rightarrow \text{HandleReturnEPC} \]
\textbf{fi}

\[ \text{value1} > \text{value2} \rightarrow \text{HandleReturnEPC} \]
\textbf{fi}

Figure 5.9: The \textit{HandleAsyncEvent} action after \textit{pc} has been eliminated from the state
5.2.2 Expand Bytecode

Before the control flow can be introduced, the bytecode instructions provided in the \textit{bc} parameter to \textit{Thr} must be expanded to allow consideration of their semantics. This is achieved using the procedure shown in Algorithm 2. This begins on line 2 by introducing a choice over all the possible values of \textit{pc}.

\begin{algorithm}
\caption{Expand Bytecode}
\begin{algorithmic}[1]
\Procedure{ExpandBytecode}{ }
\State \textbf{IntroduceChoiceOverPC}
\For {$pc \leftarrow \text{dom bc}$}
\State \textbf{CollapseHandleInstruction}
\EndFor
\EndProcedure
\end{algorithmic}
\end{algorithm}

associated with the \textit{HandleInstruction} action in \textit{Running}, replacing \textit{HandleInstruction} with a choice of the form shown below.

\begin{verbatim}
if \(pc = 0\) \(\rightarrow\) \textit{HandleInstruction}
\(pc = 1\) \(\rightarrow\) \textit{HandleInstruction}
\(pc = 2\) \(\rightarrow\) \textit{HandleInstruction}
\ldots
fi
\end{verbatim}

After that, we operate on the occurrence of \textit{HandleInstruction} at each \textit{pc} value, replacing it with its definition, which is a choice between actions to handle each type of instruction (e.g. \textit{HandleDup}, \textit{HandleAload} etc.). For brevity, we refer to these handling actions that define \textit{HandleInstruction} as \textit{Handle*} actions. Since the value of \textit{bc} at a given \textit{pc} value is known, we can determine which of the \textit{Handle*} actions is chosen for each occurrence of \textit{HandleInstruction}. The occurrences of \textit{HandleInstruction} are therefore collapsed to the appropriate \textit{Handle*} actions on line 4. This produces the following choice for our example.

\begin{verbatim}
if \(pc = 0\) \(\rightarrow\) \textit{HandleAload}
\(pc = 1\) \(\rightarrow\) \textit{HandleAload}
\(pc = 2\) \(\rightarrow\) \textit{HandleAload}
\ldots
\(pc = 7\) \(\rightarrow\) \textit{HandleNew}
\(pc = 8\) \(\rightarrow\) \textit{HandleDup}
\(pc = 9\) \(\rightarrow\) \textit{HandleAconst\_null}
\ldots
fi
\end{verbatim}

The \textit{Handle*} actions are then replaced with new actions. Those new actions are not guarded on the value of \textit{bc} at the current \textit{pc} value, since the choice those guards mediate has already been collapsed. The parameters of the bytecode instructions are also transferred to become parameters of the new actions. Finally, the updates to \textit{pc} contained in the \textit{Handle*} actions are extracted in the form of assignments to \textit{pc}.

This final transformation is not carried out in the case of the method invocation and return instructions, where the \textit{pc} updates are closely connected to the operations on the stack and require special handling. The new actions’ names are formed by appending \textit{EPC} to the names of the \textit{Handle*} actions. The overall mapping from bytecode instructions to the new actions is shown in Table 5.1. Note that the \textit{Handle*} actions are eliminated completely in the case of the \textbf{goto} and \textbf{if\_icmp\_e} instructions, since the \textit{pc} update is the main effect of these actions.

This replacement of the \textit{Handle*} actions with these new actions occurs in line 5. After this transformation has been applied, the \textit{Running} action for our example is as shown in Figure 5.3.
<table>
<thead>
<tr>
<th>Bytecode (bc i)</th>
<th>Action (handleAction(bc i))</th>
</tr>
</thead>
<tbody>
<tr>
<td>aconst_null</td>
<td>HandleAconst_nullEPC; pc := i + 1</td>
</tr>
<tr>
<td>dup</td>
<td>HandleDupEPC; pc := i + 1</td>
</tr>
<tr>
<td>aload lvi</td>
<td>HandleAloadEPC(lvi); pc := i + 1</td>
</tr>
<tr>
<td>astore lvi</td>
<td>HandleAstoreEPC(lvi); pc := i + 1</td>
</tr>
<tr>
<td>iadd</td>
<td>HandleIaddEPC; pc := i + 1</td>
</tr>
<tr>
<td>iconst n</td>
<td>HandleIconstEPC(n); pc := i + 1</td>
</tr>
<tr>
<td>ineg</td>
<td>HandleInegEPC; pc := i + 1</td>
</tr>
<tr>
<td>goto ofst</td>
<td>pc := i + ofst</td>
</tr>
<tr>
<td>if_{icmple}</td>
<td>var value1, value2 : Word ● InterpreterPop2; if value1 ≤ value2 then i + ofst else i + 1</td>
</tr>
<tr>
<td>areturn</td>
<td>HandleAreturnEPC</td>
</tr>
<tr>
<td>return</td>
<td>HandleReturnEPC</td>
</tr>
<tr>
<td>getfield cpi</td>
<td>HandleGetfieldEPC(cpi); pc := i + 1</td>
</tr>
<tr>
<td>putfield cpi</td>
<td>HandlePutfieldEPC(cpi); pc := i + 1</td>
</tr>
<tr>
<td>getstatic cpi</td>
<td>HandleGetstaticEPC(cpi); pc := i + 1</td>
</tr>
<tr>
<td>putstatic cpi</td>
<td>HandlePutstaticEPC(cpi); pc := i + 1</td>
</tr>
<tr>
<td>invokevirtual cpi</td>
<td>pc := i ; HandleInvokevirtualEPC(cpi)</td>
</tr>
<tr>
<td>invokespecial cpi</td>
<td>pc := i ; HandleInvokespecialEPC(cpi)</td>
</tr>
<tr>
<td>invokestatic cpi</td>
<td>pc := i ; HandleInvokestaticEPC(cpi)</td>
</tr>
</tbody>
</table>

Table 5.1: The syntactic function handleAction

The overall transformation of the HandleInstruction action in this step is summarised by \( \# \) which makes use of a syntactic function handleAction that maps bytecode instructions onto Circus actions as shown in Table 5.1.

**Rule** [Bytecode Expansion]. For a given bc

\[
\text{HandleInstruction}_{bc} \subseteq A \text{ if } \#_{i, pc = i} \rightarrow \text{handleAction}(bc i) \#
\]

where handleAction is a syntactic function defined by Table 5.1.

After the bytecode semantics is expanded in the Running action, the control flow that corresponds to each pc update can be introduced.

### 5.2.3 Introduce Sequential Composition

**Algorithm 3** Introduce Sequential Composition

1: **procedure** INTRODUcesequentialcomposition
2: \( cfg \leftarrow \text{MAKECONTROLFLOWGRAPH} \)
3: **for** node \( \leftarrow \) \( cfg \) **do**
4: \( \text{while HASSIMPLESEQUENCE}(\text{node}) \) **do**
5: \( \text{APPLY}(\text{node}) \)
6: **end while**
7: **end for**
8: **end procedure**

The simplest control flow to introduce is that of instructions where execution continues at the next program counter value. These control flows are introduced as shown in Algorithm 3. The algorithm consists of constructing a control flow graph for each method in the program, as specified on line 2. Since the introduction of sequential composition does not depend on the relationships between methods, the control flow graph is constructed as a disconnected control flow graph containing the control flow of all the methods in the program. Although method calls have not had their pc updates formally made
explicit, we can assume a method call will be followed by the instruction at the next program counter
value. The control flow graph for our example is shown in Figure 5.10.

After the control flow graph is constructed, we consider each node in turn, as specified by the for loop
on line 3. As mentioned earlier, we require a node to have only a single outgoing edge and its target to
have only a single incoming edge in order to be considered for the introduction of sequential composition.
The reason for this is that nodes with two outgoing edges are points at which conditionals should be
introduced, rather than sequential compositions. Such nodes in our example are the nodes for pc values
27 and 41, which represent the start of conditionals. Likewise, nodes with multiple incoming edges
represent points at which a more complex control flows occur. For our example, such nodes include 39,
which is the start of a loop, and 35, which is the end of a conditional. These prevent introduction of
sequential composition for the pc values 20, 31, 34, and 38, since those the targets of those nodes are
nodes 35 and 39.

For a node that meets the above requirement and isn’t a method call, we can introduce sequential
composition at that node by applying \( \text{seq} \) on line 5 of the algorithm.

**Rule** [Sequence introduction]. If \( i \neq j \) and

\[
\{ \text{frameStack} \neq \emptyset \} ; A = \{ \text{frameStack} \neq \emptyset \} ; A ; \{ \text{frameStack} \neq \emptyset \}
\]

then,

\[
\mu X \bullet \begin{cases}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \\
\quad \text{if } \ldots \\
\quad \text{if } \text{pc} = i \rightarrow A ; \text{pc} := j \quad \subseteq A \\
\quad \text{if } \ldots \\
\quad \text{if } \text{pc} = j \rightarrow B \\
\quad \ldots \\
\quad \text{fi} ; \text{Poll} ; X \\
\text{fi}
\end{cases}
\]

\[
\mu X \bullet \begin{cases}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \\
\quad \text{if } \ldots \\
\quad \text{if } \text{pc} = i \rightarrow A ; \text{pc} := j ; \text{Poll} ; B \\
\quad \ldots \\
\quad \text{if } \text{pc} = j \rightarrow B \\
\quad \ldots \\
\quad \text{fi} ; \text{Poll} ; X \\
\text{fi}
\end{cases}
\]

This rule works by unrolling the loop in *Running* to sequence an instruction with the instruction that
is executed after it, inserting Poll inbetween. It is required that the pc value of the node’s target, \( j \), not
be the same as the pc value of the node, \( i \), since that would introduce a loop, rather than a sequential

Figure 5.10: Control flow graph for our example program
composition. Also, the sequence of instructions at the node, A, must not affect the non-emptiness of the frameStack to ensure that the choice at the start of the main loop in Running can be resolved.

Since pulls two nodes together, we can continue to introduce sequential composition at a node after the first application of until that node no longer satisfies the conditions for introducing sequential composition. This is specified by the while loop at line of the algorithm. This means the control flow graph is updated as is applied, to take into account the merging of nodes. The resulting control flow graph after introduction of sequential composition has been performed at every point is shown in Figure 5.11. The only remaining nodes in this graph are those where the sequence of instructions ends with a method call or which represent a more complex control flow. In particular, the instructions for the f() method of TPK, which begin at pc = 43, have been completely sequenced together into a single node. The code which corresponds to this control flow graph is that shown earlier in Figure 5.4.

5.2.4 Introduce Loops and Conditionals

After sequential composition has been introduced for all methods, we must begin considering each method separately to ensure method calls are handled properly. This means the strategy must loop, introducing loops and conditionals to those methods that have no unresolved method calls and resolving calls of methods that are then complete, until every method is complete and has been separated into its own action. Introducing loops and conditionals is performed as described by Algorithm 4. This considers each method individually, as specified by the for loop on line of the algorithm. The condition on line ensures that only those methods where all method calls have already been resolved undergo loop and conditional introduction.

For each method that undergoes loop and conditional introduction, we must again consider the control flow graph of the method to ensure the loops and conditionals are introduced in the correct order to properly form the bodies of loops and conditionals. This involves constructing a control flow graph for the method, at line beginning at the entry point of the method and following each goto and if icmple instruction until a loop is detected or a return or areturn instruction is reached. The graph for the our example, beginning at pc = 7 (the entry point of the handleAsyncEvent() method), is shown in Figure 5.12 alongside the Circus code obtained at the beginning of this stage for the method. The edge which forms a loop from pc = 35 to pc = 39 is shown as a dashed line since looping edges are ignored at certain points in this part of the strategy.

A method’s control flow graph must be well-structured in order to properly introduce the control flow structures in this section. We define a rooted directed graph below. The definition is standard, but we include it here to introduce the terminology for the subsequent definition of what we mean by a structured control flow graph.

Definition 5.2.2 (Rooted Directed Graph). A rooted directed graph, \( G \), is a triple \( (V, E, r) \), where
Algorithm 4 Introduce Loops and Conditionals

1: procedure IntroduceLoopsAndConditionals
2:     for \( m \leftarrow \text{methods} \) do
3:         if \( \text{HasNoUnresolvedCalls}(m) \) then
4:             \( \text{cfg} \leftarrow \text{MakeControlFlowGraph}(m) \)
5:             for \( \text{node} \leftarrow \text{ReverseNodes}(\text{cfg}) \) do
6:                 \( \text{Apply}(\text{node}) \)
7:                 \( \text{Apply}(\text{node}) \)
8:                 if \( \text{IsSimpleConditional}(\text{node}) \) then
9:                     \( \text{Apply}(\text{node}) \)
10: end if
11: \( \text{Apply}(\text{node}) \)
12: \( \text{Apply}(\text{node}) \)
13: \( \text{Apply}(\text{node}) \)
14: \( \text{Apply}(\text{node}) \)
15: if \( \text{HasSimpleSequence}(\text{node}) \) then
16:     \( \text{Apply}(\text{node}) \)
17: end if
18: end for
19: end if
20: end for
21: end procedure

Figure 5.12: Simplified control flow graph and corresponding code for our example program
• $V$ is a set of nodes,
• $E$ is a set of ordered pairs of nodes in $V$, called edges, and
• $r$ is a node in $V$, called the root of the graph.

The first component of an edge is its source and the second component is its target. We say that an edge goes from its source to its target. The source of an edge going to a given node is said to be a predecessor of that node; similarly, the target of an edge from a given node is a successor of that node. For every node $n \in V$, the pair $(r, n)$ must be in the reflexive transitive closure of $E$, that is, there must be a path of edges from the root to any node in the graph.

In diagrams we represent the nodes as points or as the names of the nodes, the edges as arrows, and the root node as a node with an arrow pointing to it that does not come from another node. For a graph $G$, we refer to the set $T(G) = \{ n \in V \mid \forall m \in V. (n, m) \notin E \}$ of nodes with no edges coming from them as the set of end nodes of the graph.

Having defined a rooted directed graph, we now define what we mean by a structured control flow graph as a specific type of rooted directed graph. This definition we use for a structured program is based on Dijkstra’s notion of program structure found in [26]. In that definition, there are a set of known program structures that are permitted, and these structures may contain further occurrences of the same structures (e.g. a conditional in which each branch is also a conditional). In order to formalise this, we first define what it means to replace a node with a graph.

**Definition 5.2.3** (Node Replacement). Given two rooted directed graphs $G$ and $H$, we say $G'$ is the graph formed by replacing a node $n$ of $G$ with $H$ if one of the following cases holds:

• $n$ has no predecessors in $G$, $H$ has only one end node, and
  - $G'$ contains all the nodes of $H$ and $G$, except $n$,
  - $G'$ contains the edges of $G$ except those going to or from $n$,
  - $G'$ contains edges from the end node of $H$ to the successors of $n$ in $G$, and
  - the root node of $H$ is the root node of $G'$;

• $n$ has no successors in $G$, and
  - $G'$ contains all the nodes of $H$ and $G$, except $n$,
  - $G'$ contains the edges of $G$ except those going to or from $n$,
  - $G'$ contains edges from the predecessors $n$ in $G$ to the root node of $H$, and
  - the root node of $G$ is the root node of $G'$;

• $H$ has a single end node and
  - $G'$ contains all the nodes of $H$ and $G$, except $n$,
  - $G'$ contains the edges of $G$ and the edges of $H$ except those going to or from $n$,
  - $G'$ contains edges from the predecessors of $n$ in $G$ to the root node of $H$,
  - $G'$ contains edges from the end node of $H$ to the successors of $n$ in $G$, and
  - the root node of $G$ is the root node of $G'$;

• $n$ has a single successor in $G$, $H$ has a single end node, and
  - $G'$ contains all the nodes of $H$ and $G$, except $n$ and the end node of $H$,
  - $G'$ contains the edges of $G$ except those going to or from $n$,
  - $G'$ contains edges from the predecessors of the end node of $H$ to the successor of $n$ in $G$
  - $G'$ contains edges from the predecessors of $n$ in $G$ to the root node of $H$, and
  - the root node of $G$ is the root node of $G'$.
Each of the different cases of node replacement represents a different way of placing a graph inside another graph. We show an example of each of these cases in Figure 5.13. The example used is that of an \texttt{if-else} conditional, introduced later in Figure 5.14, with one of its nodes replaced with another \texttt{if-else} conditional whose nodes are shown in white.

The first case (Figure 5.13a) is that of placing a graph at the start of another graph, i.e. replacing the root node of a graph that does not have a loop to its root node. The second case (Figure 5.13b) is that of replacing one of the end nodes of a graph. The third case (Figure 5.13c) is that of replacing an internal node of the graph. There must be a single end node in this case in order to have a source for the outgoing edges of the replaced node. At the end of one of the branches of a conditional, the end node of the replacing graph may be unified with the node at the end of the conditional. This represents such cases as loops and conditionals occurring at the end of a branch of a conditional, with no instructions following them inside the conditional, and is handled by the fourth case of node replacement, shown in Figure 5.13d.

With node replacement defined, we can now define what we mean by a structure control flow graph in terms of node replacement and the structured graphs shown in Figure 5.14.

**Definition 5.2.4 (Structured Control Flow Graph).** If \( G \) is a rooted directed graph, we say \( G \) is a structured control flow graph if \( G \) is the trivial graph (the graph with a single node, which is also the root, and no edges) or if \( G \) can be created by starting with the trivial graph and performing a finite number of node replacements to replace nodes with graphs of the forms shown in Figure 5.14.

The first structure (Figure 5.14a) is that of simple sequential composition, with an edge going from the root node to a single end node. The next three structures (Figure 5.14b–d) are conditional structures: Figure 5.14b shows an \texttt{if} statement with no \texttt{else} clause, Figure 5.14c shows an \texttt{if} statement with an \texttt{else} clause, and Figure 5.14d shows a conditional in which both branches end with a (infinite) loop or a return so that there is nothing following the conditional, we refer to such conditionals as divergent conditionals since the branches do not come back together. The remaining three structures (Figure 5.14e–g) are all loop structures: Figure 5.14e shows a loop in which the loop condition is checked at the beginning (a \texttt{while} loop), Figure 5.14f shows a loop in which the loop condition is checked at the end (a \texttt{do-while} loop), and Figure 5.14g shows a loop which loops unconditionally, forming an infinite loop.

From this definition it is clear that the control flow graph of our example, shown in Figure 5.12, is a structured control flow graph. It may be obtained from the trivial graph by replacing the node with a sequential composition, replacing the end node of the sequential composition with a \texttt{while} loop, and then replacing the node inside the \texttt{while} loop with an \texttt{if-else} conditional.

In the strategy we check that the control flow graph is structured when we construct it in this section, and
Fig. 5.14: Control flow graphs of program structures

abort the strategy if it does not have the required structure. The introduction of sequential composition
does not cause the control flow graph of a structured program to cease being structured, so we may also
check that the control flow graph is structured when we construct it in Section 5.2.3.

Since we have defined the desired program structure in terms of a small number of standard structures,
we can identify each of these structures in the control flow graph and introduce them into the program,
collapsing the control flow graph in the process. In order to easily identify the structures in isolation
from other structures, we begin at the end nodes of the method (ignoring looping edges for the purposes
of determining end nodes) and work backwards, considering each node in turn. This is specified by the
loop beginning on line 5 of Algorithm 4. In our example this means we consider the $pc = 42$ and $pc = 35$
nodes first, then $pc = 28$ and $pc = 32$, then $pc = 21$, $pc = 39$, and finally $pc = 7$.

For each node, we check each type of structure to see if the control flow graph starting at that point
matches the structure, and introduce the structure if it does. The first type of structure we check for are
conditionals, beginning with those conditionals that are followed by another node, that is, those shown
in Figure 5.14b and c. These may be nested within one of the branches of another conditional in one of
the two ways shown below:

In the first case the sequential composition with the node at the end cannot be introduced until the
outermost conditional is introduced, because both of the inner conditionals end at the same point. How-
ever, in the second case the inner conditionals can be completely introduced, including the sequential
composition with the node after the end of each inner conditional, before the outer conditional is in-
troduced. To ensure both cases are covered, we separate the introduction of the conditional itself and
the sequential composition with the node after the conditional. In the first case the introduction of the
sequential composition is deferred until after the outermost conditional is resolved, whereas in the second
case it may be performed immediately after the introduction of the conditional.

We provide separate compilation rules for introducing if conditionals and if-else conditionals. An if
conditional with no else branch may be recognised from the control flow graph as having the form shown
in Figure 5.14b. However, it can also be recognised from the form of the Circus code in the Running
action, which will be that of a node whose sequence of instructions ends with an assignment of the form 
\( pc := \text{if } b \text{ then } x \text{ else } y \), and for which the \( pc = y \) node ends in an assignment \( pc := x \). Note that the branches will not be the other way round (i.e. the \( pc = x \) branch will not be the body of the conditional) since the conditional branches come from Java’s branching instructions which branch to the specified address if the condition is true and go to the next instruction if it is false. We provide for introducing such conditionals.

Rule [if conditional introduction]. If \( i \neq j \), \( i \neq k \), and

\[
\{ \text{frameStack} \neq \emptyset \} ; A \]

\[
\{ \text{frameStack} \neq \emptyset \} ; A \}; \{ \text{frameStack} \neq \emptyset \}
\]

then

\[
\mu X \bullet
\]

\[
\mu \left( \begin{array}{c}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \rightarrow \\
\text{if } \cdots \\
\text{if } \text{pc} = i \rightarrow A; \\
\text{pc} := \text{if } b \text{ then } j \text{ else } k \sqsubseteq A \\
\cdots \\
\text{if } \text{pc} = k \rightarrow B; \text{pc} := j \\
\cdots \\
\text{fi}; \text{Poll}; X \\
\text{fi}
\end{array} \right)
\]

introduces a conditional for nodes that match the form described above, which in the rule is the \( pc = i \) node. The conditional is introduced with the true branch being empty (represented here by \textbf{Skip}) and the false branch containing the instructions in the body of the conditional. The assignment \( pc := j \) is moved outside the conditional from both the empty true branch and the end of the false branch, so that a sequential composition with the node after the conditional can be introduced later on. As in , the sequence of actions for the node must not affect the nonemptiness of the \text{frameStack}. A similar condition is required for all the rules in this section. We also require that the targets of the conditional are different from the node at which the conditional is introduced, since that would introduce a loop, which is not the purpose of this rule. is applied on line 6 of Algorithm 4. Note that, since the structure can be identified from the form of the \textit{Circus} code alone, it is node necessary to guard the application of the rule with a condition on the control flow graph.

After attempting to introduce an \textbf{if} conditional, we attempt to introduce an \textbf{if-else} conditional, the form of which is shown in Figure 5.14a. As with an \textbf{if} conditional, a node with an \textbf{if-else} conditional will end with an assignment of the form \( pc := \text{if } b \text{ then } x \text{ else } y \), but the \( pc = x \) and \( pc = y \) nodes are required to end with a common assignment \( pc := z \). Conditionals matching this form may be introduced using .

Rule [if-else conditional introduction]. If \( i \neq j \), \( i \neq k \), and

\[
\{ \text{frameStack} \neq \emptyset \} ; A \]

\[
\{ \text{frameStack} \neq \emptyset \} ; A \}; \{ \text{frameStack} \neq \emptyset \}
\]
then

\[
\mu X \bullet
\]

| if frameStack = ∅ \rightarrow Skip 
| if \( \text{frameStack} \neq ∅ \rightarrow \)
| if \( \cdots \)
| pc = i \rightarrow A; 
| \text{pc} := \text{if } b \text{ then } j \text{ else } k
| \cdots 
| pc = j \rightarrow B ; \text{pc} := x 
| \cdots 
| pc = k \rightarrow C ; \text{pc} := x
| \cdots 
| \text{fi}; \text{Poll}; \; X 

\[
\mu X \bullet
\]

| if frameStack = ∅ \rightarrow \text{Skip} 
| if frameStack \neq ∅ \rightarrow 
| if \( \cdots \)
| pc = i \rightarrow A; \text{Poll}; 
| pc := \text{if } b \text{ then } j \text{ else } k
| if b \rightarrow B 
| \neg b \rightarrow C 
| fi; \text{pc} := x 
| \cdots 
| pc = j \rightarrow B; \text{pc} := x 
| \cdots 
| pc = k \rightarrow C; \text{pc} := x 
| \cdots 
| \text{fi}; \text{Poll}; \; X 

\]

\(\mu\) operates similarly to \(\nu\) in how it introduces the conditional and moves the common \(\text{pc}\) assignment outside the conditional. However, \(\mu\) includes sequences of instructions for both branches of the introduced conditional, each of which end with a \(\text{pc}\) assignment to jump to the node after the conditional. The preconditions of \(\mu\) are the same as those of \(\nu\). \(\mu\) is applied on line 7 of Algorithm 4.

Having attempted to introduce conditionals with a node following them, we then consider conditionals that are not followed by a node. These conditionals are those where both branches end in a return or an infinite loop. This includes conditionals where both branches are a return, which can arise from multiple returns in the Java source code. Though multiple returns are not allowed in the final C code, the returns will all end up in branches of conditionals at the end of the method, so the actual return statement can be placed after the conditionals to create a single return statement at the end of the C function.

The form of this type of conditionals is that shown in Figure 5.14d. We require that the nodes in both branches of the conditional have only a single incoming edge each, and do not have any outgoing edges (at the point in the strategy where this type of conditional is introduced, that is, there may be more complex structures that have already been introduced in the branches). We check that the control flow graph beginning at the node being considered has this form on line 8 of the algorithm. This is to ensure unstructured conditionals such as the one shown below are ruled out by the strategy.

If the correct structure is present, then we introduce the conditional by applying \(\nu\) on line 9. This rule introduces the conditional in much the same way as \(\nu\) and \(\mu\) but it does not place any requirement on the structure of the conditional or move a \(\text{pc}\) assignment outside of it.

**Rule** [Conditional introduction]. If \(i \neq j, i \neq k\), and

\[
\{\text{frameStack} \neq ∅\}; \; A = \{\text{frameStack} \neq ∅\}; \; A ; \{\text{frameStack} \neq ∅\}
\]
After attempting to introduce conditionals, we may attempt to introduce loops. There are three types of loop to consider, as shown earlier: while loops (Figure 5.14e), do-while loops (Figure 5.14f), and infinite loops (Figure 5.14g). A while loop has a form similar to that of a conditional, except that one of the branches ends with a jump back to the beginning of the node with the conditional. This structure may be introduced using $\mu X \bullet$. This rule introduces a conditional at a node $pc = i$ with its false branch ending in an assignment of $i$ to $pc$, and introduces a recursion to the beginning of the $pc = i$ node in that branch of the conditional, representing a loop. Since this loop may be within a conditional, we simply move the $pc$ assignment for the true branch outside the conditional. A sequential composition can then be introduced later, as with if and if-else conditionals.

**Rule [while loop introduction 1].** If $i \neq j$,

\[
\begin{align*}
\{\text{frameStack} \neq \emptyset\} \cdot A &= \{\text{frameStack} \neq \emptyset\} \cdot A; \{\text{frameStack} \neq \emptyset\}
\end{align*}
\]

then

\[
\begin{align*}
\mu X \bullet &\quad \text{if frameStack} = \emptyset \longrightarrow \text{Skip} \\
&\quad \text{frameStack} \neq \emptyset \longrightarrow \\
&\quad \text{if} \ldots \\
&\quad \quad [pc = i \longrightarrow A; \\
&\quad \quad pc := \begin{cases} i & \text{if } b \text{ then } j \text{ else } k \\
&\quad \quad \vdots \\
&\quad \quad pc = j \longrightarrow B \\
&\quad \quad \vdots \\
&\quad \quad pc = k \longrightarrow C; \quad pc := i \\
&\quad \quad \vdots \\
&\quad \quad \text{fi; Poll; } X
\end{cases} \\
&\quad \text{fi}
\end{align*}
\]

\[
\begin{align*}
\mu X \bullet &\quad \text{if frameStack} = \emptyset \longrightarrow \text{Skip} \\
&\quad \text{frameStack} \neq \emptyset \longrightarrow \\
&\quad \text{if} \ldots \\
&\quad \quad [pc = i \longrightarrow \mu Y \bullet A; \\
&\quad \quad pc := \begin{cases} i & \text{if } b \text{ then } j \text{ else } k; \quad \text{Poll; } Y \\
&\quad \quad \vdots b \longrightarrow C; \quad pc := i; \quad \text{Poll; } Y \\
&\quad \quad \text{fi}; pc := j \\
&\quad \quad \vdots pc = j \longrightarrow B \\
&\quad \quad \vdots pc = k \longrightarrow C; \quad pc := i \\
&\quad \quad \vdots \text{fi; Poll; } X
\end{cases}
\end{align*}
\]

As a while loop may occur with the loop at the end of either condition branch (since the loop may be created by a goto instruction in the Java bytecode), we also provide a similar rule, that introduces
the loop in the true branch of the conditional. These two rules are applied on lines 11 and 12 of the algorithm.

The next type of loop we consider is the do-while loop, which has the form shown in Figure 5.14f. These loops are distinguished from while loops by the fact that the conditional pc assignment which causes the loop is at the end of the loop, rather than at the beginning or in the middle. We introduce these loops using

**Rule [do-while loop introduction].** If \( i \neq j \),

\[
\{ \text{frameStack} \neq \emptyset \}; \quad A = \{ \text{frameStack} \neq \emptyset \}; \quad A; \{ \text{frameStack} \neq \emptyset \}
\]

then

\[
\begin{align*}
\mu X \bullet \\
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\quad \text{if } \cdots \\
\quad \text{if } \cdots \\
\quad \text{if } \cdots \\
\quad \text{if } \cdots \\
\quad \text{if } \cdots \\
\quad \text{fi; Poll; } X
\end{align*}
\]

This rule introduces a conditional, as with \( \text{if } \) but the true branch contains just the recursive call, since the conditional occurs at the end of the loop. The \( pc \) assignment for the false branch is moved outside the conditional to allow a sequential composition to be introduced later, as in previous rules. Note that the false branch can never cause the loop in this case, since it will just go to the next instruction. Attempting to redirect it and create the loop with a \( \text{goto} \) instruction would add an instruction within the loop after the conditional, so it would be dealt with as a while loop. Therefore, it is not necessary to provide two compilation rules for do-while loops, unlike while loops where both cases must be accounted for. \( \text{if } \) is applied on line 13 of the algorithm.

The final loop structure that we attempt to introduce is that of an infinite loop. Infinite loops are rare in most programs, but an infinite loop is nonetheless a well-structured program construct that has use in a few cases so we handle it here. An infinite loop may be identified as a block of instructions that ends with a \( pc \) assignment that causes a jump back to the beginning of the block of instructions. Such a block will have a control flow graph of the form shown in Figure 5.14g. We introduce these loops using

**Rule [Infinite loop introduction].** If

\[
\begin{align*}
\{ \text{frameStack} \neq \emptyset \}; \quad A = \{ \text{frameStack} \neq \emptyset \}; \quad A; \{ \text{frameStack} \neq \emptyset \}
\end{align*}
\]
then

\[
\mu X \bullet \begin{cases}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \rightarrow \text{if } \cdots \begin{cases}
\mu Y \bullet \text{A} \\
\mu Y \bullet \text{A} \rightarrow \text{Poll} \hspace{1em} Y
\end{cases} \\
\text{fi} \hspace{1em} \text{Poll} \hspace{1em} \text{X}
\end{cases}
\]

After we have attempted to introduce each of the structures for a particular node, we attempt to introduce a sequential composition. This ensures that if, if-else, while and do-while structures that occur within conditionals are sequentially composed with the node following them if possible. It also handles cases where sequential compositions occur before loops, preventing them from being introduced in Section 5.2.3 without interfering with the introduction of the loop. Such a case occurs at the \( \text{pc} = 7 \) node in our example. The requirement for sequential composition to be introduced is the same as in Section 5.2.3: it must be a simple sequential composition from a node with a single outgoing edge to a node with a single incoming edge. Thus we check for a simple sequence on line 15 of Algorithm 4. The sequential composition is then introduced on line 16 if it is a simple sequential composition.

As mentioned earlier, these steps are repeated for each node, working backwards through the control flow graph of each method. Given a structured control flow graph at the beginning, this means all the structures in the method are introduced, reducing the control flow graph to a single node.

In our example, we begin at the \( \text{pc} = 35 \) node, where there are no structures to introduce. The same holds true of the \( \text{pc} = 28 \) and \( \text{pc} = 32 \) nodes (note that the edges coming from them are not simple sequential compositions). An if-else conditional is introduced at \( \text{pc} = 21 \), absorbing the \( \text{pc} = 28 \) and \( \text{pc} = 32 \) nodes. The sequential composition from the \( \text{pc} = 21 \) node to the \( \text{pc} = 35 \) node can then be introduced immediately as it is now a simple sequential composition (because it is not at the end of an outer conditional). We then introduce a while loop at the \( \text{pc} = 39 \) loop (using \( [\hspace{1em} ] \)), and the sequential composition with the \( \text{pc} = 42 \) node is introduced afterwards. Finally, a sequential composition from the \( \text{pc} = 7 \) to the \( \text{pc} = 39 \) node is introduced, collapsing the control flow graph to a single node. The code at \( \text{pc} = 7 \) is then that shown earlier in Figure 5.5.

5.2.5 Resolve Method Calls

When a method is complete, calls to that method can then be resolved. This is performed after introduction of loops and conditionals, ensuring methods with loops and conditionals are complete so that this step can be applied.

This step begins with the copying of the method into a separate action, so that it can be referenced elsewhere. This is performed by as described by Algorithm 5.

Algorithm 5 Separate Complete Methods

1: \textbf{procedure} \textit{SeparateCompleteMethods}  
2: \hspace{1em} \textbf{for} \hspace{0.5em} m \leftarrow \textit{methods} \hspace{0.5em} \textbf{do}  
3: \hspace{1.5em} \textbf{if} \hspace{0.5em} \textit{MethodIsComplete}(m) \hspace{0.5em} \textbf{then}  
4: \hspace{2em} \textit{ApplyCopyRule}(m)  
5: \hspace{1.5em} \textbf{end if}  
6: \hspace{1em} \textbf{end for}  
7: \textbf{end procedure}

Algorithm 5 looks at each method separately, as specified by the loop on line 2 and determines if it is complete, on line 3. This involves a simple syntactic check that each conditional branch ends in a return
instruction or a recursion. Those methods that are complete are moved into a separate action by an application of the copy rule.

In our example, the method $f$ of the $TPK$ class, which starts at $pc = 43$, is complete on the first iteration of the loop on line 3 of Algorithm 6 with the $Running$ action as shown below. The method is complete in this case because it consists of a straight sequence of instructions ending with $HandleAreturnEPC$, which represents the $return$ instruction.

```
Running ≜
  if frameStack = ∅ → Skip
  else frameStack ≠ ∅ →
    if pc = 0 → · · ·
      · · ·
    pc = 43 → HandleAloadEPC(0) ; Poll ; HandleAloadEPC(0) ; Poll ; HandleIaddEPC ; Poll ; HandleAloadEPC(0) ; Poll ; HandleIaddEPC ; Poll ; HandleIconstEPC(5) ; Poll ; HandleIaddEPC ; Poll ; HandleAreturnEPC
    · · ·
  fi ; Poll ; Running
fi
```

The sequence of instructions at $pc = 43$ can then be copied into a separate action, shown below. The name of this action contains the name of the class and method identifier of the method it represents.

```
TPK_f ≜ HandleAloadEPC(0) ; Poll ; HandleAloadEPC(0) ; Poll ; HandleIaddEPC ; Poll ; HandleAloadEPC(0) ; Poll ; HandleIaddEPC ; Poll ; HandleIconstEPC(5) ; Poll ; HandleIaddEPC ; Poll ; HandleAreturnEPC
```

After all the complete methods have been copied into separate actions, calls to those methods are resolved. This is performed as described by Algorithm 6.

```
Algorithm 6 Resolve Method Calls
1:   procedure ResolverMethodCalls
2:     for $m$ ← methods do
3:       for $mc$ ← UnresolvedMethodsCalls($m$) do
4:         targets ← DetermineMethodCallTargets($mc$)
5:         if $\# targets = 1$ then
6:           Apply
7:         else
8:           Apply
9:         end if
10:     end for
11:   end for
12: end procedure
```

To resolve a method call, the type of method call must be considered. Some methods are handled directly by the SCJVM as they relate to the SCJVM services. Such methods are treated specially by the interpreter, communicating with the launcher to perform the behaviour of the method. In cases where the method invocation is simply handled by communication with the launcher and then followed by execution of the next instruction, the control flow can be introduced using $[]$ as for other instructions with simple control flow.

For method calls that do not require special handling, the control flow is that of the corresponding method’s action, followed by execution of the next instruction. If the method is called with static dispatch (as is the case with the $invokespecial$ and $invokestatic$ instructions), the correct method,
and hence the corresponding action can be easily determined. The method call is then resolved using.

We require as a precondition of the rule that the method action returns to the return address stored on the stack, to ensure that the control flow is resolved correctly. This will be the case for a method where every path of execution ends in a return or loop, since returns establish the condition and loops will either lead to a return eventually or form an infinite loop that may be followed by any action (including the assumption we require).

**Rule** [Method call resolution]. If an action $M$ is such that

\[
\{(\text{head frameStack}).\text{storedPC} = i \land \text{frameStack} = fs\}; \ M
\]

\[
\{(\text{head frameStack}).\text{storedPC} = i \land \text{frameStack} = fs\}; \ M; \ \{\text{pc} = i \land \text{frameStack} = \text{tail fs}\}
\]

and $i \neq j$,

\[
\begin{align*}
\mu X \cdot & \begin{cases} 
\text{if frameStack} = \emptyset \rightarrow \text{Skip} \\
\quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \text{if} \cdots \\
\quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad \quad
for \( \{1, \ldots, n\} \) and \( i \neq j \),

\[
\begin{align*}
\mu X \bullet \\
\text{if } frameStack = \emptyset \rightarrow \text{Skip} \\
\qquad \text{if } \cdots \\
\quad \text{if } pc = i \rightarrow A(\text{class}, \text{method}); \\
\quad \qquad \{\text{class} \in \{c_1, \ldots, c_n\}\}; \\
\quad \qquad \left(\exists \text{retAddr}?: == pc + 1 \bullet \right. \\
\quad \quad \text{SetReturnAddr}; \\
\quad \quad pc := \text{entry}(\text{class}, \text{method}); \\
\quad \quad \text{Poll}; \ X
\end{align*}
\]

The resolution of loops, conditionals and methods is performed in a loop until all the methods have been separated into their own action. The remaining use of the program counter in the main actions of \( \text{Thr} \) can then be eliminated as described in the next section.

### 5.2.6 Refine Main Actions

At this stage of the strategy, the only place that the program counter is used is when the first method is started, when it is used to select the method action to execute, which will then proceed without any need for the program counter value. This can be eliminated by replacing it with a choice over the method rather than the program counter. This must be performed in the two places that the \( \text{Running} \) action occurs: the \( \text{MainThread} \) and \( \text{NotStarted} \) actions. Since the main action of \( \text{Thr} \) is a guarded choice of these actions depending on whether its thread parameter is the \textit{main} thread, these actions may be thought of as two alternative main actions for \( \text{Thr} \).

The context of the \( \text{Running} \) action in both of these main actions is the same: the frame stack has only one frame and the program counter is set to the entry point of a method. This means that the same rule can be used for both main actions by introducing an assumption that states that context. Additionally, we can use the fact that each method action will, when started with a frame stack containing a single frame, cause the frame stack to become empty. This allows us to eliminate the loop in \( \text{Running} \), reducing it entirely to a choice of method action from the class and method identifier. The overall transformation of \( \text{Running} \) in its context is described by

**Rule [Main Action Refinement]**. If \( entry(c_i, m_i) = j_i \) for \( i \in \{1, \ldots, n\} \) and

\[
\{\# frameStack = 1\}; \ M_i = \\
\{\# frameStack = 1\}; \ M_i; \ \{frameStack = \emptyset\}
\]
Once this has been performed, the program counter value is no longer used to determine the control flow of the program, so a trivial data refinement can be performed to eliminate \( pc \) from the state of the \( Thr \) process.

5.2.7 Remove \( pc \) From State

5.3 Elimination of Frame Stack

5.4 Data Refinement of Objects
Chapter 6

Conclusions

In this chapter we conclude by summarising the contributions of this dissertation in Section 6.1. We then discuss directions of future work in Section 6.2.

6.1 Summary of Contributions

We have considered the safety-critical variant of Java and the virtual machines designed to run programs written in it. We have concluded that none of the virtual machines is formally verified and that many of them precompile programs to native code. Given the need for a formally verified virtual machine, we have stated our aim to specify a framework within which an SCJVM can be verified.

Having noted that SCJ virtual machines employ compilation, we have surveyed some of the work on compiler correctness, particularly those related to Java compilation. We have established that two approaches to compiler correctness have been used: the commuting-diagram approach and the algebraic approach. We have decided to adopt the algebraic approach and chosen Circus as a specification language.

To specify an SCJVM we have identified the requirements of the virtual machine services required to support SCJ programs. We have also constructed a formal model of those requirements in the Circus specification language.

Contact with one of the authors of the SCJ specification has allowed us to obtain clarifications where the specification is unclear. The development of the formal model has helped in the identification of the areas that require clarification. It may be noted that the interface we have defined is not the only one that can support SCJ, but its overall functionality must be present in all SCJ virtual machines in some way.

We have also created a formal model of the core execution environment that executes SCJ programs in an SCJVM. This model has been created by identifying a minimal subset of Java bytecode and defining its semantics, and then constructing a Circus model of an interpreter for that subset with the necessary infrastructure around it.

Our work is done in the context of a wider effort to facilitate fully verified SCJ programs. There has already been work on generating correct SCJ programs from Circus specifications and formalisation of the SCJ memory model. These works allow for verification of SCJ programs, with our work covering the next stage in ensuring those programs can be run correctly.

Since our work addresses the execution of Java bytecode, it must still be ensured that SCJ programs can be compiled to bytecode correctly. Since SCJ does not make any syntactic changes to Java and the semantic changes can be dealt with at the level of Java bytecode, a standard Java compiler suffices for SCJ. As discussed earlier, there has been plenty of work on correct compilation of Java programs, so it can be seen that there is already sufficient work to permit correct compilation to Java.
bytecode. This then leaves us with correct SCJ programs in Java bytecode and the focus of our work is on the next stage of running those programs.

Finally, as we are adopting the approach of compilation to C, it must also be ensured that the C code can be compiled correctly. We note that there has been much work on verified C compilation \cite{14, 50, 52-54} and, in particular, that the CompCert project provides a functioning formally verified C compiler that can be used.

So our proposed work is the final piece needed for complete verification of SCJ programs down to executable code.

6.2 Future Work

As part of our agenda for future work, in the scope of this thesis, we will specify and prove the correctness of a strategy for compiling Java bytecode to C using the algebraic approach. This will involve defining the semantics of a subset of C in terms of Circus and proving refinement laws between our Java bytecode subset model in the core execution environment and the C subset. We will mechanise these proofs in the Isabelle proof assistant in order to have assurance that the proofs are sound.

We will also continue to work on validation of the formal models we have created by proving correctness properties. This may require revision of the formal model due to errors uncovered in the process of proving essential properties.

Beyond the scope of our work, possible future work includes the verification of an SCJ virtual machine using our framework or even the creation of a correct-by-construction virtual machine from our specification. The option of deriving a correct virtual machine from our specification may be more desirable than verifying an existing one. This is because virtual machines can often be complex and therefore difficult to verify in a structured way. Moreover, while the effort of proving a virtual machine correct may uncover bugs, it may be a challenge to fix them. Also, the design of an existing virtual machine may not exactly meet the structure of our specification, requiring restructuring to allow the proof effort to begin.

On the other hand, the fact that Circus allows for refinement means that a correct virtual machine can be constructed from our model in a stepwise and modular fashion, being shown to be correct at each stage of the process. Facilitating such work is the ultimate aim of our work, in order to provide for the correct running of SCJ programs.


Appendix A

Full SCJVM Services Model

A.1 Memory Manager

section memorymanager parents circus_toolkit

[BackingStoreID]

MemoryAddress == N

ContiguousMemory == { m : P MemoryAddress | ∃ a, b : MemoryAddress • m = a .. b }

[StackID]

channel MMgetRootBackingStore
channel MMgetRootBackingStoreRet : BackingStoreID
channel MMgetTotalSize : BackingStoreID; MMgetTotalSizeRet : N
channel MMgetUsedSize : BackingStoreID; MMgetUsedSizeRet : N
channel MMgetFreeSize : BackingStoreID; MMgetFreeSizeRet : N
channel MMgetRemainingBackingStore : BackingStoreID; MMgetRemainingBackingStoreRet : N
channel MMfindBackingStore : MemoryAddress
channel MMfindBackingStoreRet : BackingStoreID
channel MMallocateMemory : BackingStoreID × N
channel MMallocateMemoryRet : MemoryAddress
channel MMmakeBackingStore : BackingStoreID × N × N
channel MMmakeBackingStoreRet : BackingStoreID
channel MMclearBackingStore : BackingStoreID
channel MMresizeBackingStore : BackingStoreID × N
channel MMresizeBackingStoreRet : BackingStoreID
channel MMcreateStack : N; MMcreateStackRet : StackID
channel MMdestroyStack : StackID
channel MMlockMemory, MMunlockMemory
MMReport ::= MMokay | MMoutOfMemory | MMnotEmpty |
  MMnonexistentAllocation | MMsizeTooSmall | MMnonexistentBS |
  MMrootBSResize | MMnotOnlyChild | MMunknownAddress | MMnonexistentStack |
  MMcannotShrink | MMfragmentation | MMnotContiguousWithFree |
  MMnoFreeIdentifier | MMnotLocked | MMalreadyLocked

channel MMreport : MMReport

channel MMinit : ContiguousMemory × ℕ × ContiguousMemory

process MemoryManager ≡ begin

| allocationOverhead, backingStoreOverhead : ℕ

A.1.1 Memory Blocks

\[
\begin{align*}
\text{MemoryBlock} & : \\
& \text{free, total} : \text{ContiguousMemory} \\
& \text{used} : \mathbb{F} \text{MemoryAddress} \\
& \text{used} \cup \text{free} \subseteq \text{ContiguousMemory} \\
& \text{used} \cup \text{free} \subseteq \text{total} \\
& \text{used} \cap \text{free} = \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{MemoryBlockInit} & : \\
& \text{MemoryBlock}' \\
& \text{addresses} : \text{ContiguousMemory} \\
& \text{total}' = \text{addresses} \\
& \text{free}' \subseteq \text{addresses} \\
& \text{used}' = \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{MAllocate} & : \\
& \Delta \text{MemoryBlock} \\
& \text{size} : \mathbb{N} \\
& \text{allocated}! : \text{ContiguousMemory} \\
& \text{size} \leq \# \text{free} \\
& \# \text{allocated}! = \text{size}! \\
& \text{allocated}! \subseteq \text{free} \\
& \text{used}' = \text{used} \cup \text{allocated}! \\
& \text{free}' = \text{free} \setminus \text{allocated}! \\
& \text{total}' = \text{total}
\end{align*}
\]
**MBClear**

\[Δ\text{MemoryBlock}\]

\[\begin{align*}
\text{free}' &= \text{used} \cup \text{free} \\
\text{used}' &= \emptyset \\
\text{total}' &= \text{total}
\end{align*}\]

**MBResize**

\[Δ\text{MemoryBlock}\]

\[\begin{align*}
\text{newAddresses}! : \text{ContiguousMemory} \\
\text{oldAddresses}! : \text{ContiguousMemory}
\end{align*}\]

\[\begin{align*}
\# \text{newAddresses} &\geq \# \text{total} - \# \text{free} \\
\text{used} &= \emptyset \\
\text{total}' &= \text{newAddresses}! \\
\text{free}' &\subseteq \text{newAddresses}! \\
\# \text{free}' &= \# \text{newAddresses}! - (\# \text{total} - \# \text{free}) \\
\text{used}' &= \text{used} \\
\text{oldAddresses}! &= \text{total}
\end{align*}\]

**MBContentsResize**

\[Δ\text{MemoryBlock}\]

\[\begin{align*}
\text{oldAddresses}! : \text{ContiguousMemory} \\
\text{newSize}! : \text{N} \\
\text{newAddresses}! : \text{ContiguousMemory}
\end{align*}\]

\[\begin{align*}
\text{oldAddresses}! &\subseteq \text{used} \\
\text{free} \cup \text{oldAddresses}! &\in \text{ContiguousMemory} \\
\text{newSize}! &\leq (\#(\text{free} \cup \text{oldAddresses}!)) \\
\exists \text{newFree} : \text{ContiguousMemory}; \text{newUsed} : \mathbb{P} \text{MemoryAddress} | \\
\text{newFree} &= \text{free} \cup \text{oldAddresses}! \\
\text{newUsed} &= \text{used} \setminus \text{oldAddresses}! \\
\text{MBAllocate}[\text{newFree}/\text{free}, \text{newUsed}/\text{used}, \\
\text{newSize}!/\text{size}!, \text{newAddresses}!/\text{allocated}!]
\end{align*}\]

**MBGetTotalSize**

\[Ξ\text{MemoryBlock}\]

\[\begin{align*}
\text{size}! : \text{N} \\
\text{size}! &= \# \text{used} + \# \text{free}
\end{align*}\]

**MBGetUsedSize**

\[Ξ\text{MemoryBlock}\]

\[\begin{align*}
\text{size}! : \text{N} \\
\text{size}! &= \# \text{used}
\end{align*}\]

**MBGetFreeSize**

\[Ξ\text{MemoryBlock}\]

\[\begin{align*}
\text{size}! : \text{N} \\
\text{size}! &= \# \text{free}
\end{align*}\]
Success
report! : MMReport
report! = MMokay

MBOutOfMemory
\exists MemoryBlock
size? : N
report! : MMReport
¬ size? \leq \# free
report! = MMoutOfMemory

MBNotEmpty
\exists MemoryBlock
report! : MMReport
used \neq \emptyset
report! = MMnotEmpty

MBResizeBelowOverhead
\exists MemoryBlock
newAddresses? : ContiguousMemory
report! : MMReport
¬ \# newAddresses? \geq \# total - \# free
report! = MMcannotShrink

MBNonexistentAllocation
\exists MemoryBlock
oldAddresses? : ContiguousMemory
report! : MMReport
¬ (oldAddresses? \subseteq used)
report! = MMnonexistentAllocation

MBInsufficientFreeSize
\exists MemoryBlock
oldAddresses? : ContiguousMemory
newSize? : N
report! : MMReport
¬ newSize? \leq \#(free \cup oldAddresses?)
report! = MMoutOfMemory

MBNotContiguousWithFree
\exists MemoryBlock
oldAddresses? : ContiguousMemory
report! : MMReport
free \cup oldAddresses? \notin ContiguousMemory
report! = MMnotContiguousWithFree
RMBAllocate == (MBAlocate ∧ Success) ∨ MBOutOfMemory
RMBClear == MBClear ∧ Success
RMBResize == (MBResize ∧ Success) ∨ MBNotEmpty ∨ MBResizeBellowOverhead
RMBContentsResize == (MBContentsResize ∧ Success) ∨ MBNonexistentAllocation ∨ MBInsufficientFreeSize ∨ MBNotContiguousWithFree
RMBGetTotalSize == MBGetTotalSize ∧ Success
RMBGetUsedSize == MBGetUsedSize ∧ Success
RMBGetFreeSize == MBGetFreeSize ∧ Success

A.1.2 Backing Stores

<table>
<thead>
<tr>
<th>BackingStore</th>
</tr>
</thead>
<tbody>
<tr>
<td>objectSpace : MemoryBlock</td>
</tr>
<tr>
<td>bsSpace : MemoryBlock</td>
</tr>
<tr>
<td>children : F BackingStoreID</td>
</tr>
<tr>
<td>self : BackingStoreID</td>
</tr>
</tbody>
</table>

self ∉ children
objectSpace.used ∪ objectSpace.free ∪ bsSpace.used ∪ bsSpace.free ∈ ContiguousMemory
objectSpace.total ∩ bsSpace.total = ∅
#(objectSpace.used ∪ objectSpace.free ∪ bsSpace.used ∪ bsSpace.free) + backingStoreOverhead = #(objectSpace.total ∪ bsSpace.total)

Most of the operations on BackingStores are specified by promoting operations on the objectSpace and bsSpace of the BackingStore. We therefore define a promotion schema PromoteMBsToBS to perform this.

<table>
<thead>
<tr>
<th>PromoteMBsToBS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ΔBackingStore</td>
</tr>
<tr>
<td>ΔMemoryBlock₁</td>
</tr>
<tr>
<td>ΔMemoryBlock₂</td>
</tr>
</tbody>
</table>

θ MemoryBlock₁ = objectSpace
θ MemoryBlock₂ = bsSpace
objectSpace' = θ MemoryBlock₁
bsSpace' = θ MemoryBlock₂
**BackingStoreInit**

<table>
<thead>
<tr>
<th>BackingStore'</th>
<th>addresses? : ContiguousMemory</th>
</tr>
</thead>
<tbody>
<tr>
<td>self? : BackingStoreID</td>
<td></td>
</tr>
<tr>
<td>objectSpaceSize?: N</td>
<td></td>
</tr>
</tbody>
</table>

\[ \exists \text{objectAddresses}, \text{bsAddresses} : \text{ContiguousMemory} \bullet \]

\[ \begin{aligned}
& \exists \text{MemoryBlock}'_1, \text{MemoryBlock}'_2 \mid \\
& \quad \text{MemoryBlockInit}'_1[\text{objectAddresses}/\text{addresses}?] \land \\
& \quad \text{MemoryBlockInit}'_2[\text{bsAddresses}/\text{addresses}?] \bullet \\
& \quad \text{objectSpace}' = \emptyset \text{MemoryBlock}'_1 \land \\
& \quad \text{bsSpace}' = \emptyset \text{MemoryBlock}'_2 \land \\
& \quad \text{objectAddresses} \cup \text{bsAddresses} = \text{addresses}? \\
& \quad \# \text{objectAddresses} = \# \text{objectSpace}'.\text{free} + \# \text{bsAddresses} + \text{backingStoreOverhead} \\
& \quad \# \text{addresses}? = \# \text{objectSpace}'.\text{free} + \# \text{bsSpace}'.\text{free} + \text{backingStoreOverhead} \\
& \quad \text{self}' = \text{self}? \\
& \quad \text{children}' = \emptyset
\end{aligned} \]

**BSAllocateChild0**

<table>
<thead>
<tr>
<th>BSAllocateChild0</th>
<th>\Delta \text{BackingStore}</th>
</tr>
</thead>
<tbody>
<tr>
<td>size?: N</td>
<td>\text{BackingStoreID}</td>
</tr>
<tr>
<td>childID!: \text{BackingStoreID}</td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{aligned}
& \text{size}? \geq \text{backingStoreOverhead} \\
& \text{childID}! \notin \text{children} \land \text{childID}! \neq \text{self} \\
& \text{children}' = \text{children} \cup \{ \text{childID}! \} \\
& \text{self}' = \text{self}
\end{aligned} \]

**BSAllocateChild == \exists \Xi \text{MemoryBlock}_1; \Delta \text{MemoryBlock}_2 \bullet**

\[ \text{BSAllocateChild0} \land \text{MBAllocate}_2[\text{size}/\text{size}?_2, \text{allocated}!/\text{allocated}?_2] \land \text{PromoteMBsToBS} \]

**BSClear0**

<table>
<thead>
<tr>
<th>BSClear0</th>
<th>\Delta \text{BackingStore}</th>
</tr>
</thead>
<tbody>
<tr>
<td>children' = \emptyset</td>
<td></td>
</tr>
<tr>
<td>self' = self</td>
<td></td>
</tr>
</tbody>
</table>

**BSClear == \exists \Delta \text{MemoryBlock}_1; \Delta \text{MemoryBlock}_2 \bullet**

\[ \text{BSClear0} \land \text{MBClear}_1 \land \text{MBClear}_2 \land \text{PromoteMBsToBS} \]

**BSAllocate0**

<table>
<thead>
<tr>
<th>BSAllocate0</th>
<th>\Delta \text{BackingStore}</th>
</tr>
</thead>
<tbody>
<tr>
<td>size?: N</td>
<td>\text{allocated}! : \text{ContiguousMemory}</td>
</tr>
<tr>
<td>allocated!: \text{ContiguousMemory}</td>
<td></td>
</tr>
<tr>
<td>actualSize: N</td>
<td></td>
</tr>
</tbody>
</table>

\[ \begin{aligned}
& \text{actualSize} = \text{size}? + \text{allocationOverhead} \\
& \text{children}' = \text{children} \\
& \text{self}' = \text{self}
\end{aligned} \]
BSAllocate == ∃ ΔMemoryBlock_1; ∃MemoryBlock_2; actualSize : N •
BSAllocate0 ∧ MBSAllocate_1[actualSize/size_1, allocated!/allocated_1] ∧ PromoteMBsToBS

BSResize

ΔBackingStore
newSize? : N

newSize? ≥ # objectSpace.used
objectSpace'.used = objectSpace.used
#{objectSpace'.used ∪ objectSpace'.free} = newSize?
objectSpace'.free ∪ bsSpace'.free = objectSpace.free ∪ bsSpace.free
bsSpace.used = ∅ = bsSpace'.used
children = ∅ = children'
self' = self

BSGetTotalSize == ∃ ΔMemoryBlock_1; ∃MemoryBlock_2 • MBGetTotalSize_1[size!/size_1] ∧ PromoteMBsToBS
BSGetUsedSize == ∃ ΔMemoryBlock_1; ∃MemoryBlock_2 • MBGetUsedSize_1[size!/size_1] ∧ PromoteMBsToBS
BSGetFreeSize == ∃ ΔMemoryBlock_1; ∃MemoryBlock_2 • MBGetFreeSize_1[size!/size_1] ∧ PromoteMBsToBS

BSGetRemainingBS == ∃ΔMemoryBlock_1; ΔMemoryBlock_2 • MBGetFreeSize_2[size!/size_2] ∧ PromoteMBsToBS

BSHasChildren

∃BackingStore
report! : MMReport

children ≠ ∅ ∨ bsSpace.used ≠ ∅
report! = MMnotEmpty

BSNonexistentAllocation

∃BackingStore
oldID? : BackingStoreID
report! : MMReport

oldID? ∉ children
report! = MMnonexistentAllocation

BSSizeTooSmall

∃BackingStore
newSize? : N
report! : MMReport

¬ newSize? ≥ # objectSpace.used
report! = MMSizeTooSmall
A.1.3 GlobalMemoryManager

GlobalStoresManager

stores : BackingStoreID \rightarrow BackingStore
rootBackingStore : BackingStoreID

rootBackingStore \in \text{dom stores}
\forall bsid : \text{dom stores} •
  (\text{stores bsid}).self = bsid \land
  (\text{stores bsid}).children \subseteq \text{dom stores} \land
  (\lambda childID : (\text{stores bsid}).children •
    (\text{stores childID}).objectSpace.total \cup (\text{stores childID}).bsSpace.total)
  \text{partition} (\text{stores bsid}).bsSpace.used
GlobalMemoryManager

GlobalStoresManager
childRelation : BackingStoreID ↔ BackingStoreID

∀ bsid : dom stores • childRelation \( \{ \{ bsid \} \} \) = (stores bsid).children

dom stores = (childRelation \( \ast \) \( \{ \{ \text{rootBackingStore} \} \} \)

∀ bsid : dom stores • bsid \( \not\in \) childRelation \( \ast \) \( \{ \{ bsid \} \} \)

GlobalMemoryManagerInit

GlobalMemoryManager'
addresses? : ContiguousMemory
objectSpaceSize? : \( \mathbb{N} \)
report! : MMReport

∃ BackingStore' \( \mid \) RBackingStoreInit[rootBackingStore' / self?] •
stores' = \{rootBackingStore' \( \mapsto \) \( \theta \) BackingStore'\}
childRelation' = \( \emptyset \)

PromoteBS

\( \Delta \) GlobalMemoryManager
\( \Delta \) BackingStore
bs? : BackingStoreID

bs? \( \in \) dom stores
\( \theta \) BackingStore = stores bs?
stores' = stores \( \oplus \) \{bs? \( \mapsto \) \( \theta \) BackingStore'\}
rootBackingStore' = rootBackingStore

GlobalAllocateMemory ==

\( \exists \Delta \) BackingStore; allocated! : ContiguousMemory • RBSAllocate \( \land \) PromoteBS \( \land \)
[allocated! : ContiguousMemory; address! : MemoryAddress
| address! = min allocated!!]

GlobalResizeBS == \( \exists \Delta \) BackingStore • RBSResize \( \land \) PromoteBS

GlobalGetTotalSize == \( \exists \Delta \) BackingStore • RBSGetTotalSize \( \land \) PromoteBS

GlobalGetUsedSize == \( \exists \Delta \) BackingStore • RBSGetUsedSize \( \land \) PromoteBS

GlobalGetFreeSize == \( \exists \Delta \) BackingStore • RBSGetFreeSize \( \land \) PromoteBS

GlobalGetRemainingBS == \( \exists \Delta \) BackingStore • RBSGetRemainingBS \( \land \) PromoteBS
GlobalMakeBS

\[ \Delta \text{GlobalMemoryManager} \]
\[ \text{size?} : \mathbb{N} \]
\[ \text{objectSpaceSize?} : \mathbb{N} \]
\[ \text{parentID?} : \text{BackingStoreID} \]
\[ \text{childID!} : \text{BackingStoreID} \]

\[ \text{parentID?} \in \text{dom stores} \]
\[ \exists \text{actualSize} : \mathbb{N} \mid \text{actualSize} = \text{size?} + \text{backingStoreOverhead} \]
\[ \exists \text{allocated!} : \text{ContiguousMemory} \]
\[ \exists \text{Parent, child} : \text{BackingStore} \]
\[ (\exists \Delta \text{BackingStore}; \text{report!} : \text{MMReport} \mid \]
\[ \text{RBSAllocateChild}[\text{actualSize}/\text{size?}] \]
\[ \theta \text{BackingStore} = \text{stores parentID?} \land \]
\[ \text{Parent} = \theta \text{BackingStore}' \land \]
\[ \text{childID!} \notin \text{dom stores} \land \]
\[ \text{report!} = \text{MMokay} \land \]

\[ (\exists \text{BackingStore}'; \text{report!} : \text{MMReport} \mid \]
\[ \text{RBackingStoreInit}[	ext{allocated!/addresses?}, \text{childID!/self?}] \]
\[ \text{self}' = \text{childID!} \land \]
\[ \text{child} = \theta \text{BackingStore}' \land \]
\[ \text{report!} = \text{MMokay} \land \]
\[ \text{stores'} = \text{stores} \oplus \{ \text{parentID?} \mapsto \text{Parent}, \text{childID!} \mapsto \text{child} \} \]
\[ \text{rootBackingStore}' = \text{rootBackingStore} \]

GlobalClearBS

\[ \Delta \text{GlobalMemoryManager} \]
\[ \text{toClear?} : \text{BackingStoreID} \]

\[ \text{toClear?} \in \text{dom stores} \]
\[ \exists \Delta \text{BackingStore}; \text{report!} : \text{MMReport} \mid \text{RBSClear} \]
\[ \theta \text{BackingStore} = \text{stores toClear?} \land \]
\[ \text{report!} = \text{MMokay} \land \]
\[ \exists \text{reachable} : \mathcal{F} \text{BackingStoreID} \mid \]
\[ \text{reachable} = \text{childRelation} + \{(\text{toClear?})\} \]
\[ \text{stores'} = (\text{reachable} \in \text{stores}) \oplus \{ \text{toClear?} \mapsto \theta \text{BackingStore}' \} \land \]
\[ \text{childRelation}' = \text{childRelation} \uplus \text{reachable} \]
\[ \text{rootBackingStore}' = \text{rootBackingStore} \]

GlobalFindAddress

\[ \exists \text{GlobalMemoryManager} \]
\[ \text{address?} : \text{MemoryAddress} \]
\[ \text{backingStore!} : \text{BackingStoreID} \]

\[ \text{address?} \in (\text{stores rootBackingStore}).\text{objectSpace}.\text{total} \]
\[ \cup (\text{stores rootBackingStore}).\text{objectSpace}.\text{total} \]
\[ \text{backingStore!} = (\mu \text{bsid} : \text{dom stores} \mid \text{address?} \in (\text{stores bsid}).\text{objectSpace}.\text{total}) \]

GlobalGetRootBackingStore

\[ \exists \text{GlobalMemoryManager} \]
\[ \text{rbs!} : \text{BackingStoreID} \]

\[ \text{rbs!} = \text{rootBackingStore} \]
**GlobalNonexistentBS**

\[ \exists \text{GlobalMemoryManager} \]

**bs?** : BackingStoreID

**report!** : MMReport

\[ bs? \not\in \text{dom stores} \]

**report! = MMnonexistentBS**

**GlobalRootBSResize**

\[ \exists \text{GlobalMemoryManager} \]

**toResize?** : BackingStoreID

**report!** : MMReport

\[ toResize? = \text{rootBackingStore} \]

**report! = MMrootBSResize**

**GlobalNotOnlyChild**

\[ \exists \text{GlobalMemoryManager} \]

**toResize?** : BackingStoreID

**report!** : MMReport

\[ toResize? \neq \text{rootBackingStore} \]

\[ toResize? \in \text{dom stores} \]

\[ \forall \text{bsid} : \text{dom stores} \bullet (\text{stores(} \text{bsid} \text{)}).\text{children} \neq \{\text{toResize}\} \]

**report! = MMonotOnlyChild**

**GlobalAllocateChildError**

\[ \exists \text{GlobalMemoryManager} \]

**parentID?** : BackingStoreID

**size?** : N

**report!** : MMReport

\[ \text{parentID?} \in \text{dom stores} \]

\[ \exists \text{childID} : \text{BackingStoreID}; \text{allocated}! : \text{ContiguousMemory} \bullet \]

\[ \exists \text{actualSize} : \text{N} \mid \text{actualSize} = \text{size?} + \text{backingStoreOverhead} \bullet \]

\[ \exists \Delta \text{BackingStore} \mid \text{RBSAllocateChild[actualSize/size?] } \bullet \]

\[ \text{θ BackingStore} = \text{stores(} \text{parentID?} \text{)} \land \]

**report! = MMokay**

**GlobalResizeError**

\[ \exists \text{GlobalMemoryManager} \]

**toResize?** : BackingStoreID

**newSize?** : N

**report!** : MMReport

\[ toResize? \neq \text{rootBackingStore} \]

\[ toResize? \in \text{dom stores} \]

\[ \exists \Delta \text{BackingStore}; \text{newAddresses}? , \text{oldAddresses}! : \text{F MemoryAddress} \mid \text{RBSResize} \bullet \]

\[ \text{θ BackingStore} = \text{stores(} \text{toResize?} \text{)} \land \]

**report! = MMokay**
A.1.4 Stack Memory Manager

stackOverhead : N

StackMemoryManager  
MemoryBlock  
stacks : StackID ↦→ ContiguousMemory

stacks partition used

StackMemoryManagerInit  
StackMemoryManager′  
MemoryBlockInit  
stacks′ = ∅

StackCreate  
∆StackMemoryManager  
size? : N  
newStack! : StackID

ewStack ! ∉ dom stacks  
∃ actualSize : N | actualSize = size? + stackOverhead  
∃ report! : MMReport; allocated! : ContiguousMemory  
RMBAllocate[actualSize/size?] ∧ report! = MMokay ∧  
stacks′ = stacks ⊕ {newStack! ↦→ allocated!}
StackDelete

∆StackMemoryManager

stack? : StackID

stack? ∈ dom stacks
used' = used \ stacks stack?
free' = free \ stacks stack? ∈ ContiguousMemory
stacks' = { stack? } \ stacks
total' = total

StackNonexistentStack

ΞStackMemoryManager

stack? : StackID

report! : MMReport

stack? ∉ dom stacks
report! = MMnonexistentStack

StackMBAllocateError

ΞStackMemoryManager

size? : N

report! : MMReport

∃ actualSize : N | actualSize = size? + stackOverhead
∃ allocated! : ContiguousMemory

RMBAllocate[actualSize/size?] \ report! ≠ MMokay

StackFragmentation

ΞStackMemoryManager

stack? : StackID

report! : MMReport

stack? ∈ dom stacks
free ∪ stacks stack? ∉ ContiguousMemory
report! = MMfragmentation

RStackCreate == (StackCreate ∧ Success) ∨ StackMBAllocateError
RStackDelete == (StackDelete ∧ Success) ∨ StackNonexistentStack ∨ StackFragmentation

A.1.5 Memory Manager Operations

state GlobalMemoryManager ∧ StackMemoryManager

Init ≜ var report : MMReport

MMInit?addresses?objectSpaceSize?stackSpace →

RGlobalMemoryManagerInit ∧ StackMemoryManagerInit; 

MMReport!report → Skip
GetRootBackingStore ≡ var report : MMReport; rbs : BackingStoreID •
   MMgetRootBackingStore → (RGlobalGetRootBackingStore);
   MMgetRootBackingStoreRet!rbs → MMreport!report → Skip

GetTotalSize ≡ var report : MMReport; size : N •
   MMgetTotalSize?bs → (RGlobalGetTotalSize);
   MMgetTotalSizeRet!size → MMreport!report → Skip

GetUsedSize ≡ var report : MMReport; size : N •
   MMgetUsedSize?bs → (RGlobalGetUsedSize);
   MMgetUsedSizeRet!size → MMreport!report → Skip

GetFreeSize ≡ var report : MMReport; size : N •
   MMgetFreeSize?bs → (RGlobalGetFreeSize);
   MMgetFreeSizeRet!size → MMreport!report → Skip

GetRemainingBS ≡ var report : MMReport; size : N •
   MMgetRemainingBackingStore?bs → (RGlobalGetRemainingBS);
   MMgetRemainingBackingStoreRet!size → MMreport!report → Skip

FindBackingStore ≡ var backingStore : BackingStoreID; report : MMReport •
   MMfindBackingStore?address → (RGlobalFindAddress);
   MMfindBackingStoreRet!backingStore → MMreport!report → Skip

AllocateMemory ≡ var address : MemoryAddress; report : MMReport •
   MMallocateMemory?bs?size → (RGlobalAllocateMemory);
   MMallocateMemoryRet!address → MMreport!report → Skip

MakeBackingStore ≡ var childID : BackingStoreID; report : MMReport •
   MMmakeBackingStore?parentID?size?objectSpaceSize → (RGlobalMakeBS);
   MMmakeBackingStoreRet!childID → MMreport!report → Skip

ClearBackingStore ≡ var report : MMReport •
   MMClearBackingStore?toClear → (RGlobalClearBS);
   MMreport!report → Skip

ResizeBackingStore ≡ var newID : BackingStoreID; report : MMReport •
   MMresizeBackingStore?bs?newSize → (RGlobalResizeBS);
   MMresizeBackingStoreRet!newID → MMreport!report → Skip

CreateStack ≡ var newStack : StackID; report : MMReport •
   MMcreateStack?size → (RStackCreate);
   MMcreateStackRet!newStack → MMreport!report → Skip

DestroyStack ≡ var report : MMReport •
   MMdestroyStack?stack → (RStackDelete);
   MMreport!report → Skip
Loop ≜ GetRootBackingStore □ AllocateMemory
□ GetTotalSize □ GetUsedSize □ GetFreeSize
□ FindBackingStore □ MakeBackingStore □ ClearBackingStore
□ ResizeBackingStore □ CreateStack □ DestroyStack;
Loop

• Init ; Loop
end

A.2 Scheduler

section scheduler parents memorymanager

[ThreadID]

idle, main : ThreadID
idle ≠ main

minHwPriority, maxHwPriority : N
minSwPriority, maxSwPriority : N
normSwPriority : N

(maxHwPriority − minHwPriority) + (maxSwPriority − minSwPriority) ≥ 28
minSwPriority < maxSwPriority < minHwPriority < maxHwPriority
minSwPriority ≤ normSwPriority ≤ maxSwPriority

ThreadPriority == minSwPriority..maxSwPriority
InterruptPriority == minHwPriority..maxHwPriority

Priority == ThreadPriority ∪ InterruptPriority

channel CEEswitchThread : ThreadID × ThreadID

[ClassID, MethodID]

channel CEEstartThread : ThreadID × BackingStoreID × StackID × ClassID × MethodID × seq Word

channel CEEremoveThread : ThreadID

channel CEEproceed : ThreadID
ObjectID == Word

| null : ObjectID

[InterruptID]

channel HWinterrupt : InterruptID

channel HWenableInterrupts, HWdisableInterrupts

| clockInterrupt : InterruptID

channel RTClockInterrupt

channel Sinit : ThreadPriority × BackingStoreID
channel SgetMaxSoftwarePriority : Priority
channel SgetMinSoftwarePriority : Priority
channel SgetNormSoftwarePriority : Priority
channel SgetMaxHardwarePriority : Priority
channel SgetMinHardwarePriority : Priority
channel SgetMainThread : ThreadID
channel SmakeThread : ThreadPriority × ClassID × MethodID × seq Word
channel SmakeThreadRet : ThreadID
channel SstartThreads : F(ThreadID × BackingStoreID × StackID)
channel SgetCurrentThread : ThreadID
channel SsuspendThread
channel SresumeThread : ThreadID
channel SsetPriorityCeiling : ObjectID × Priority
channel StakeLock : ObjectID
channel SreleaseLock : ObjectID
channel SattachInterruptHandler : InterruptID × BackingStoreID × StackID × ClassID × ObjectID
channel SdetachInterruptHandler : InterruptID
channel SgetInterruptPriority : InterruptPriority
channel SgetInterruptPriorityRet : InterruptPriority
channel SdisableInterrupts, SenableInterrupts
channel SendThread

SReport ::= Sokay | SnonexistentThread | SthreadAlreadyStarted |
            SthreadNotBlocked | SthreadNotBlockable | SthreadNotDestroyable |
            SlockAlreadyTaken | SlockNotHeld | SthreadHoldingLocks | SnotInInterrupt

channel Sreport : SReport

process Scheduler ≡ begin
A.2.1 Threads

ThreadInfo

\[
\begin{align*}
\text{threadClass} &: \text{ThreadID} \rightarrow \text{ClassID} \\
\text{threadMethod} &: \text{ThreadID} \rightarrow \text{MethodID} \\
\text{threadArgs} &: \text{ThreadID} \rightarrow \text{seq Word} \\
\text{basePriority} &: \text{ThreadID} \rightarrow \text{Priority} \\
\text{currentPriority} &: \text{ThreadID} \rightarrow \text{Priority}
\end{align*}
\]

\[
\begin{align*}
\text{dom threadClass} &= \text{dom threadMethod} = \text{dom threadArgs} = \\
\text{dom currentPriority} &= \text{dom basePriority} \\
\forall t : \text{dom currentPriority} \cdot \text{currentPriority}_t \geq \text{basePriority}_t
\end{align*}
\]

\[\text{PreserveThreadInfo} == \Xi \text{ThreadInfo} \setminus (\text{currentPriority})\]

\[\Delta \text{ThreadInfo} \]

\[
\begin{align*}
\text{thread} &: \text{ThreadID} \\
\text{threadClass}' &= \{\text{thread}\} \triangleleft \text{threadClass} \\
\text{threadMethod}' &= \{\text{thread}\} \triangleleft \text{threadMethod} \\
\text{threadArgs}' &= \{\text{thread}\} \triangleleft \text{threadArgs} \\
\text{basePriority}' &= \{\text{thread}\} \triangleleft \text{basePriority} \\
\text{currentPriority}' &= \{\text{thread}\} \triangleleft \text{currentPriority}
\end{align*}
\]

\[\text{ThreadManager} \]

\[
\begin{align*}
\text{free}, \text{created}, \text{started}, \text{blocked} &: \mathbb{P} \text{ThreadID} \\
\text{current} &: \text{ThreadID} \\
\langle \text{free}, \text{created}, \text{started}, \text{blocked}, \{\text{current}, \text{idle}\} \rangle \text{ partition } \text{ThreadID} \\
\text{main} &\in \text{started} \cup \text{blocked} \cup \{\text{current}\}
\end{align*}
\]

\[\text{ThreadManagerInit} \]

\[\text{ThreadManager'} \]

\[
\begin{align*}
\text{free}' &= \text{ThreadID} \setminus \{\text{idle, main}\} \\
\text{current}' &= \text{main} \\
\text{created}' &= \text{started}' = \text{blocked}' = \emptyset
\end{align*}
\]

\[\text{ChangeFreeCreated} == \Xi \text{ThreadManager} \setminus (\text{free, created})\]
\[\text{ChangeFreeStarted} == \Xi \text{ThreadManager} \setminus (\text{free, started})\]
\[\text{ChangeCurrentFree} == \Xi \text{ThreadManager} \setminus (\text{current, free})\]
\[\text{ChangeCreatedStarted} == \Xi \text{ThreadManager} \setminus (\text{created, started})\]
\[\text{ChangeStartedCurrent} == \Xi \text{ThreadManager} \setminus (\text{started, current})\]
\[\text{ChangeCurrentBlocked} == \Xi \text{ThreadManager} \setminus (\text{current, blocked})\]
\[\text{ChangeBlockedStarted} == \Xi \text{ThreadManager} \setminus (\text{blocked, started})\]
A.2.2 Priority Scheduler

Queue == iseq ThreadID

pushFront, pushBack : ThreadID → Queue → Queue

∀ thread : ThreadID; queue : Queue •

pushFront thread queue = queue ^ ⟨thread⟩ ∧
pushBack thread queue = ⟨thread⟩ ^ queue

queueFront : Queue → ThreadID
removeFromQueue : ThreadID → Queue → Queue

queueFront = last

∀ thread : ThreadID; queue : Queue •
removeFromQueue thread queue = queue | \{thread\}

Scheduler

ThreadInfo
ThreadManager
priorityQueues : Priority → Queue
interruptThreads : IF ThreadID

∪{ q : ran priorityQueues • ran q } = (started ∪ \{current\}) \ \{idle\}
disjoint (λ p : Priority • ran (priorityQueues p))

∀ p : Priority • ∀ t : ran (priorityQueues p) • currentPriority t = p
dom currentPriority = created ∪ started ∪ blocked ∪ \{current, idle\}
interruptThreads ⊆ started ∪ \{current\}

updatePriorityQueue

: Priority → (Queue → Queue) → (Priority → Queue) → (Priority → Queue)

∀ priority : Priority; f : Queue → Queue; pqs : Priority → Queue •
updatePriorityQueue priority f pqs = pqs ⊕ \{priority → f (pqs priority)\}

SchedulerInit

Scheduler'
ThreadManagerInit
mainPriority? : ThreadPriority

currentPriority' = basePriority' = \{main ↦ mainPriority?, idle ↦ minSwPriority\}
priorityQueues' mainPriority? = \{main\}
∀ p : Priority | p ≠ mainPriority? • priorityQueues' p = \{
interruptThreads' = ∅
ThreadCreate
\[ \Delta \text{Scheduler} \]
\[ \text{ChangeFreeCreated} \]
\[ \text{priority}\? : \text{ThreadPriority} \]
\[ \text{class}\! : \text{ClassID} \]
\[ \text{method}\? : \text{MethodID} \]
\[ \text{args}\? : \text{seq Word} \]
\[ \text{newID}\! : \text{ThreadID} \]
\[ \text{newID}\! \in \text{free} \]
\[ \text{threadClass}’ = \text{threadClass} \oplus \{ \text{newID}\! \mapsto \text{class}\? \} \]
\[ \text{threadMethod}’ = \text{threadMethod} \oplus \{ \text{newID}\! \mapsto \text{method}\? \} \]
\[ \text{threadArgs}’ = \text{threadArgs} \oplus \{ \text{newID}\! \mapsto \text{args}\? \} \]
\[ \text{currentPriority}’ = \text{currentPriority} \oplus \{ \text{newID}\! \mapsto \text{priority}\? \} \]
\[ \text{basePriority}’ = \text{basePriority} \oplus \{ \text{newID}\! \mapsto \text{priority}\? \} \]
\[ \text{free}’ = \text{free} \setminus \{ \text{newID}\! \} \]
\[ \text{created}’ = \text{created} \cup \{ \text{newID}\! \} \]
\[ \text{interruptThreads}’ = \text{interruptThreads} \land \text{priorityQueues}’ = \text{priorityQueues} \]

PickNewCurrent
\[ \Delta \text{Scheduler} \]
\[ \text{previous}\! : \text{ThreadID} \]
\[ \text{current}’ = \text{queueFront} (\text{pushBack} \text{ idle} (\lnot/ \text{priorityQueues}’)) \]
\[ \text{started}’ = (\text{started} \cup \{ \text{current} \}) \setminus \{ \text{current}’, \text{idle} \} \]
\[ \text{previous}! = \text{current} \]

ThreadStarts
\[ \Delta \text{Scheduler} \]
\[ \text{ChangeCreatedStarted} \]
\[ \exists \text{ThreadInfo} \]
\[ \text{toStart}\? : \text{F ThreadID} \]
\[ \text{previous}\! : \text{ThreadID} \]
\[ \text{toStart}\? \subseteq \text{created} \]
\[ \text{created}’ = \text{created} \setminus \text{toStart}\? \]
\[ \forall t : \text{toStart}\? \bullet \text{priorityQueues}’ = \]
\[ \text{updatePriorityQueue} (\text{currentPriority} t) (\text{pushBack} t) \text{priorityQueues} \]
\[ \exists \text{started}0 == \text{started} \cup \text{toStart}\? \bullet \text{PickNewCurrent}[\text{started}0/\text{started}] \]
\[ \text{interruptThreads}’ = \text{interruptThreads} \]
ThreadDestroy

\[ \Delta \text{Scheduler} \]
\[ \text{RemoveThreadInfo} \]
\[ \text{thread}?: \text{ThreadID} \]
\[ \text{previous}!: \text{ThreadID} \]

\[ \text{thread}? \in \text{blocked} \cup \text{started} \cup \{\text{current}\} \]
\[ \text{thread}\notin\{\text{idle, main}\} \]
\[ \text{free}' = \text{free} \cup \{\text{thread}\?}\] 
\[ \text{created}' = \text{created} \]
\[ \text{blocked}' = \text{blocked} \setminus \{\text{thread}\?\} \]
\[ \exists \text{priority} = \text{currentPriority} \text{ thread}?' \]
\[ \quad \text{priorityQueues}' = \]
\[ \quad \text{updatePriorityQueue priority (removeFromQueue thread)?) priorityQueues} \]
\[ \exists \text{started}0 = \text{started} \setminus \{\text{thread}\?\} \quad \text{PickNewCurrent[started0]/started} \]
\[ \text{interruptThreads}' = \text{interruptThreads} \]

ThreadSuspend

\[ \Delta \text{Scheduler} \]
\[ \Xi \text{ThreadInfo} \]
\[ \text{toSuspend}?: \text{ThreadID} \]
\[ \text{previous}!: \text{ThreadID} \]

\[ \text{toSuspend}? \in \{\text{current}\} \cup \text{started} \]
\[ \text{toSuspend}\notin\{\text{idle}\} \cup \text{interruptThreads} \]
\[ \text{blocked}' = \text{blocked} \cup \{\text{current}\} \]
\[ \exists \text{priority} = \text{currentPriority} \text{ toSuspend}? \]
\[ \quad \text{priorityQueues}' = \]
\[ \quad \text{updatePriorityQueue priority (removeFromQueue toSuspend)?) priorityQueues} \]
\[ \exists \text{started}0 = \text{started} \setminus \{\text{toSuspend}\?\} \quad \text{PickNewCurrent[started0]/started} \]
\[ \text{free}' = \text{free} \land \text{created}' = \text{created} \]
\[ \text{interruptThreads}' = \text{interruptThreads} \]

ThreadResume

\[ \Delta \text{Scheduler} \]
\[ \text{ChangeBlockedStarted} \]
\[ \Xi \text{ThreadInfo} \]
\[ \text{thread}?: \text{ThreadID} \]
\[ \text{previous}!: \text{ThreadID} \]

\[ \text{thread}? \in \text{blocked} \]
\[ \text{blocked}' = \text{blocked} \setminus \{\text{thread}\?\} \]
\[ \text{priorityQueues}' = \]
\[ \text{updatePriorityQueue (currentPriority thread)?) (pushBack thread)?) priorityQueues} \]
\[ \exists \text{started}0 = \text{started} \cup \{\text{thread}\?\} \quad \text{PickNewCurrent[started0]/started} \]
\[ \text{interruptThreads}' = \text{interruptThreads} \]

SSSuccess

\[ \text{report}!: \text{SReport} \]
\[ \text{report}! = \text{Sokay} \]
<table>
<thead>
<tr>
<th><strong>NonexistentThread</strong></th>
</tr>
</thead>
</table>
| \[ \exists \text{Scheduler} \]
| \[ \text{thread} ? : \text{ThreadID} \] |
| \[ \text{report} ! : \text{SReport} \] |
| \[ \text{thread} ? \in \text{free} \] |
| \[ \text{report} ! = \text{SnonexistentThread} \] |

<table>
<thead>
<tr>
<th><strong>NonexistentThreads</strong></th>
</tr>
</thead>
</table>
| \[ \exists \text{Scheduler} \]
| \[ \text{toStart} ? : F \text{ThreadID} \] |
| \[ \text{report} ! : \text{SReport} \] |
| \[ \text{toStart} ? \cap \text{free} \neq \emptyset \] |
| \[ \text{report} ! = \text{SnonexistentThread} \] |

<table>
<thead>
<tr>
<th><strong>ThreadAlreadyStarted</strong></th>
</tr>
</thead>
</table>
| \[ \exists \text{Scheduler} \]
| \[ \text{toStart} ? : F \text{ThreadID} \] |
| \[ \text{report} ! : \text{SReport} \] |
| \[ \text{toStart} ? \cap (\text{started} \cup \text{blocked} \cup \{\text{current}, \text{idle}\}) \neq \emptyset \] |
| \[ \text{report} ! = \text{SthreadAlreadyStarted} \] |

<table>
<thead>
<tr>
<th><strong>ThreadNotDestroyable</strong></th>
</tr>
</thead>
</table>
| \[ \exists \text{Scheduler} \]
| \[ \text{thread} ? : \text{ThreadID} \] |
| \[ \text{report} ! : \text{SReport} \] |
| \[ \text{thread} ? \in \{\text{main}, \text{idle}, \text{current}\} \cup \text{interruptThreads} \] |
| \[ \text{report} ! = \text{SthreadNotDestroyable} \] |

<table>
<thead>
<tr>
<th><strong>ThreadNotBlocked</strong></th>
</tr>
</thead>
</table>
| \[ \exists \text{Scheduler} \]
| \[ \text{thread} ? : \text{ThreadID} \] |
| \[ \text{report} ! : \text{SReport} \] |
| \[ \text{thread} ? \in \text{created} \cup \text{started} \cup \{\text{current}, \text{idle}\} \] |
| \[ \text{report} ! = \text{SthreadNotBlocked} \] |

<table>
<thead>
<tr>
<th><strong>ThreadNotBlockable</strong></th>
</tr>
</thead>
</table>
| \[ \exists \text{Scheduler} \]
| \[ \text{report} ! : \text{SReport} \] |
| \[ \text{current} \in \{\text{idle}\} \cup \text{interruptThreads} \] |
| \[ \text{report} ! = \text{SthreadNotBlockable} \] |

\[
R\text{ThreadCreate} \equiv (\text{ThreadCreate} \land \text{SSuccess})
\]
\[
R\text{ThreadStarts} \equiv (\text{ThreadStarts} \land \text{SSuccess}) \lor \text{NonexistentThreads} \lor \text{ThreadAlreadyStarted}
\]
\[
R\text{ThreadDestroy} \equiv 
(\text{ThreadDestroy} \land \text{SSuccess}) \lor \text{NonexistentThread} \lor \text{ThreadNotDestroyable}
\]
\[
R\text{ThreadSuspend} \equiv (\text{ThreadSuspend} \land \text{SSuccess}) \lor \text{ThreadNotBlockable}
\]
\[
R\text{ThreadResume} \equiv (\text{ThreadResume} \land \text{SSuccess}) \lor \text{NonexistentThread} \lor \text{ThreadNotBlocked}
\]
### A.2.3 Priority Ceiling Emulation

**PCEScheduler**

<table>
<thead>
<tr>
<th>PCEScheduler</th>
<th>Scheduler</th>
</tr>
</thead>
<tbody>
<tr>
<td>priorityCeiling: ObjectID → Priority</td>
<td></td>
</tr>
<tr>
<td>lockHolder: ObjectID → ThreadID</td>
<td></td>
</tr>
<tr>
<td>lockCount: ObjectID → ( \mathbb{N}_1 )</td>
<td></td>
</tr>
<tr>
<td>locksHeld: ThreadID → ( \mathbb{P} ) ObjectID</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{dom} \ lockCount = \text{dom} \ lockHolder \\
\text{ran} \ lockHolder \subseteq \text{started} \cup \{ \text{current} \} \\
\forall t: \text{ThreadID} \bullet \text{locksHeld} t = \text{lockHolder} \sim \{ t \} \\
\forall t: \text{ThreadID} \bullet \\
\quad \text{currentPriority} t = \\
\quad \quad \max \{ \{ \text{basePriority} t \} \cup \{ o: \text{locksHeld} t \bullet \text{priorityCeiling} o \} \}
\]

**PCESchedulerInit**

<table>
<thead>
<tr>
<th>PCESchedulerInit</th>
<th>PCEScheduler'</th>
</tr>
</thead>
<tbody>
<tr>
<td>SchedulerInit</td>
<td>SchedulerInit</td>
</tr>
</tbody>
</table>

\[
\forall x: \text{ObjectID} \bullet \text{priorityCeiling}' x = \text{maxSwPriority} \\
\text{lockHolder}' = \emptyset \\
\text{lockCount}' = \emptyset
\]

**PCETakeLock**

\[ \Delta \text{PCEScheduler} \]

<table>
<thead>
<tr>
<th>ChangeStartedCurrent</th>
<th>PreserveThreadInfo</th>
</tr>
</thead>
<tbody>
<tr>
<td>PickNewCurrent</td>
<td>object? : ObjectID</td>
</tr>
<tr>
<td>thread? : ThreadID</td>
<td></td>
</tr>
</tbody>
</table>

\[
\text{object?} \notin \text{dom} \ lockHolder \\
\text{currentPriority} \ current \leq \ \text{priorityCeiling} \ object? \\
\text{lockHolder}' = \text{lockHolder} \oplus \{ \text{object?} \rightarrow \text{current} \} \\
\text{lockCount}' = \text{lockCount} \oplus \{ \text{object?} \rightarrow 1 \} \\
\text{currentPriority}' = \text{currentPriority} \oplus \\
\quad \{ \text{current} \rightarrow \max \{ \text{currentPriority} \ current, \text{priorityCeiling} \ object? \} \} \\
\text{priorityQueues}' = \\
\quad \{ \text{updatePriorityQueue} (\text{currentPriority}' \ current) \} \ (\text{removeFromQueue} \ current) \} \\
\quad \{ \text{updatePriorityQueue} (\text{currentPriority}' \ current) \} \ (\text{pushFront} \ current) \} \text{priorityQueues} \\
\text{priorityCeiling}' = \text{priorityCeiling} \land \text{interruptThreads}' = \text{interruptThreads}
\]

153
PCERetakeLock

\[ \Delta \text{PCEScheduler} \]

\[ \Xi \text{ThreadManager} \]

\[ \Xi \text{ThreadInfo} \]

\[ \text{object?} : \text{ObjectID} \]

\[ \text{previous!} : \text{ThreadID} \]

\[ \text{object?} \in \text{dom lockHolder} \]

\[ \text{lockHolder object?} = \text{current} \]

\[ \text{lockCount}' \text{ object?} = \text{lockCount object?} + 1 \]

\[ \{ \text{object?} \} \ll \text{lockCount}' = \{ \text{object?} \} \ll \text{lockCount} \]

\[ \text{previous!} = \text{current} \]

\[ \text{priorityCeiling}' = \text{priorityCeiling} \]

\[ \text{lockHolder}' = \text{lockHolder} \land \text{locksHeld}' = \text{locksHeld} \]

\[ \text{priorityQueues}' = \text{priorityQueues} \land \text{interruptThreads}' = \text{interruptThreads} \]

PCEReleaseLock

\[ \Delta \text{PCEScheduler} \]

\[ \text{ChangeStartedCurrent} \]

\[ \text{PreserveThreadInfo} \]

\[ \text{PickNewCurrent} \]

\[ \text{object?} : \text{ObjectID} \]

\[ \text{object?} \in \text{locksHeld current} \]

\[ \text{lockCount object?} = 1 \]

\[ \text{lockHolder}' = \{ \text{object?} \} \ll \text{lockHolder} \]

\[ \text{lockCount}' = \{ \text{object?} \} \ll \text{lockCount} \]

\[ \text{currentPriority}' = \text{currentPriority} \oplus \{ \text{current} \mapsto \text{max} \}
  \quad \{ \{ \text{o} : \text{locksHeld}' \text{ current} \land \text{priorityCeiling o} \} \cup \{ \text{basePriority current} \}) \}
\]

\[ \text{priorityQueues}' = \]

\[ \quad \{ \text{updatePriorityQueue (currentPriority current) (removeFromQueue current)}; \]

\[ \quad \text{updatePriorityQueue (currentPriority' current) (pushFront current)} \} \text{ priorityQueues} \]

\[ \text{priorityCeiling}' = \text{priorityCeiling} \land \text{interruptThreads}' = \text{interruptThreads} \]

PCEReleaseNestedLock

\[ \Delta \text{PCEScheduler} \]

\[ \Xi \text{Scheduler} \]

\[ \text{object?} : \text{ObjectID} \]

\[ \text{previous!} : \text{ThreadID} \]

\[ \text{object?} \in \text{locksHeld current} \]

\[ \text{lockCount object?} > 1 \]

\[ \text{lockCount}' \text{ object?} = \text{lockCount object?} - 1 \]

\[ \{ \text{object?} \} \ll \text{lockCount}' = \{ \text{object?} \} \ll \text{lockCount} \]

\[ \text{previous!} = \text{current} \]

\[ \text{lockHolder}' = \text{lockHolder} \land \text{locksHeld}' = \text{locksHeld} \]

\[ \text{priorityCeiling}' = \text{priorityCeiling} \]
A.2.4 Interrupt Handling
InterruptScheduler

PCEScheduler
interruptPriority : InterruptID → InterruptPriority
interruptHandler : InterruptID → ClassID × ObjectID
interruptAC : InterruptID → BackingStoreID
interruptStack : InterruptID → StackID
maskedInterrupts : ℙ InterruptID
interruptsEnabled : ℐ
interruptThreadMap : InterruptID → ThreadID

\[ \text{dom interruptHandler} = \text{dom interruptAC} = \text{dom interruptStack} \]
\[ \text{dom interruptThreadMap} \subseteq \text{maskedInterrupts} \]
\[ \text{ran interruptThreadMap} = \text{interruptThreads} \]
\[ \text{interruptThreadMap} \uplus \text{basePriority} \subseteq \text{interruptPriority} \]

InterruptSchedulerInit

\[ \text{InterruptScheduler}' = \text{PCESchedulerInit} \]

\[ \text{interruptHandler}' = \emptyset \]
\[ \text{interruptAC}' = \emptyset \]
\[ \text{interruptStack}' = \emptyset \]
\[ \text{interruptThreadMap}' = \emptyset \]
\[ \text{maskedInterrupts}' = \emptyset \]
\[ \text{interruptsEnabled}' = \text{True} \]

InterruptAttachHandler

\[ \Delta \text{InterruptScheduler} \]
\[ \Xi \text{PCEScheduler} \]
interrupt? : InterruptID
handlerClass? : ClassID
handlerObject? : ObjectID
ac? : BackingStoreID
stack? : StackID

\[ \text{interruptHandler}' = \]
\[ \quad \text{interruptHandler} \uplus \{ \text{interrupt}? \mapsto (\text{handlerClass}?, \text{handlerObject}?) \} \]
\[ \text{interruptAC}' = \text{interruptAC} \uplus \{ \text{interrupt}? \mapsto \text{ac}? \} \]
\[ \text{interruptStack}' = \text{interruptStack} \uplus \{ \text{interrupt}? \mapsto \text{stack}? \} \]
\[ \text{interruptPriority}' = \text{interruptPriority} \]
\[ \text{maskedInterrupts}' = \text{maskedInterrupts} \]
\[ \text{interruptThreadMap}' = \text{interruptThreadMap} \]
\[ \text{interruptsEnabled}' = \text{interruptsEnabled} \]
InterruptDetachHandler

\[ \Delta \text{InterruptScheduler} \]
\[ \Xi \text{PCEScheduler} \]
\[ \text{interrupt? : InterruptID} \]
\[ \text{interruptHandler'} = \{ \text{interrupt?} \} \leq \text{interruptHandler} \]
\[ \text{interruptAC'} = \{ \text{interrupt?} \} \leq \text{interruptAC} \]
\[ \text{interruptStack'} = \{ \text{interrupt?} \} \leq \text{interruptStack} \]
\[ \text{interruptPriority'} = \text{interruptPriority} \]
\[ \text{maskedInterrupts'} = \text{maskedInterrupts} \]
\[ \text{interruptThreadMap'} = \text{interruptThreadMap} \]
\[ \text{interruptsEnabled'} = \text{interruptsEnabled} \]

InterruptGetPriority

\[ \Xi \text{InterruptScheduler} \]
\[ \text{interrupt? : InterruptID} \]
\[ \text{priority! : InterruptPriority} \]
\[ \text{priority! = interruptPriority interrupt?} \]

InterruptEnableFixedVars \(==\) \(\Xi\text{InterruptScheduler}\ \backslash\ (\text{interruptsEnabled})\)

InterruptEnable

\[ \Delta \text{InterruptScheduler} \]
\[ \text{InterruptEnableFixedVars} \]
\[ \text{interruptsEnabled'} = \text{True} \]

InterruptDisable

\[ \Delta \text{InterruptScheduler} \]
\[ \text{InterruptEnableFixedVars} \]
\[ \text{interruptsEnabled'} = \text{False} \]
\( \text{HandleInterrupt} \)

\[ \Delta \text{InterruptScheduler} \]

\( \text{ChangeFreeStarted} \)

\( \text{interrupt}?: \text{InterruptID} \)

\( \text{handler}!: \text{ThreadId} \)

\( \text{ac}!: \text{BackingStoreID} \)

\( \text{stack}!: \text{StackID} \)

\( \text{handled}!: \mathbb{B} \)

\( \text{previous}!: \text{ThreadId} \)

\( \text{interruptsEnabled} = \text{True} \)

\( \text{interrupt}? \notin \text{maskedInterrupts} \)

\( \text{interrupt}? \in \text{dom} \text{interruptHandler} \)

\( \text{handler}! \in \text{free} \)

\( \text{free}' = \text{free} \setminus \{\text{handler}!\} \)

\( \exists \text{started}0 = \text{started} \cup \{\text{handler}!\} \bullet \text{PickNewCurrent}[	ext{started0}/\text{started}] \)

\( \text{currentPriority}' = \text{currentPriority} \oplus \{\text{handler}! \mapsto \text{interruptPriority interrupt}?\} \)

\( \text{basePriority}' = \text{basePriority} \oplus \{\text{handler}! \mapsto \text{interruptPriority interrupt}?\} \)

\( \exists \text{priority} = \text{interruptPriority interrupt}? \bullet \)

\( \text{priorityQueues}' = \text{updatePriorityQueue} \text{ priority (pushBack handler!)} \text{ priorityQueues} \)

\( \text{interruptThreads}' = \text{interruptThreads} \cup \{\text{handler}!\} \)

\( \text{maskedInterrupts}' = \)

\( \{i: \text{InterruptID} | \text{interruptPriority i} \leq \text{interruptPriority interrupt}?\} \)

\( \{\text{handler}!\} \triangleleft \text{threadClass}' = \{\text{handler}!\} \triangleleft \text{threadClass} \)

\( \{\text{handler}!\} \triangleleft \text{threadMethod}' = \{\text{handler}!\} \triangleleft \text{threadMethod} \)

\( \{\text{handler}!\} \triangleleft \text{threadArgs}' = \{\text{handler}!\} \triangleleft \text{threadArgs} \)

\( \text{interruptPriority}' = \text{interruptPriority} \)

\( \text{priorityCeiling}' = \text{priorityCeiling} \land \text{lockHolder}' = \text{lockHolder} \)

\( \text{lockCount}' = \text{lockCount} \land \text{locksHeld}' = \text{locksHeld} \)

\( \text{interruptHandler}' = \text{interruptHandler} \land \text{interruptAC}' = \text{interruptAC} \)

\( \text{ac}! = \text{interruptAC interrupt}? \)

\( \text{stack}! = \text{interruptStack interrupt}? \)

\( \text{handled}! = \text{True} \)

\( \text{IgnoreInterrupt} \)

\[ \Xi \text{InterruptScheduler} \]

\( \text{interrupt}?: \text{InterruptID} \)

\( \text{handled}!: \mathbb{B} \)

\( \text{interruptsEnabled} = \text{False} \lor \)

\( \text{interrupt}? \in \text{maskedInterrupts} \lor \)

\( \text{interrupt}? \notin \text{dom} \text{interruptHandler} \)

\( \text{handled}! = \text{False} \)
ThreadSwitchInfo ::= 
    start⟨⟨ThreadID × BackingStoreID × StackID × ClassID × MethodID × seq Word⟩⟩ | switch⟨⟨ThreadID × ThreadID⟩⟩

SwitchManager

switchQueue : seq ThreadSwitchInfo
phantomCurrent : ThreadID

∀ t : ThreadID • switch (t, t) ∉ ran switchQueue
¬ ∃ t1, t2 : ThreadID • (switch (t1, t2), switch (t2, t1)) infix switchQueue

PushThreadSwitchNormal

SwitchManager

from?, to? : ThreadID

from? ≠ to? ∧ ¬ (switch (to?, from?)) prefix switchQueue
switchQueue' = (switch (from?, to?)) ^ switchQueue
phantomCurrent' = phantomCurrent
PushThreadSwitchSelf
ΞSwitchManager
from?, to? : ThreadID
from? = to?

PushThreadSwitchReverse
∆SwitchManager
from?, to? : ThreadID

⟨switch (to?, from?)⟩ prefix switchQueue
switchQueue' = tail switchQueue
phantomCurrent' = phantomCurrent

PushThreadSwitch ==
PushThreadSwitchNormal ∧ PushThreadSwitchSelf ∧ PushThreadSwitchReverse

PushThreadStart
∆SwitchManager
thread? : ThreadID
bsid? : BackingStoreID
stack? : StackID
class? : ClassID
method? : MethodID
args? : seq Word

switchQueue' = ⟨start (thread?, bsid?, stack?, class?, method?, args?)⟩ ∩ switchQueue

PopThreadSwitch
∆SwitchManager
from! , to! : ThreadID

switchQueue ≠ ∅
head switchQueue ∈ ran switch
from! = ((switch ~) (head switchQueue)).1
to! = ((switch ~) (head switchQueue)).2
switchQueue' = tail switchQueue

PopThreadStart
∆SwitchManager
threadStartInfo!
: ThreadID × BackingStoreID × StackID × ClassID × MethodID × seq Word

switchQueue ≠ ∅ ∧ head switchQueue ∈ ran start
threadStartInfo! = (start ~) (head switchQueue)
switchQueue' = tail switchQueue
A.2.5 Scheduler Operations

\[
\text{state } \text{InterruptScheduler} \land \text{SwitchManager}
\]

\[
\text{Init} \triangleq \text{var } \text{mainAC} : \text{BackingStoreID} \bullet
\]
\[
\text{Sinit} \text{?mainPriority?mainAC} \rightarrow (\text{InterruptSchedulerInit})
\]

\[
\text{GetMaxSoftwarePriority} \triangleq
\]
\[
\text{SgetMaxSoftwarePriority?maxSwPriority} \rightarrow \text{Sreport!Sokay} \rightarrow \text{Skip}
\]

\[
\text{GetMinSoftwarePriority} \triangleq
\]
\[
\text{SgetMinSoftwarePriority?minSwPriority} \rightarrow \text{Sreport!Sokay} \rightarrow \text{Skip}
\]

\[
\text{GetNormSoftwarePriority} \triangleq
\]
\[
\text{SgetNormSoftwarePriority?normSwPriority} \rightarrow \text{Sreport!Sokay} \rightarrow \text{Skip}
\]

\[
\text{GetMaxHardwarePriority} \triangleq
\]
\[
\text{SgetMaxSoftwarePriority?maxHwPriority} \rightarrow \text{Sreport!Sokay} \rightarrow \text{Skip}
\]

\[
\text{GetMinHardwarePriority} \triangleq
\]
\[
\text{SgetMinSoftwarePriority?minHwPriority} \rightarrow \text{Sreport!Sokay} \rightarrow \text{Skip}
\]

\[
\text{GetMainThread} \triangleq \text{SgetMainThread?main} \rightarrow \text{Sreport!Sokay} \rightarrow \text{Skip}
\]

\[
\text{GetCurrentThread} \triangleq \text{SgetCurrentThread?current} \rightarrow \text{Sreport!Sokay} \rightarrow \text{Skip}
\]

\[
\text{GetInterruptPriority} \triangleq \text{var } \text{priority} : \text{Priority}; \text{report} : \text{SReport} \bullet
\]
\[
\text{SgetInterruptPriority?interrupt} \rightarrow (\text{RInterruptGetPriority});
\]
\[
\text{SgetInterruptPriorityRet!priority} \rightarrow \text{Sreport!report} \rightarrow \text{Skip}
\]

\[
\text{MakeThread} \triangleq \text{var } \text{newID} : \text{ThreadID}; \text{report} : \text{SReport} \bullet
\]
\[
\text{SmakeThread?priority?class?method?args} \rightarrow (\text{RThreadCreate});
\]
\[
\text{SmakeThreadRet!newID} \rightarrow \text{Sreport!report} \rightarrow \text{Skip}
\]

\[
\text{SetPriorityCeiling} \triangleq \text{var } \text{report} : \text{SReport} \bullet
\]
\[
\text{SsetPriorityCeiling?object?ceiling} \rightarrow (\text{RPCESetPriorityCeiling});
\]
\[
\text{Sreport!report} \rightarrow \text{Skip}
\]

\[
\text{AttachInterruptHandler} \triangleq \text{var } \text{report} : \text{SReport} \bullet
\]
\[
\text{SattachInterruptHandler?interrupt?ac?stack?handlerClass?handlerObject}
\]
\[
\rightarrow (\text{RInterruptAttachHandler});
\]
\[
\text{Sreport!report} \rightarrow \text{Skip}
\]

\[
\text{DetachInterruptHandler} \triangleq \text{var } \text{report} : \text{SReport} \bullet
\]
\[
\text{SdetachInterruptHandler?interrupt} \rightarrow (\text{RInterruptDetachHandler});
\]
\[
\text{Sreport!report} \rightarrow \text{Skip}
\]

\[
\text{HandleThreadSwitch} \triangleq \text{val } \text{report} : \text{SReport}; \text{val } \text{from}, \text{to} : \text{ThreadID} \bullet
\]
\[
\text{if } \text{report} = \text{Sokay} \rightarrow (\text{PushThreadSwitch})
\]
\[
\text{else } \text{report} \neq \text{Sokay} \rightarrow \text{Skip}
\]

161
ResumeThread ≡ var thread, previous : ThreadID; report : SReport ●
            SResumeThread?thread → (RThreadResume);
            SReport!report → HandleThreadSwitch(report, previous, current)

TakeLock ≡ var report : SReport; previous : ThreadID ●
            StakeLock?object → (RPCETakeLock[phantomCurrent/thread!?]);
            SReport!report → HandleThreadSwitch(report, previous, current)

ReleaseLock ≡ var report : SReport; previous : ThreadID ●
            SReleaseLock?object → (RPCEReleaseLock[phantomCurrent/thread!?]);
            SReport!report → HandleThreadSwitch(report, previous, current)

SuspendThread ≡ var report : SReport; previous : ThreadID ●
            SSuspendThread → (RPCESuspend[phantomCurrent/toSuspend!?]);
            SReport!report → HandleThreadSwitch(report, previous, current)

StartThreads ≡
    var threadsInfo : Φ(ThreadID × BackingStoreID × StackID) ●
    var report : SReport; previous : ThreadID ●
    SStartThreads?!ts → threadsInfo := ts;
    (∃ toStart? = {t : threadsInfo ∈ t.1} ● RThreadStarts);
    if report = Sokay → (threadinfo : threadsInfo ●
        var thread : ThreadID; bsid : BackingStoreID; stack : StackID ●
        thread, bsid, stack := threadinfo.1, threadinfo.2, threadinfo.3;
        var class : ClassID; method : MethodID; args : seq Word ●
        class, method, args := threadClass thread, threadMethod thread, threadArgs thread;
        (PushThreadStart))
        [report ≠ Sokay → Skip]
    fi;
    SReport!report → HandleThreadSwitch(report, previous, current)

EnableInterrupts ≡ var report : SReport ●
            SEnableInterrupts → (RInterruptEnable);
            HWEnableInterrupts → SReport!report → Skip

DisableInterrupts ≡ var report : SReport ●
            SDisableInterrupts → (RInterruptDisable);
            HWDdisableInterrupts → SReport!report → Skip

| handleID : MethodID

Handle ≡ val interrupt : InterruptID ●
            var handler, previous : ThreadID; ac : BackingStoreID; handled : B; stack : StackID ●
            (HandleInterrupt ∨ IgnoreInterrupt);
    if handled = True →
        var class : ClassID; method : MethodID; args : seq Word ●
        class, method, args :=
        (interruptHandler interrupt).1, handleID, ((interruptHandler interrupt).2);
        (PushThreadStart[handler/thread?, ac/bsid?])
    [handled = False → Skip]
fi
HandleNonclockInterrupt ≡
  HWinterrupt?interrupt : (interrupt ≠ clockInterrupt) → Handle(interrupt)

HandleClockInterrupt ≡ RTCclockInterrupt → Handle(clockInterrupt)

EndThread ≡ var previous : ThreadID; report : SReport ⋆
  SendThread → (∃ thread? == phantomCurrent ⋆ RInterruptEnd ∨ RThreadDestroy);
  HandleThreadSwitch(report, previous, current)

StartThread ≡
  var threadStartInfo :
    ThreadID × BackingStoreID × StackID × ClassID × MethodID × seq Word ⋆
    (switchQueue ≠ ∅ ∧ head switchQueue ∈ ran start) &
    (PopThreadStart) ; CEEstartThread!threadStartInfo → Skip

SwitchThread ≡ var from, to : ThreadID ⋆
  (switchQueue ≠ ∅ ∧ head switchQueue ∈ ran switch) &
  (PopThreadSwitch) ; CEEswitchThread!from!to → phantomCurrent := to

Proceed ≡ (switchQueue = ∅) & CEEproceed!current → Skip

Loop ≡ GetMainThread □ MakeThread □ StartThreads
□ GetCurrentThread □ SuspendThread □ ResumeThread
□ SetPriorityCeiling □ TakeLock □ ReleaseLock
□ AttachInterruptHandler □ DetachInterruptHandler
□ GetInterruptPriority □ EnableInterrupts
□ DisableInterrupts □ HandleNonclockInterrupt
□ HandleClockInterrupt □ EndThread
□ SwitchThread □ Proceed ; Loop

● Init ; Loop
end

A.3 Real-Time Clock

section realtimeclock parents scheduler

Time == N

| precision : Time |
| precision > 0 |

channel HWtime : Time
channel RTCgetTime, RTCgetPrecision : Time
channel RTCsetAlarm : Time
channel RTCclearAlarm

RTCreport ::= RTCokay | RTCtimeInPast

channel RTCreport : RTCreport

process RealtimeClock ≡ begin

| RTCState
| currentTime : Time
| currentAlarm : Time
| alarmSet : bool

alarmSet = True ⇒ currentAlarm ≥ currentTime

state RTCState

| RTCInit
| RTCState′
| initTime? : Time

currentTime′ = initTime?
alarmSet′ = False

Init ≡ HWtime?initTime → (RTCInit)

GetTime ≡ RTCgetTime!currentTime → RTCreport!RTCokay → Skip

GetPrecision ≡ RTCgetPrecision!precision → RTCreport!RTCokay → Skip

| RTCsetAlarm
\[ \Delta RTCState \]
| alarmTime? : Time
| report! : RTCreport

alarmTime? ≥ currentTime
currentTime′ = alarmTime?
alarmSet′ = True
currentTime′ = currentTime
report! = RTCokay

| RTCtimeInPast
\[ \exists RTCState \]
| alarmTime? : Time
| report! : RTCreport

alarmTime? < currentTime
report! = RTCtimeInPast

164
\begin{verbatim}
SetAlarm ⇒ var report : RTCReport •
  RTCsetAlarm?alarmTime → (RTCsetAlarm ∨ TimeInPast);
  RTCreport!report → Skip

ClearAlarm ⇒ RTCclearAlarm → alarmSet := False ; RTCreport!RTCokay → Skip

TriggerAlarm ⇒ RTCclockInterrupt → alarmSet := False

Tick ⇒ HWinterrupt?interrupt : (interrupt = clockInterrupt) → if
  currentTime + precision ≥ currentAlarm → TriggerAlarm
  currentTime + precision < currentAlarm → Skip
fi ; currentTime := currentTime + precision

Loop ⇒ SetAlarm □ ClearAlarm □ GetTime □ GetPrecision □ Tick ; Loop

• Init ; Loop
end

A.4 Complete SCJVM Services Model

section scjvmServices parents realtimeclock

channelset RTCInterface == { RTCclockInterrupt }

process VMSServices ≜ ((MemoryManager || Scheduler)
  || RTCInterface ) RealtimeClock
\end{verbatim}
Appendix B

Full Core Execution Environment Model

B.1 Classes

\[\text{section classes parent scjemservices}\]

\([\text{FieldID}]\]

\(\text{CPEntry} ::=\)
\(\text{ClassRef}\[\langle\langle\text{ClassID}\rangle\rangle|\]
\(\text{MethodRef}\[\langle\langle\text{ClassID} \times \text{MethodID}\rangle\rangle|\]
\(\text{FieldRef}\[\langle\langle\text{ClassID} \times \text{FieldID}\rangle\rangle]\)

\mid \text{methodArguments} : \text{MethodID} \rightarrow \mathbb{N}\]

\([\text{CPIndex}]\]

\mid \text{nullCPIndex} : \text{CPIndex}\]

ProgramAddress == N

\underline{\text{ClassConstantPool}}

constantPool : CPIndex \rightarrow CPEntry
\text{this, super} : CPIndex
\text{interfaces} : \mathcal{F} \text{CPIndex}

nullCPIndex \notin \text{dom constantPool}
\{\text{this}\} \cup \{\text{super}\} \setminus \{\text{nullCPIndex}\} \cup \text{interfaces} \subseteq \text{dom constantPool}
constantPool \{\text{this, super}\cup \text{interfaces}\} \subseteq \text{ran ClassRef}
ClassMethods

\begin{align*}
\text{methodEntry, methodEnd} & : \text{MethodID} \rightarrow \text{ProgramAddress} \\
\text{methodLocals, methodStackSize} & : \text{MethodID} \rightarrow \mathbb{N}
\end{align*}

\begin{align*}
\text{dom methodEntry} &= \text{dom methodEnd} = \text{dom methodLocals} = \text{dom methodStackSize} \\
\forall m : \text{dom methodEntry} & \cdot \text{methodEntry m} \leq \text{methodEnd m} \\
\forall m : \text{dom methodLocals} & \cdot \text{methodArguments m} \leq \text{methodLocals m}
\end{align*}

ClassFields

\begin{align*}
\text{fields, staticFields} & : \mathbb{F} \text{ FieldID} \\
\text{fields} \cap \text{staticFields} &= \emptyset
\end{align*}

Class \equiv ClassConstantPool \land \text{ClassMethods} \land \text{ClassFields}

\begin{align*}
\text{classOf} & : \text{Class} \rightarrow \text{CPIndex} \rightarrow \text{ClassID} \\
\text{fieldOf} & : \text{Class} \rightarrow \text{CPIndex} \rightarrow \text{FieldID} \\
\text{methodOf} & : \text{Class} \rightarrow \text{CPIndex} \rightarrow \text{MethodID}
\end{align*}

\begin{align*}
\forall c : \text{Class} & \cdot \\
\text{classOf c} &= c.\text{constantPool} \uparrow \\
(\text{ClassRef} \sim \uparrow (\text{FieldRef} \sim \uparrow \text{first}) \cup (\text{MethodRef} \sim \uparrow \text{first})) \land \\
\text{fieldOf c} &= c.\text{constantPool} \uparrow \text{FieldRef} \sim \uparrow \text{second} \land \\
\text{methodOf c} &= c.\text{constantPool} \uparrow \text{MethodRef} \sim \uparrow \text{second}
\end{align*}

\begin{align*}
\text{classRefIndices} & : \text{Class} \rightarrow \mathbb{F} \text{ CPIndex} \\
\text{fieldRefIndices} & : \text{Class} \rightarrow \mathbb{F} \text{ CPIndex} \\
\text{methodRefIndices} & : \text{Class} \rightarrow \mathbb{F} \text{ CPIndex}
\end{align*}

\begin{align*}
\forall c : \text{Class} & \cdot \\
\text{classRefIndices c} &= \text{dom}((c.\text{constantPool}) \triangleright \text{ran ClassRef}) \land \\
\text{fieldRefIndices c} &= \text{dom}((c.\text{constantPool}) \triangleright \text{ran FieldRef}) \land \\
\text{methodRefIndices c} &= \text{dom}((c.\text{constantPool}) \triangleright \text{ran MethodRef})
\end{align*}

\begin{align*}
\text{thisClassID} & : \text{Class} \rightarrow \text{ClassID} \\
\text{superClassID} & : \text{Class} \rightarrow \text{ClassID} \\
\text{interfacesClassIDs} & : \text{Class} \rightarrow \mathbb{F} \text{ ClassID}
\end{align*}

\begin{align*}
\forall c : \text{Class} & \cdot \text{thisClassID c} = \text{classOf c c.this} \\
\forall c : \text{Class} & \cdot c.\text{super} \neq \text{nullCPIndex} \cdot \\
& \quad c \in \text{dom superClassID} \land \text{superClassID c} = \text{classOf c c.super} \\
\forall c : \text{Class} & \cdot \text{interfacesClassIDs c} = \{ \text{cpid} : c.\text{interfaces} \cdot \text{classOf c cpid} \}
\end{align*}

relation( \_ \triangleright_d \_ )

\begin{align*}
\_ \triangleright_d \_ : \text{Class} \leftrightarrow \text{Class} \\
\forall c_1, c_2 : \text{Class} & \cdot c_1 \triangleright_d c_2 \iff \\
& (c_1.\text{super} \neq \text{nullCPIndex} \land \text{superClassID c_1} = \text{thisClassID c_2}) \\
& \quad \lor \text{thisClassID c_2} \in \text{interfacesClassIDs c_1}
\end{align*}
\[ \text{subclassRel} : (\text{ClassID} \rightarrow \text{Class}) \rightarrow (\text{ClassID} \leftrightarrow \text{ClassID}) \]

\[ \forall cs : \text{ClassID} \rightarrow \text{Class} \mid (\forall c : \text{dom cs} \bullet \text{thisClassID} (cs \ c) = c) \bullet \]

\[ (\forall c_1, c_2 : \text{ClassID} \bullet (c_1, c_2) \in \text{subclassRel} cs \Leftrightarrow\{c_1, c_2\} \subseteq \text{dom cs} \wedge cs \ c_1 \prec_d cs \ c_2) \]

\[ \wedge \text{subclassRel} cs = (\text{subclassRel} cs)^* \]

### B.2 Channels declarations

**section** Llchans parents sejvmservices

**channel** executeMethod : ThreadID \times \text{ClassID} \times \text{MethodID} \times \text{seq Word}

**channel** executeMethodRet : ThreadID \times \text{Word}

**channel** continueExecution : ThreadID

**channel** initMainThread : StackID

**channel** register : ThreadID \times \text{ObjectID}; \text{registerRet}

**channel** clearCurrentAC, clearCurrentACRet

**channel** suspend, suspendRet

**channel** resumeThread : ThreadID; \text{resumeThreadRet}

**channel** enterPrivateMemory : ThreadID \times \mathbb{N} \times \text{ObjectID}

**channel** executeInAreaOf : ThreadID \times ObjectID \times ObjectID

**channel** executeInOuterArea : ThreadID \times \text{ObjectID}

**channel** enterPerReleaseMemory : ThreadID \times \text{ObjectID}

**channel** output : Word

**channel** input : Word

**section** memory-chans parents classes

**channel** enterBackingStore : ThreadID \times \text{BackingStoreID}

**channel** exitBackingStore : ThreadID

**channel** exitBackingStoreRet : \text{BackingStoreID} \times \text{B}

**channel** getCurrentAC : ThreadID

**channel** getCurrentACRet : \text{BackingStoreID}
channel newObject : ThreadID × ClassID
channel newObjectRet : ObjectID
channel getClassIDOf : ObjectID × ClassID

channel getField : ObjectID × FieldID; getFieldRet : Word
channel putField : ObjectID × FieldID × Word
channel getStatic : ClassID × FieldID; getStaticRet : Word
channel putStatic : ClassID × FieldID × Word

channel addThreadMemory : ThreadID × BackingStoreID
channel removeThreadMemory : ThreadID

B.3 Object Manager

section memory parents memory_chans, classes

<table>
<thead>
<tr>
<th>Object</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields : FieldID → Word</td>
</tr>
<tr>
<td>class : Class</td>
</tr>
<tr>
<td>dom fields = class.fields</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ObjectInit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object'</td>
</tr>
<tr>
<td>class? : Class</td>
</tr>
<tr>
<td>class' = class?</td>
</tr>
<tr>
<td>fields' = λ f : class?.fields • null</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ObjectPutField</th>
</tr>
</thead>
<tbody>
<tr>
<td>∆ Object</td>
</tr>
<tr>
<td>field? : FieldID</td>
</tr>
<tr>
<td>value? : Word</td>
</tr>
<tr>
<td>field? ∈ dom fields</td>
</tr>
<tr>
<td>fields' = fields ⊕ {field? → value?}</td>
</tr>
<tr>
<td>class' = class</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>ObjectGetField</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ξ Object</td>
</tr>
<tr>
<td>field? : FieldID</td>
</tr>
<tr>
<td>value! : Word</td>
</tr>
<tr>
<td>field? ∈ dom fields</td>
</tr>
<tr>
<td>value! = fields field?</td>
</tr>
</tbody>
</table>
\[\text{sizeOfObject} : \text{Class} \rightarrow \mathbb{N}\]

\[\text{classClass} : \text{ClassID}\]

**process** \(\text{ObjMan} \equiv cs : \text{ClassID} \rightarrow \text{Class} \bullet \text{begin}\)

\[\text{StaticFieldsStructID} \triangleq \text{Uninitialised} | \text{Initialised}\langle\langle \text{ObjectID}\rangle\rangle\]

\[\text{toSet} : \text{StaticFieldsStructID} \rightarrow \mathbb{P} \text{ObjectID}\]

\[\text{toSet Uninitialised} = \{\}\]

\[\forall o : \text{ObjectID} \bullet \text{toSet} (\text{Initialised} o) = \{o\}\]

\[\text{ObjManState}\]

\[\text{objects} : \text{ObjectID} \rightarrow \text{Object}\]

\[\text{backingStoreMap} : \text{BackingStoreID} \rightarrow \mathbb{P} \text{ObjectID}\]

\[\text{backingStoreStacks} : \text{ThreadID} \rightarrow \text{seq}_1 \text{BackingStoreID}\]

\[\text{rootBS} : \text{BackingStoreID}\]

\[\text{staticClassFieldsID} : \text{StaticFieldsStructID}\]

\[\text{staticClassFields} : (\text{ClassID} \times \text{FieldID}) \rightarrow \text{Word}\]

\[\text{backingStoreMap partition dom objects} \cup \text{toSet staticClassFieldsID}\]

\[\bigcup \{t : \text{dom backingStoreStacks} \bullet \text{ran (backingStoreStacks} t)\} = \text{dom backingStoreMap}\]

\[\text{rootBS} \in \text{dom backingStoreMap}\]

\[\text{staticClassFieldsID} = \text{Uninitialised} \Rightarrow \text{staticClassFields} = \{\}\]

\[\text{toSet staticClassFieldsID} \subseteq \text{backingStoreMap rootBS}\]

\[\text{state} \text{ObjManState}\]

\[\text{ObjManInit}\]

\[\text{ObjManState'}\]

\[\text{rootBS}? : \text{BackingStoreID}\]

\[\text{objects'} = \emptyset\]

\[\text{backingStoreMap'} = \{\text{rootBS}? \mapsto \emptyset\}\]

\[\text{backingStoreStacks'} = \{\text{main} \mapsto \langle\langle \text{rootBS}?\rangle\rangle, \text{idle} \mapsto \langle\langle \text{rootBS}?\rangle\rangle\}\]

\[\text{rootBS'} = \text{rootBS}'?\]

\[\text{staticClassFieldsID} = \text{Uninitialised}\]

\[\text{staticClassFields} = \{\}\]

\[\text{Init} \triangleq \text{MMgetRootBackingStore} \rightarrow \text{MMgetRootBackingStoreRet?rootBS} \rightarrow (\text{ObjManInit})\]
\[ \text{staticFieldsSize} : \mathbb{F}(\text{ClassID} \times \text{FieldID}) \rightarrow \mathbb{N} \]

**InitStaticFields**

\[ \Delta\text{ObjManState} \]
\[ \text{objectID}? : \text{ObjectID} \]
\[ \text{staticFields}? : \mathbb{F}(\text{ClassID} \times \text{FieldID}) \]

\[ \text{staticClassFieldsID} = \text{Initialised objectID}? \]
\[ \text{staticClassFields} = \lambda x : \text{staticFields}? \cdot \text{null} \]
\[ \text{backingStoreMap}' \ 	ext{rootBS} = \text{backingStoreMap rootBS} \cup \{\text{objectID}??\} \]
\[ \{\text{rootBS}? \prec \text{backingStoreMap} = \{\text{rootBS}\} \prec \text{backingStoreMap} \]
\[ \text{objects}' = \text{objects} \]
\[ \text{backingStoreStacks}' = \text{backingStoreStacks} \]
\[ \text{rootBS}' = \text{rootBS} \]

**AllocateStaticFields**

\[ \Hateq \text{var} \text{staticFields} : \mathbb{F}(\text{ClassID} \times \text{FieldID}) \cdot \]
\[ \text{staticFields} := \bigcup \{\text{cid} : \text{dom cs} \cdot \{\text{cid}\} \times (\text{cs cid}).\text{staticFields}\}; \]
\[ \text{MMAllocateMemory}!(\text{last (backingStoreStacks main)})!(\text{staticFieldsSize staticFields}) \rightarrow \]
\[ \text{MMAllocateMemoryRet}?\text{objectID} \rightarrow \]
\[ \square \text{MMreport}?r : (r = \text{MMokay}) \rightarrow \left(\text{InitStaticFields}\right) \]
\[ \square \text{MMreport}?r : (r = \text{MMokay}) \rightarrow \text{Chaos} \]

**PromoteObject**

\[ \Delta\text{ObjManState} \]
\[ \Delta\text{Object} \]
\[ \text{objectID}? : \text{ObjectID} \]

\[ \text{objectID}? \in \text{dom objects} \]
\[ \theta \text{Object} = \text{objects objectID}? \]
\[ \text{objects}' = \text{objects} \uplus \{\text{objectID}? \mapsto \theta \text{Object}'\} \]
\[ \text{backingStoreMap}' = \text{backingStoreMap} \]
\[ \text{backingStoreStacks}' = \text{backingStoreStacks} \]
\[ \text{rootBS}' = \text{rootBS} \]

\[ \text{ObjManGetField} = = \exists \Delta\text{Object} \cdot \text{ObjectGetField} \land \text{PromoteObject} \]
\[ \text{ObjManPutField} = = \exists \Delta\text{Object} \cdot \text{ObjectPutField} \land \text{PromoteObject} \]

\[ \text{ObjManObjectInit} \]

\[ \Delta\text{ObjManState} \]
\[ \text{thread}? : \text{ThreadID} \]
\[ \text{objectID}? : \text{ObjectID} \]
\[ \text{class}? : \text{Class} \]

\[ \exists \text{Object} \mid \text{ObjectInit} \cdot \]
\[ \exists \text{objectID}? \not\in \text{dom objects} \]
\[ \text{objects}' = \text{objects} \uplus \{\text{objectID}? \mapsto \theta \text{Object}'\} \]
\[ \exists \text{currentBS} : \text{BackingStoreID} \mid \]
\[ \text{currentBS} = \text{last (backingStoreStacks thread)?} \cdot \]
\[ \text{backingStoreMap}' = \]
\[ \text{backingStoreMap} \uplus \]
\[ \{\text{currentBS} \mapsto \text{backingStoreMap currentBS} \cup \{\text{objectID}?\}\} \]
\[ \text{backingStoreStacks}' = \text{backingStoreStacks} \]
\[ \text{rootBS}' = \text{rootBS} \]
EnterBS
\[\Delta \text{ObjManState} \]
thread? : ThreadID
bsid? : BackingStoreID

\[
\begin{align*}
\text{backingStoreStacks}' \text{ thread}? &= \text{backingStoreStacks} \text{ thread}? \cap \{\text{bsid}?\} \\
\{\text{thread}?\} \triangleleft \text{backingStoreStacks}' &= \{\text{thread}?\} \triangleleft \text{backingStoreStacks} \\
\text{backingStoreMap}' &= \text{backingStoreMap} \cup (\text{dom } \text{backingStoreMap} \triangleleft \{\text{bsid}? \mapsto \emptyset\}) \\
\text{objects}' &= \text{objects} \land \text{rootBS}' = \text{rootBS}
\end{align*}
\]

ExitBS1
\[\Delta \text{ObjManState} \]
thread? : ThreadID
bsid! : BackingStoreID
clear! : B

\[
\begin{align*}
\# \text{backingStoreStacks} \text{ thread}? &\geq 2 \\
\text{bsid!} &= \text{last} (\text{backingStoreStacks} \text{ thread}?) \\
\text{backingStoreStacks}' \text{ thread}? &= \text{front} (\text{backingStoreStacks} \text{ thread}?) \\
\{\text{thread}?\} &\triangleleft \text{backingStoreStacks}' &= \{\text{thread}?\} \triangleleft \text{backingStoreStacks} \\
\exists \text{ thread} : \text{dom } \text{backingStoreStacks}' &\bullet \text{bsid!} \in \text{ran} (\text{backingStoreStacks}' \text{ thread}) \\
\text{clear!} &= \text{False} \\
\text{backingStoreMap}' &= \text{backingStoreMap} \\
\text{objects}' &= \text{objects} \\
\text{rootBS}' &= \text{rootBS}
\end{align*}
\]

ExitBS2
\[\Delta \text{ObjManState} \]
thread? : ThreadID
bsid! : BackingStoreID
clear! : B

\[
\begin{align*}
\# \text{backingStoreStacks} \text{ thread}? &\geq 2 \\
\text{bsid!} &= \text{last} (\text{backingStoreStacks} \text{ thread}?) \\
\text{backingStoreStacks}' \text{ thread}? &= \text{front} (\text{backingStoreStacks} \text{ thread}?) \\
\{\text{thread}?\} &\triangleleft \text{backingStoreStacks}' &= \{\text{thread}?\} \triangleleft \text{backingStoreStacks} \\
\neg \exists \text{ thread} : \text{dom } \text{backingStoreStacks}' &\bullet \text{bsid!} \in \text{ran} (\text{backingStoreStacks}' \text{ thread}) \\
\text{backingStoreMap}' &= \{\text{bsid!}\} \triangleleft \text{backingStoreMap} \\
\text{objects}' &= \text{backingStoreMap bsid} \triangleleft \text{objects} \\
\text{clear!} &= \text{True} \\
\text{rootBS}' &= \text{rootBS}
\end{align*}
\]

AddThread
\[\Delta \text{ObjManState} \]
thread? : ThreadID
bsid? : BackingStoreID

\[
\begin{align*}
\text{thread}? &\not\in \text{dom } \text{backingStoreStacks} \\
\text{backingStoreStacks}' &= \text{backingStoreStacks} \oplus \{\text{thread}? \mapsto \{\text{bsid}?'\}\} \\
\text{backingStoreMap}' &= \{\text{bsid}? \mapsto \emptyset\} \oplus \text{backingStoreMap} \\
\text{objects}' &= \text{objects} \\
\text{rootBS}' &= \text{rootBS}
\end{align*}
\]
RemoveThread

$$\Delta \text{ObjManState} \\ \ \ \ \ \ \ \text{toRemove} : \ \text{ThreadID}$$

$$\text{toRemove} \in \text{dom} \ \backingStoreStacks$$
$$\text{toRemove} \not= \text{idle}$$

$$\backingStoreStacks' = \{ \text{toRemove} \} \triangleleft \backingStoreStacks$$

$$\backingStoreMap' = \backingStoreMap$$
$$\text{objects}' = \text{objects}$$
$$\text{rootBS}' = \text{rootBS}$$

MakeObject $$\triangleq \text{val class} : \text{Class}; \ \text{val thread} : \text{ThreadID}; \ \text{res objectID} : \text{ObjectID} \bullet$$

$$\text{MMallocateMemory}! \{(\text{last} (\backingStoreStacks \ \text{thread}))! (\text{sizeOfObject class}) \longrightarrow \text{MMallocateMemoryRet? objectID} \longrightarrow$$

$$\text{MMreport?} r : (r = \text{Mokay}) \longrightarrow \left( \text{ObjManObjectInit} \right)$$
$$\Box \text{MMreport?} r : (r \not= \text{Mokay}) \longrightarrow \text{Chaos}$$

NewObject $$\triangleq \text{var objectID} : \text{ObjectID} \bullet$$

$$\text{newObject?} \text{thread}\text{?} \text{classID} \longrightarrow \text{MMallocateMemory}! \{(\text{last} (\backingStoreStacks \ \text{thread}))! (\text{sizeOfObject} (\text{cs classID}) \longrightarrow \text{MMallocateMemoryRet? objectID} \longrightarrow$$

$$\text{MMreport?} r : (r = \text{Mokay}) \longrightarrow \left( \exists \ \text{class?} = \text{cs classID} \bullet \text{ObjManObjectInit} \right); \ \text{newObjectRet? objectID} \longrightarrow$$

$$\Box \text{MMreport?} r : (r \not= \text{Mokay}) \longrightarrow \text{Chaos}$$

GetField $$\triangleq \text{var value} : \text{Word} \bullet$$

$$\text{getField?} \text{objectID}\text{?} \text{field} \longrightarrow \left( \text{ObjManGetField} \right); \ \text{getFieldRet!} \text{value} \longrightarrow \text{Skip}$$

PutField $$\triangleq \text{putField?} \text{objectID}\text{?} \text{field}\text{?} \text{value} \longrightarrow \left( \text{ObjManPutField} \right)$$

GetStatic $$\triangleq$$

$$\text{getStatic?} \text{classID}\text{?} \text{field} : ((\text{classID}, \text{field}) \in \text{dom} \ \text{staticClassFields}) \longrightarrow$$

$$\text{getStaticRet!} \text{staticClassFields}((\text{classID}, \text{field})) \longrightarrow \text{Skip}$$
$$\Box \text{getStatic?} \text{classID}\text{?} \text{field} : ((\text{classID}, \text{field}) \not\in \text{dom} \ \text{staticClassFields}) \longrightarrow \text{Chaos}$$

PutStatic $$\triangleq$$

$$\text{putStatic?} \text{classID}\text{?} \text{field}\text{?} \text{value} : ((\text{classID}, \text{field}) \in \text{dom} \ \text{staticClassFields}) \longrightarrow$$

$$\text{staticClassFields} := \text{staticClassFields} \oplus \{(\text{classID}, \text{field}) \rightarrow \text{value} \}$$
$$\Box \text{putStatic?} \text{classID}\text{?} \text{field}\text{?} \text{value} : ((\text{classID}, \text{field}) \in \text{dom} \ \text{staticClassFields}) \longrightarrow \text{Chaos}$$

GetClassIDOf $$\triangleq \text{getClassIDOf?} \text{oid}! (\text{thisClassID} ((\text{objects} \ \text{oid}).\text{class})) \longrightarrow \text{Skip}$$

GetCurrentAC $$\triangleq \text{getCurrentAC?} \text{thread} \longrightarrow$$

$$\text{if} \ \text{thread} \in \text{dom} \ \backingStoreStacks \wedge \backingStoreStacks \ \text{thread} \not= \emptyset \longrightarrow$$

$$\text{getCurrentACRet!} (\text{last} (\backingStoreStacks \ \text{thread})) \longrightarrow \text{Skip}$$
$$\Box \ \text{thread} \not\in \text{dom} \ \backingStoreStacks \lor \backingStoreStacks \ \text{thread} = \emptyset \longrightarrow \text{Chaos}$$
CheckMMReport \( \triangleq \) MMreport? \( r \rightarrow \) if \( r = \text{MMokay} \rightarrow \text{Skip} \) \( \) \( r \neq \text{MMokay} \rightarrow \text{Chaos} \) fi

EnterBackingStore \( \triangleq \) var bsid : BackingStoreID •
  enterBackingStore?thread?bs \( \rightarrow \) (EnterBS)

ExitBackingStore \( \triangleq \) var thread : ThreadID; bsid : BackingStoreID; clear : \text{B} •
  exitBackingStore?t \( \rightarrow \) thread := t;
  \((\text{ExitBS1} \land \text{ExitBS2})\);
  if clear = \text{True} \rightarrow \text{MMclearBackingStore!bsid} \rightarrow \text{CheckMMReport}
    \[ \] clear = \text{False} \rightarrow \text{Skip}
fi; exitBackingStoreRet!bsid!clear \( \rightarrow \) Skip

AddThreadMemory \( \triangleq \) addThreadMemory?thread?bsid \( \rightarrow \) (AddThread)

RemoveThreadMemory \( \triangleq \) removeThreadMemory?toRemove \( \rightarrow \) (RemoveThread)

Loop \( \triangleq \)
  (NewObject \( \Box \) GetField \( \Box \) PutField \( \Box \) GetStatic \( \Box \) PutStatic \( \Box \) GetClassIDOf \( \Box \) EnterBackingStore \( \Box \) ExitBackingStore \( \Box \) AddThreadMemory \( \Box \) RemoveThreadMemory \( \Box \) GetCurrentAC); Loop

• Init; AllocateStaticFields; Loop
end

B.4 Stack Frames

section stack frames parents memory

_Frame
  localVariables : seq Word
  operandStack : seq Word
  storedPC : ProgramAddress
  frameClass : Class
  stackSize : \text{N}
  baseFrame : \text{B}

  # operandStack \leq stackSize
StackFrameInit

StackFrame′
numLocals? : N
storedPC? : ProgramAddress
class? : Class
baseFrame? : B
initLocals? : seq Word

# initLocals? ≤ numLocals?
initLocals? ⊆ localVariables′
# localVariables′ = numLocals?
stackSize′ = stackSize?
storedPC′ = storedPC?
frameClass′ = class?
baseFrame′ = baseFrame?

StackFrameFixedVars == ΞStackFrame \ (localVariables, operandStack)

StackFrameACONST_NULL

StackFrame′

# operandStack < stackSize
operandStack′ = operandStack \ (null)
localVariables′ = localVariables

StackFrameALOAD

StackFrame′

# operandStack < stackSize
variableIndex? < # localVariables
operandStack′ = operandStack \ (localVariables (variableIndex? + 1))
localVariables′ = localVariables

StackFrameASTORE

StackFrame′

# operandStack > 0
variableIndex? < # localVariables
operandStack′ = front operandStack
localVariables′ = localVariables ⊕ \{variableIndex? + 1 → last operandStack\}
StackFrameDUP
\[\Delta StackFrame\]
StackFrameFixedVars
\[
\begin{align*}
\# \text{operandStack} &> 0 \\
\text{operandStack}' = \text{operandStack} \neg \langle \text{last operandStack} \rangle \\
\text{localVariables}' &= \text{localVariables}
\end{align*}
\]

StackFramePush
\[\Delta StackFrame\]
StackFrameFixedVars
\[
\begin{align*}
\text{value}? : &\text{Word} \\
\text{operandStack}' &= \text{operandStack} \neg \langle \text{value}? \rangle \\
\text{localVariables}' &= \text{localVariables}
\end{align*}
\]

StackFramePop
\[\Delta StackFrame\]
StackFrameFixedVars
\[
\begin{align*}
\# \text{operandStack} &> 0 \\
\text{operandStack}' &= \text{front operandStack} \\
\text{value}! &= \text{last operandStack} \\
\text{localVariables}' &= \text{localVariables}
\end{align*}
\]

StackFramePop2 == StackFramePop[\text{value}1!/\text{value}!] \overset{?}{\rightarrow} StackFramePop[\text{value}2!/\text{value}!]

StackFrameReplace == StackFramePop[\text{popped!/\text{value}!}] \overset{?}{\rightarrow} StackFramePush[\text{toPush!/\text{value}?!]

StackFrameIADD ==
\[\exists \text{value}1!, \text{value}2! : \text{Word} \bullet \text{StackFramePop2} \overset{?}{\rightarrow} \exists \text{value}? == \text{value}1! + \text{value}2! \bullet \text{StackFramePush}\]

StackFrameINEG ==
\[\exists \text{value}! : \text{Word} \bullet \text{StackFramePop} \overset{?}{\rightarrow} \exists \text{value}? == - \text{value}! \bullet \text{StackFramePush}\]

StackFrameInvoke
\[\Delta StackFrame\]
argsToPop?: N
pc?: ProgramAddress
poppedArgs?: seq Word
\[
\begin{align*}
\text{argsToPop} &\leq \# \text{operandStack} \\
\text{operandStack}' &= (1 \ldots \# \text{operandStack} - \text{argsToPop}) \lt \text{operandStack} \\
poppedArgs! &= \\
\lambda n : 1 \ldots \text{argsToPop}? \bullet \text{operandStack} \neg \langle \# \text{operandStack} - \text{argsToPop}? + n \rangle \\
\text{storedPC}' &= \text{pc}? \\
\text{localVariables}' &= \text{localVariables} \\
\text{stackSize}' &= \text{stackSize} \\
\text{frameClass}' &= \text{frameClass}
\end{align*}
\]
B.5 Interpreter

Bytecode ::= aconst_null | dup | areturn | return | iadd | ineg
| new{CPIndex} | iconst{N}
| aload{N} | astore{N}
| getfield{CPIndex} | putfield{CPIndex}
| getstatic{CPIndex} | putstatic{CPIndex}
| invokespecial{CPIndex} | invokevirtual{CPIndex}
| invokestatic{CPIndex} | if_icmple{ProgramAddress} | goto{ProgramAddress}

Stack ::= Uninitialised | Initialised{StackID}

enterPrivateMemoryID, executeInAreaOfID : MethodID
executeInOuterAreaID, enterPerReleaseMemoryID : MethodID
managedMemoryClass : ClassID
registerID : MethodID; managedSchedulableClass : ClassID

methodArguments enterPrivateMemoryID = 2
methodArguments executeInAreaOfID = 2
methodArguments executeInOuterAreaID = 1
methodArguments enterPerReleaseMemoryID = 1
methodArguments registerID = 1

clearCurrentACID, suspendID, resumeThreadID : MethodID
clearCurrentACClass, suspendClass, resumeThreadClass : ClassID

methodArguments clearCurrentACID = 0
methodArguments suspendID = 0
methodArguments resumeThreadID = 1

writeClass, readClass : ClassID
writeID, readID : MethodID

methodArguments writeID = 1

WordToThreadID : Word → ThreadID

process Thr ≡
be : ProgramAddress → Bytecode; cs : ClassID → Class; thread : ThreadID
begin
\textit{InterpreterState} \\
frameStackID : Stack  
frameStack : seq StackFrame  
\textit{pc} : ProgramAddress  
currentClass : Class  

frameStackID = Uninitialised \Rightarrow \text{frameStack} = \emptyset  
frameStack \neq \emptyset \Rightarrow \text{currentClass} = (\text{last frameStack})\text{.frameClass}  
frameStack \neq \emptyset \Rightarrow (\text{head frameStack})\text{.baseFrame} = \textbf{True}  
\exists m : \text{MethodID} | m \in \text{dom currentClass}.\text{methodEntry} \bullet  
\hspace{1cm} \text{pc} \in \text{currentClass}.\text{methodEntry} m \ldots \text{currentClass}.\text{methodEnd} m  

state \textit{InterpreterState} 

\textit{InterpreterInit} \\
\hspace{1cm} \textit{InterpreterState}'  
\hspace{2cm} \text{frameStackID}' = \text{Uninitialised}  
\hspace{2cm} \text{frameStack}' = \langle \rangle  

\textit{InterpreterNewStackFrame} \\
\Delta \textit{InterpreterState}  
\hspace{1cm} \textit{class}? : \text{Class}  
\hspace{1cm} \textit{methodID}? : \text{MethodID}  
\hspace{1cm} \textit{methodArgs}? : \text{seq Word}  
\hspace{1cm} \textit{baseFrame}? : B  
\hspace{1cm} \exists \text{numLocals}? : \mathbb{N} | \text{numLocals}? = \textit{class}?\text{.methodLocals} \textit{methodID}? \bullet  
\hspace{1cm} \exists \text{stackSize}? : \mathbb{N} | \text{stackSize}? = \textit{class}?\text{.methodStackSize} \textit{methodID}? \bullet  
\hspace{1cm} \exists \text{storedPC}? : \text{ProgramAddress} \mid \text{storedPC}? = \textit{class}?\text{.methodEntry} \textit{methodID}? \bullet  
\hspace{1cm} \exists \text{StackFrame}' \mid \text{StackFrameInit}[\text{methodArgs}]/\text{initLocals}?] \bullet  
\hspace{1cm} \text{frameStack}' = \text{frameStack} \setminus \langle \emptyset \text{StackFrame}' \rangle \land  
\hspace{1.5cm} \text{pc}' = \text{storedPC}?  
\hspace{1.5cm} \text{currentClass}' = \text{class}?  
\hspace{1.5cm} \text{frameStackID}' = \text{frameStackID}  

\textit{InterpreterReturn} \\
\Delta \textit{InterpreterState}  
\hspace{1cm} \text{fromBaseFrame}! : B  
\hspace{1cm} \# \text{frameStack} \geq 1  
\hspace{1cm} \text{frameStack}' = (\text{front frameStack})  
\hspace{1.5cm} \text{frameStack}' \neq \langle \rangle \Rightarrow  
\hspace{2.5cm} \text{pc}' = (\text{last frameStack}')\text{.storedPC} \land  
\hspace{2.5cm} \text{currentClass}' = (\text{last frameStack}')\text{.frameClass}  
\hspace{1cm} \text{fromBaseFrame}! = (\text{last frameStack}).\text{baseFrame}  
\hspace{1cm} \text{frameStackID}' = \text{frameStackID}  

178
\[\text{InterpreterAreturn}_1\]

\[\Delta \text{InterpreterState}\]
\[\text{fromBaseFrame! : } \mathbb{B}\]

\[\text{frameStack} \neq \emptyset \land (\text{last frameStack}).\text{baseFrame} = \text{False}\]

\[\exists \text{returnValue} : \text{Word} \land (\exists \Delta \text{StackFrame} \mid \text{StackFramePop}[\text{returnValue}/\text{value}]\land \theta \text{StackFrame} = \text{last frameStack}) \land (\exists \Delta \text{StackFrame} \mid \text{StackFramePush}[\text{returnValue}/\text{value}]\land \theta \text{StackFrame} = \text{last (front frameStack)} \land \text{frameStack}' = \text{front (front frameStack)} \cap (\theta \text{StackFrame'}) \land \text{pc}' = \text{last frameStack}'.\text{storedPC} \land \text{currentClass}' = \text{last frameStack}'.\text{frameClass})\]

\[\text{fromBaseFrame!} = \text{False}\]
\[\text{frameStackID'} = \text{frameStackID}\]

\[\text{InterpreterAreturn}_2\]

\[\Delta \text{InterpreterState}\]
\[\text{returnValue}! : \text{Word}\]
\[\text{fromBaseFrame! : } \mathbb{B}\]

\[\text{frameStack} \neq \emptyset \land (\text{last frameStack}).\text{baseFrame} = \text{True}\]

\[\exists \Delta \text{StackFrame} \mid \text{StackFramePop}[\text{returnValue}/\text{value}]\land \theta \text{StackFrame} = \text{last frameStack}\]

\[\text{frameStack}' = \text{front frameStack}\]
\[\text{frameStack'} \neq \emptyset \Rightarrow\]
\[\text{pc}' = \text{last frameStack}'.\text{storedPC} \land \text{currentClass}' = \text{last frameStack}'.\text{frameClass}\]
\[\text{fromBaseFrame!} = \text{True}\]
\[\text{frameStackID'} = \text{frameStackID}\]

\[\text{InterpreterAreturn} == \text{InterpreterAreturn}_1 \lor \text{InterpreterAreturn}_2\]

\[\text{InterpreterIf}_{\text{icmple}}\]

\[\Delta \text{InterpreterState}\]
\[\text{branchOffset}? : \text{ProgramAddress}\]

\[\exists \text{value1!}, \text{value2!} : \text{Word} \land (\exists \Delta \text{StackFrame} \mid \text{StackFramePop}\text{2}\land \theta \text{StackFrame} = \text{last frameStack}) \land \text{frameStack}' = \text{front frameStack} \cap (\theta \text{StackFrame'}) \land \text{pc}' = \text{pc} + \]
\[\text{if}(\text{value1!} \leq \text{value2!}) \text{ then } \text{branchOffset? else 1}\]
\[\text{currentClass}' = \text{currentClass}\]
\[\text{frameStackID'} = \text{frameStackID}\]
\begin{align*}
\text{InterpreterGoto} & \quad \Delta \text{InterpreterState} \\
& \quad \text{branchOffset}?: \text{ProgramAddress} \\
\end{align*}

\begin{align*}
\text{pc}' &= \text{pc} + \text{branchOffset}?: \\
\text{frameStack}' &= \text{frameStack} \\
\text{currentClass}' &= \text{currentClass} \\
\text{frameStackID}' &= \text{frameStackID}
\end{align*}

\begin{align*}
\text{PromoteStackFrameOp} & \quad \Delta \text{InterpreterState} \\
& \quad \Delta \text{StackFrame} \\
\theta \text{ StackFrame} &= \text{last frameStack} \\
\text{frameStack}' &= (\text{front frameStack}) \cup \{\theta \text{ StackFrame}'\} \\
\text{pc}' &= \text{pc} + 1 \\
\text{currentClass}' &= \text{currentClass} \\
\text{frameStackID}' &= \text{frameStackID}
\end{align*}

\begin{align*}
\exists \Delta \text{StackFrame} & \quad \text{StackFrameACONST\_NULL} \land \text{PromoteStackFrameOp} \\
\text{InterpreterAconst\_null} &= \exists \Delta \text{StackFrame} \quad \text{StackFrameACONST\_NULL} \land \text{PromoteStackFrameOp} \\
\text{InterpreterAload} &= \exists \Delta \text{StackFrame} \quad \text{StackFrameALOAD} \land \text{PromoteStackFrameOp} \\
\text{InterpreterAstore} &= \exists \Delta \text{StackFrame} \quad \text{StackFrameASTORE} \land \text{PromoteStackFrameOp} \\
\text{InterpreterDup} &= \exists \Delta \text{StackFrame} \quad \text{StackFrameDUP} \land \text{PromoteStackFrameOp} \\
\text{InterpreterPop} &= \exists \Delta \text{StackFrame} \quad \text{StackFramePop} \land \text{PromoteStackFrameOp} \\
\text{InterpreterPop2} &= \exists \Delta \text{StackFrame} \quad \text{StackFramePop2} \land \text{PromoteStackFrameOp} \\
\text{InterpreterPush} &= \exists \Delta \text{StackFrame} \quad \text{StackFramePush} \land \text{PromoteStackFrameOp} \\
\text{InterpreterIadd} &= \exists \Delta \text{StackFrame} \quad \text{StackFrameIADD} \land \text{PromoteStackFrameOp} \\
\text{InterpreterIneg} &= \exists \Delta \text{StackFrame} \quad \text{StackFrameINEG} \land \text{PromoteStackFrameOp} \\
\text{InterpreterStackFrameInvoke} &= \exists \Delta \text{StackFrame} \quad \text{StackFrameInvoke} \land \text{PromoteStackFrameOp}
\end{align*}

\begin{align*}
\text{ResolveMethodDirect} & \quad \text{cs?}: \text{ClassID} \mapsto \text{Class} \\
\text{classID?}: \text{ClassID} \\
\text{methodID?}: \text{MethodID} \\
\text{class!}: \text{Class} \\
\text{classID?} \in \text{dom cs?} \\
\text{methodID?} \in \text{dom(cs? classID?).methodEntry} \\
\text{class!} = \text{cs? classID?}
\end{align*}
ResolveMethodSuperclass

\[ cs? : \text{ClassID} \rightarrow \text{Class} \]
\[ \text{classID}?: \text{ClassID} \]
\[ \text{methodID}?: \text{MethodID} \]
\[ \text{class!} : \text{Class} \]
\[ \text{classID?} \in \text{dom} \text{cs}? \]
\[ \text{methodID?} \notin \text{dom}(\text{cs}? \text{classID}?).\text{methodEntry} \]
\[ \exists \text{superclasses} == \{ \text{cid} : \text{dom} \text{cs}? | (\text{classID}? , \text{cid}) \in \{ \text{c} : \text{dom} \text{cs}? | \text{c} \mapsto \text{superClassID} (\text{cs}? \text{c})\}^* \wedge \text{methodID?} \in \text{dom}(\text{cs}? \text{cid}).\text{methodEntry} \} \]
\[ \exists \text{cid} : \text{superclasses} \mid (\forall \text{c} : \text{superclasses} | (\text{cid}, \text{c}) \in \{ \text{c} : \text{dom} \text{cs}? | \text{c} \mapsto \text{superClassID} (\text{cs}? \text{c})\}^* \cdot \text{class!} = \text{cs}? \text{cid} \]

ResolveMethodSuperinterface

\[ cs? : \text{ClassID} \rightarrow \text{Class} \]
\[ \text{classID}?: \text{ClassID} \]
\[ \text{methodID}?: \text{MethodID} \]
\[ \text{class!} : \text{Class} \]
\[ \text{classID?} \in \text{dom} \text{cs}? \]
\[ \{ \text{cid} : \text{dom} \text{cs}? | (\text{classID}? , \text{cid}) \in \{ \text{c} : \text{dom} \text{cs}? | \text{c} \mapsto \text{superClassID} (\text{cs}? \text{c})\}^* \wedge \text{methodID?} \in \text{dom}(\text{cs}? \text{cid}).\text{methodEntry} \} = \emptyset \]
\[ \exists \text{superinterfaces} == \{ \text{c} : \text{dom} \text{cs}? | (\text{classID}? , \text{c}) \in \text{subclassRel} \text{cs} \wedge \text{methodID?} \in \text{dom}(\text{cs}? \text{c}).\text{methodEntry} \} \]
\[ \exists \text{cid} : \text{superinterfaces} \mid (\forall \text{i} : \text{superinterfaces} | (\text{cid}, \text{i}) \in \text{subclassRel} \text{cs}? \cdot \text{class!} = \text{cs}? \text{cid} \]

ResolveMethod == ResolveMethodDirect \lor ResolveMethodSuperclass \lor ResolveMethodSuperinterface

StartInterpreter \[\equiv\]

\[ \text{var} \quad \text{classID} : \text{ClassID}; \quad \text{methodID} : \text{MethodID}; \quad \text{methodArgs} : \text{seq Word}; \quad \text{class} : \text{Class} \quad \text{executeMethod}? : \text{(t = thread)?c?m?a \rightarrow classID, methodID, methodArgs := c, m, a;} \]
\[ (\text{ResolveMethod}[cs/cs?]) \quad (\exists \text{baseFrame}? == \text{True} \cdot \text{InterpreterNewStackFrame}) \]

HandleAconst_null \[\equiv\]
\[ (\text{bc pc} = \text{aconst_null}) \& (\text{InterpreterAconst_null}) \]

HandleDup \[\equiv\]
\[ (\text{bc pc} = \text{dup}) \& (\text{InterpreterDup}) \]

HandleIadd \[\equiv\]
\[ (\text{bc pc} = \text{iadd}) \& (\text{InterpreterIadd}) \]

HandleIneg \[\equiv\]
\[ (\text{bc pc} = \text{ineg}) \& (\text{InterpreterIneg}) \]
HandleAload ≡ (bc pc ∈ ran aload) &
  var variableIndex : N · variableIndex := (aload ~) (bc pc) ; (InterpreterAload)

HandleAstore ≡ (bc pc ∈ ran astore) &
  var variableIndex : N · variableIndex := (astore ~) (bc pc) ; (InterpreterAstore)

HandleGoto ≡ (bc pc ∈ ran goto) &
  var branchOffset : ProgramAddress · branchOffset := (goto ~) (bc pc) ; (InterpreterGoto)

HandleIconst ≡ (bc pc ∈ ran icode) &
  var value : N · value := (iconst ~) (bc pc) ; (InterpreterPush)

HandleIf_iclemple ≡ (bc pc ∈ ran if_iclemple) &
  var branchOffset : ProgramAddress ·
  branchOffset := (if_iclemple ~) (bc pc) ; (InterpreterIf_iclemple)

CheckLauncherReturn ≡ val returnValue : Word; val fromBaseFrame : B ·
  if fromBaseFrame = True ∧ (frameStack ≠ ∅ ∨ thread = main) →
    executeMethodReturn(thread!returnValue → continueExecution? (t = thread) → Skip
  fi

HandleAreturn ≡ var returnValue : Word; fromBaseFrame : B ·
  (bc pc = arreturn) & (InterpreterAreturn);
  CheckLauncherReturn(returnValue, fromBaseFrame)

HandleReturn ≡ var returnValue : Word; fromBaseFrame : B ·
  (bc pc = return) & (InterpreterReturn);
  CheckLauncherReturn(returnValue, fromBaseFrame)

HandleNew ≡ (bc pc ∈ ran new ∧ (new ~) (bc pc) ∈ classRefIndices currentClass) &
  newObject!thread!(classOf currentClass ((new ~) (bc pc))) → newObjectRet?oid → (InterpreterPush[oid/value?])

HandleGetfield ≡ (bc pc ∈ ran getfield ∧ (getfield ~) (bc pc) ∈ fieldRefIndices currentClass) &
  var oid : ObjectID · (InterpreterPop[oid/value!] \ (pc, pc'));
  getField!oid!(fieldOf currentClass ((getfield ~) (bc pc))) →
  getFieldRet?value → (InterpreterPush)

HandlePutfield ≡ (bc pc ∈ ran putfield ∧ (putfield ~) (bc pc) ∈ fieldRefIndices currentClass) &
  var oid : ObjectID; value : Word · (InterpreterPop2[value!/value1!, oid!/value2!]);
  putField!oid!(fieldOf currentClass ((putfield ~) (bc pc)))[value] → Skip

HandleGetstatic ≡ (bc pc ∈ ran getstatic ∧ (getstatic ~) (bc pc) ∈ fieldRefIndices currentClass) &
  getStatic!(classOf currentClass ((getstatic ~) (bc pc)))[fieldOf currentClass ((getstatic ~) (bc pc))] →
  getStaticRet?value → (InterpreterPush)
InvokePutstatic $\equiv$ (bc pc $\in$ ran putstatic $\land$ (putstatic $\sim$) (bc pc) $\in$ fieldRefIndices currentClass) $\&$

\[\text{var value : Word } \bullet (\text{InterpreterPop})\]

putStatic!(classOf currentClass ((putstatic $\sim$) (bc pc)))!(fieldOf currentClass ((putstatic $\sim$) (bc pc)))!value $\longrightarrow$ Skip

InvokeResumeThread $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{resumeThreadClass}) \in \text{subclassRel cs } \land \text{method} = \text{resumeThreadID} ) \&
\]

\[\text{resumeThread!(WordToThreadID (head methodArgs)) } \longrightarrow \text{resumeThreadRet } \longrightarrow \text{Skip}\]

InvokeSuspend $\equiv$

\[\text{val classID : ClassID; val method : MethodID } \bullet\]

\[(\text{classID}, \text{suspendClass}) \in \text{subclassRel cs } \land \text{method} = \text{suspendID}) \&
\]

\[\text{suspend } \longrightarrow \text{suspendRet } \longrightarrow \text{Skip}\]

InvokeRegister $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{managedSchedulableClass}) \in \text{subclassRel cs } \land \text{method} = \text{registerID}) \&
\]

\[\text{register!(head methodArgs) } \longrightarrow \text{registerRet } \longrightarrow \text{Skip}\]

InvokeEnterPrivateMemory $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{managedMemoryClass}) \in \text{subclassRel cs } \land \text{method} = \text{enterPrivateMemoryID}) \&
\]

\[\text{enterPrivateMemory!thread!(methodArgs 1)!(methodArgs 2) } \longrightarrow \text{StartInterpreter}\]

InvokeExecuteInAreaOf $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{managedMemoryClass}) \in \text{subclassRel cs } \land \text{method} = \text{executeInAreaOfID}) \&
\]

\[\text{executeInAreaOf!thread!(methodArgs 1)!(methodArgs 2) } \longrightarrow \text{StartInterpreter}\]

InvokeExecuteInOuterArea $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{managedMemoryClass}) \in \text{subclassRel cs } \land \text{method} = \text{executeInOuterAreaID}) \&
\]

\[\text{executeInOuterArea!thread!(methodArgs 1) } \longrightarrow \text{StartInterpreter}\]

InvokeEnterPerReleaseMemory $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{managedMemoryClass}) \in \text{subclassRel cs } \land \text{method} = \text{enterPerReleaseMemoryID}) \&
\]

\[\text{enterPerReleaseMemory!thread!(methodArgs 1) } \longrightarrow \text{StartInterpreter}\]

InvokeWrite $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{writeClass}) \in \text{subclassRel cs } \land \text{method} = \text{writeID}) \&
\]

\[\text{output!(methodArgs 1) } \longrightarrow \text{Skip}\]

InvokeRead $\equiv$

\[\text{val classID : ClassID; val method : MethodID; val methodArgs : seq Word } \bullet\]

\[(\text{classID}, \text{readClass}) \in \text{subclassRel cs } \land \text{method} = \text{readID}) \&
\]

\[\text{input?value } \longrightarrow \left(\text{InterpreterPush} \setminus \{\text{pc, pc'}\}\right)\]
InvokeOther ≡ 

val classID : ClassID; val methodID : MethodID; val methodArgs : seq Word; class : Class • 
((classID, suspendClass) ∉ subclassRel cs ∨ methodID ≠ suspendID) 
∧ ((classID, resumeThreadClass) ∉ subclassRel cs ∨ methodID ≠ resumeThreadID) 
∧ ((classID, managedSchedulerClass) ∉ subclassRel cs ∨ methodID ≠ registerID) 
∧ ((classID, managedMemoryClass) ∉ subclassRel cs 
∧ methodID ∉ {enterPrivateMemoryID, executeInAreaOfID, executeInOuterAreaID, enterPerReleaseMemoryID}) 
∧ ((classID, readClass) ∉ subclassRel cs ∨ methodID ≠ readID) 
∧ ((classID, writeClass) ∉ subclassRel cs ∨ methodID ≠ writeID) & 

(ResolveMethod [cs/cs?]) ; (∃ baseFrame? == False • InterpreterNewStackFrame)

Invoke ≡ val classID : ClassID; val method : MethodID; val args : seq Word •
InvokeResumeThread(classID, method, args) □ InvokeSuspend(classID, method)
  □ InvokeRegister(classID, method, args)
  □ InvokeEnterPrivateMemory(classID, method, args)
  □ InvokeExecuteInAreaOf(classID, method, args)
  □ InvokeExecuteInOuterArea(classID, method, args)
  □ InvokeEnterPerReleaseMemory(classID, method, args)
  □ InvokeWrite(classID, method, args)
  □ InvokeOther(classID, method, args)

HandleInvokevirtual ≡ var cid : ClassID; mid : MethodID; poppedArgs : seq Word •
  (bc pc ∈ ran invokevirtual ∧ (invokevirtual ~) (bc pc)) ∈ methodRefIndices currentClass)&
  mid := methodOf currentClass ((invokevirtual ~) (bc pc));
  (∃ argsToPop? == methodArguments mid + 1 • InterpreterStackFrameInvoke);
  getClassIDOF!(head poppedArgs)?cid → Invoke(cid, mid, poppedArgs)

HandleInvokespecial ≡ var cid : ClassID; mid : MethodID; poppedArgs : seq Word •
  (bc pc ∈ ran invokespecial ∧ (invokespecial ~) (bc pc)) ∈ methodRefIndices currentClass)&
  mid := methodOf currentClass ((invokespecial ~) (bc pc));
  (∃ argsToPop? == methodArguments mid + 1 • InterpreterStackFrameInvoke);
  Invoke(classOf currentClass ((invokespecial ~) (bc pc)), mid, poppedArgs)

HandleInvokestatic ≡ var cid : ClassID; mid : MethodID; poppedArgs : seq Word •
  (bc pc ∈ ran invokestatic ∧ (invokestatic ~) (bc pc)) ∈ methodRefIndices currentClass)&
  mid := methodOf currentClass ((invokestatic ~) (bc pc));
  (∃ argsToPop? == methodArguments mid • InterpreterStackFrameInvoke);
  Invoke(classOf currentClass ((invokestatic ~) (bc pc)), mid, poppedArgs)

HandleInstruction ≡
HandleAconst_null □ HandleDup □ HandleAload □ HandleAstore
  □ HandleAdd □ HandleAconst □ HandleINeg
  □ HandleGoto □ HandleIf_imple □ HandleAreturn □ HandleReturn
  □ HandleGetfield □ HandlePutfield □ HandleGetstatic □ HandlePutstatic
  □ HandleInvokevirtual □ HandleInvokespecial □ HandleInvokestatic

MainThread ≡ initMainThread?stack → frameStackID := Initialised stack ; μX •
  (StartInterpreter ; Running ; X □
    (CEEswitchThread?from?to: (from = thread) → Blocked ; X))
Blocked ≜ CEEswitchThread?from?to : (to = thread) ⟷ Skip

Poll ≜
CEEswitchThread?from?to : (from = thread) ⟷ Blocked ; Poll

CEEproceed?toProceed : (toProceed = thread) ⟷ Skip

Running ≜
if frameStack = ∅ ⟷ Skip
frameStack ≠ ∅ ⟷ HandleInstruction ; Poll ; Running
fi

NotStarted ≜
var classID : ClassID; methodID : MethodID; methodArgs : seq Word; class : Class ⋅
addThreadMemory!thread!bsid ⟷
frameStackID, classID, methodID, methodArgs := Initialised stack, cid, mid, args;
(ResolveMethod[cs/cs?]); (∃ baseFrame? == True ⋅ InterpreterNewStackFrame);
Blocked ; Running ; CEEremoveThread!thread ⟷
CEEswitchThread?from?to : (from = thread) ⟷ NotStarted

• (InterpreterInit);
  (thread = main) & MainThread
  ⋅
  (thread ≠ main) & NotStarted
end

section interpreter parents interpreter_thread

channelset ThrChans == { || }

process Interpreter ≜
bc : ProgramAddress → Bytecode; cs : ClassID → Class ⋅
|| t : ThreadID \ { idle } [ ThrChans(t)] ⋅ Thr(bc, cs, t)

B.6 Launcher

section launcher parents LIchans, memory_chans, classes

immortalMemoryClass, missionMemoryClass : ClassID
perReleaseMemoryClass, privateMemoryClass : ClassID
\textbf{initSafelet : MethodID}

\textbf{process Launcher} \triangleq \text{safeletClass} : \text{ClassID}; \ \text{initOrder} : \text{seq ClassID} \bullet \text{begin}

\text{CheckMMReport} \triangleq \text{MMreport} ? r \rightarrow \text{if } r = \text{MMokay} \rightarrow \text{Skip} \ \| \ r \neq \text{MMokay} \rightarrow \text{Chaos fi}

\text{CheckSReport} \triangleq \text{Sreport} ? r \rightarrow \text{if } r = \text{Sokay} \rightarrow \text{Skip} \ \| \ r \neq \text{Sokay} \rightarrow \text{Chaos fi}

\text{GetFreeSize} \triangleq \text{val bs} : \text{BackingStoreID}; \ \text{res sz} : \mathbb{N} \bullet
\hspace{1em} \text{MMgetTotalSize}! \text{bs} \rightarrow \text{MMgetTotalSizeRet}? \text{sz} \rightarrow \text{CheckMMReport}

\text{GetUsedSize} \triangleq \text{val bs} : \text{BackingStoreID}; \ \text{res sz} : \mathbb{N} \bullet
\hspace{1em} \text{MMgetUsedSize}! \text{bs} \rightarrow \text{MMgetUsedSizeRet}? \text{sz} \rightarrow \text{CheckMMReport}

\text{GetRootBackingStore} \triangleq \text{res rbs} : \text{BackingStoreID} \bullet
\hspace{1em} \text{MMgetRootBackingStore} \rightarrow \text{MMgetRootBackingStoreRet}? \text{rbs} \rightarrow \text{CheckMMReport}

\text{MakeBackingStore} \triangleq \text{val parentid} : \text{BackingStoreID}; \ \text{val sz} : \mathbb{N}; \ \text{val objSpace} : \mathbb{N}; \ \text{res childid} : \text{BackingStoreID} \bullet
\hspace{1em} \text{MMmakeBackingStore}! \text{parentid}! \text{sz}! \text{objSpace} \rightarrow \text{MMmakeBackingStoreRet}? \text{bsid} \rightarrow \text{childid} := \text{bsid} ; \text{CheckMMReport}

\text{ResizeBackingStore} \triangleq \hspace{1em} \text{val bsid} : \text{BackingStoreID}; \ \text{val size} : \mathbb{N} \bullet
\hspace{2em} \text{MMresizeBackingStore}! \text{bsid}! \text{size} \rightarrow \text{MMresizeBackingStoreRet}? \text{bsid} \rightarrow \text{CheckMMReport}

\text{FindBackingStore} \triangleq \text{val obj} : \text{ObjectID}; \ \text{res bsid} : \text{BackingStoreID} \bullet
\hspace{2em} \text{MMfindBackingStore}! \text{obj} \rightarrow \text{MMfindBackingStoreRet}? \text{bsid} \rightarrow \text{bsid} := b ; \text{CheckMMReport}

\text{GetRemainingBackingStore} \triangleq \text{val bsid} : \text{BackingStoreID}; \ \text{res size} : \mathbb{N} \bullet
\hspace{2em} \text{MMgetRemainingBackingStore}! \text{bsid} \rightarrow \text{MMgetRemainingBackingStoreRet}? \text{size} \rightarrow \text{size} := \text{size} ; \text{CheckMMReport}

\text{CreateStack} \triangleq \text{val size} : \mathbb{N}; \ \text{res stack} : \text{StackID} \bullet
\hspace{2em} \text{MMcreateStack}! \text{size} \rightarrow \text{MMcreateStackRet}? \text{stack} \rightarrow \text{stack} := \text{stack} ; \text{CheckMMReport}

\text{MakeThread} \triangleq \hspace{1em} \text{val priority} : \text{ThreadPriority}; \ \text{val class} : \text{ClassID}; \ \text{val method} : \text{MethodID}; \ \text{val args} : \text{seq Word}; \ \text{res thread} : \text{ThreadID} \bullet
\hspace{2em} \text{SmakeThread}! \text{priority}! \text{class}! \text{method}! \text{args} \rightarrow \text{SmakeThreadRet}? \text{t} \rightarrow \text{thread} := \text{t} ; \text{CheckSReport}

\text{SuspendThread} \triangleq \text{SsuspendThread} \rightarrow \text{CheckSReport}

\text{ResumeThread} \triangleq \text{val toResume} : \text{ThreadID} \bullet \text{SresumeThread}! \text{toResume} \rightarrow \text{CheckSReport}

\text{ExecuteMethod} \triangleq \hspace{1em} \text{val class} : \text{ClassID}; \ \text{val method} : \text{MethodID}; \ \text{val args} : \text{seq Word}; \ \text{res retval} : \text{Word} \bullet
\hspace{2em} \text{executeMethod}! \text{class}! \text{method}! \text{args} \rightarrow \text{executeMethodRet}? \text{t} := \text{t} ; \text{continueExecution}! \text{main} \rightarrow \text{retval} := \text{v}

186
ExecuteVoidMethod ≜
   val class : ClassID; val method : MethodID; val args : seq Word ●
   var void : Word ● ExecuteMethod(class, method, args, void)

MissionManager
   safelet, missionSequencer : ObjectID
currentMission : ObjectID

MemoryAreaManager
   immortalMemory : ObjectID
   missionMemory : ObjectID
   backingStores : ObjectID ↦ BackingStoreID
   perReleaseMemories : ThreadID ↦ BackingStoreID
   privateMemoryMap : BackingStoreID ↦ BackingStoreID
   immortalMemory ≠ null ⇒ immortalMemory ∈ dom backingStores
   missionMemory ≠ null ⇒ missionMemory ∈ dom backingStores
   ran perReleaseMemories ⊆ ran backingStores
   backingStores ~ (ran perReleaseMemories) ∩ \{immortalMemory, missionMemory\} = ∅
   id BackingStoreID ∩ privateMemoryMap + = ∅
   → backingStores (\{immortalMemory, missionMemory\}) ∪ ran perReleaseMemories
   ⊆ ran privateMemoryMap

SchedulerManager
   schedulableThreads : ObjectID ↦ ThreadID

state LauncherState == MissionManager ∧ MemoryAreaManager ∧ SchedulerManager

LauncherInit
   LauncherState'
   \{safelet', missionSequencer', currentMission', immortalMemory', missionMemory'\}
   ⊆ \{null\}
   backingStores' = ∅
   perReleaseMemories' = ∅
   privateMemoryMap' = ∅
   schedulableThreads' = ∅

mainStackSize : N

MakeMainStack ≜ var stack : StackID ●
   CreateStack(mainStackSize, stack); initMainThread!stack → Skip

clinit : MethodID
RunClassInitialiser ≜ val initsRemaining : seq ClassID •
     if initsRemaining ≠ ⟨⟩ →
         ExecuteVoidMethod(head initsRemaining, clinit, ⟨⟩);
         RunClassInitialiser(tail initsRemaining)
     initsRemaining = ⟨⟩ → Skip
fi
RunClassInitialisers ≜ RunClassInitialiser(initOrder)

CreateImmortalMemory ≜ var rbs : BackingStoreID •
     GetRootBackingStore(rbs);
     newObject!main!immortalMemoryClass → newObjectRet?obj
     → immortalMemory, backingStores := obj, backingStores ⊖ {obj → rbs}

CreateSafelet ≜
     newObject!main!safeletClass → newObjectRet?objectID → safelet := objectID;
     ExecuteVoidMethod(safeletClass, initSafelet, ⟨safelet⟩)

immortalMemorySize : MethodID
handleStartupError : MethodID
INSUFFICIENT_IMMORTAL_MEMORY : Word

CheckImmortalMemory ≜ var imsiz, freesz, usedsz, bool : Word; bs : BackingStoreID •
     GetFreeSize(backingStores immortalMemory, freesz);
     ExecuteMethod(safeletClass, immortalMemorySize, ⟨safelet⟩, imsiz);
     if imsiz ≤ freesz →
         GetUsedSize(backingStores immortalMemory, usedsz);
         ResizeBackingStore(backingStores immortalMemory, imsiz + usedsz, bs);
         backingStores := backingStores ⊖ {immortalMemory → bs}
     imsiz > freesz →
         ExecuteMethod(safeletClass, handleStartupError,
             ⟨safelet, INSUFFICIENT_IMMORTAL_MEMORY, imsiz − freesz⟩, bool);
     if bool = 0 → CheckImmortalMemory
     bool = 1 → Chaos
fi

globalBackingStoreSize : MethodID
INSUFFICIENT_BACKING_STORE : Word

CheckRemainingBackingStore ≜ var rbssz, gbssz, bool : Word •
     ExecuteMethod(safeletClass, globalBackingStoreSize, ⟨safelet⟩, gbssz);
     GetRemainingBackingStore(backingStores immortalMemory, rbssz);
     if gbssz ≤ rbssz → Skip
     gbssz > rbssz →
         ExecuteMethod(safeletClass, handleStartupError,
             ⟨safelet, INSUFFICIENT_BACKING_STORE, gbssz − rbssz⟩, bool);
     if bool = 0 → CheckImmortalMemory ; CheckRemainingBackingStore
     bool = 1 → Chaos
fi

fi
initializeApplication : MethodID

InitializeApplication ≜ ExecuteVoidMethod(safeletClass, initializeApplication, ⟨safelet⟩)

getSequencer : MethodID

GetSequencer ≜ ExecuteMethod(safeletClass, getSequencer, ⟨safelet⟩, missionSequencer);
  if missionSequencer = null → Chaos
  if missionSequencer ≠ null → Skip
  fi

CreateMissionMemory ≜
  var mm : ObjectID; remainingSize : N; mmbs : BackingStoreID •
  newObject!main!missionMemoryClass → newObjectRef?obj → mm := obj;
  GetRemainingBackingStore(backingStores immortalMemory, remainingSize);
  MakeBackingStore(backingStores immortalMemory, remainingSize, mmbs);
  missionMemory, backingStores := mm, backingStores ⊕ {mm → mmbs}

Startup ≜ MakeMainStack; RunClassInitialisers; CreateImmortalMemory; CreateSafelet;
    CheckImmortalMemory; CheckRemainingBackingStore; InitializeApplication;
    GetSequencer; CreateMissionMemory

getNextMission : MethodID

GetNextMission ≜
  enterBackingStore!main!(backingStores missionMemory)
  → getClassIDOf!missionSequencer?msClass
  → ExecuteMethod(msClass, getNextMission, ⟨missionSequencer⟩, currentMission)

missionMemorySize : MethodID

ResizeMissionMemory ≜ var mmSize : N; mmbs : BackingStoreID •
  getClassIDOf!currentMission?cmc →
  ExecuteMethod(cmc, missionMemorySize, ⟨currentMission⟩, mmSize);
  ResizeBackingStore(backingStores missionMemory, mmSize, mmbs);
  backingStores := backingStores ⊕ {missionMemory → mmbs}

RegisterSchedulable

ΔSchedulableManager
ΞMissionManager
ΞMemoryAreaManager
threadObj? : ObjectID
tid? : ThreadID

threadObj? ∉ dom schedulableThreads
schedulableThreads' = schedulableThreads ⊕ {threadObj? ↦ tid?}
getPriority : MethodID
run : MethodID

Register ≜ var threadObj : ObjectID; priority : Word; tid : ThreadID •
register? t : (t = main)? obj → threadObj := obj; getClassIDOf!threadObj?threadClass
→ executeMethod!main!threadClass!getPriority!(threadObj)
→ executeMethodRet? t : (t = main)? priority
→ MakeThread(priority, threadClass, run, (threadObj), tid);
( RegisterSchedulable ) ; continueExecution!main → Skip

NewPrivateMemory

<table>
<thead>
<tr>
<th>MethodAreaManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>≜ SchedulableManager</td>
</tr>
<tr>
<td>≜ MissionManager</td>
</tr>
<tr>
<td>ac? : BackingStoreID</td>
</tr>
<tr>
<td>newpm? : ObjectID</td>
</tr>
<tr>
<td>newpmbs? : BackingStoreID</td>
</tr>
</tbody>
</table>

ac? ∉ dom privateMemoryMap
backingStores' = backingStores ⊕ {newpm? → newpmbs?}
privateMemoryMap' = privateMemoryMap ⊕ {ac? → newpmbs?}
immortalMemory' = immortalMemory ∧ missionMemory' = missionMemory
perReleaseMemories' = perReleaseMemories

ResizePrivateMemory

<table>
<thead>
<tr>
<th>MethodAreaManager</th>
</tr>
</thead>
<tbody>
<tr>
<td>≜ SchedulableManager</td>
</tr>
<tr>
<td>≜ MissionManager</td>
</tr>
<tr>
<td>ac? : BackingStoreID</td>
</tr>
<tr>
<td>newpmbs? : BackingStoreID</td>
</tr>
</tbody>
</table>

if ac? ∈ dom privateMemoryMap
∃ pm : ObjectID | backingStores pm = privateMemoryMap ac? •
backingStores' = backingStores ⊕ {pm → newpmbs?}
privateMemoryMap' = privateMemoryMap ⊕ {ac? → newpmbs?}
immortalMemory' = immortalMemory ∧ missionMemory' = missionMemory
perReleaseMemories' = perReleaseMemories

EnterPrivateMemory ≜ val thread : ThreadID •
val size : N; runnable : ObjectID; ac, newpmbs : BackingStoreID •
enterPrivateMemory? t : (t = thread)? s? r → getCurrentAC!t → getCurrentACRet? a
→ thread, size, runnable, ac := t, s, r, a;
if ac ∈ dom privateMemoryMap →
ResizeBackingStore(privateMemoryMap ac, size, newpmbs);
( ResizePrivateMemory )
[ ac ∉ dom privateMemoryMap →
val rembssize : N •
GetRemainingBackingStore(ac, rembssize);
MakeBackingStore(ac, rembssize, size, newpmbs);
newObject!thread!privateMemoryClass → newObjectRet! newpm
→ ( NewPrivateMemory )
fi;
enterBackingStore!thread!newpmbs → getClassIDOf!runnable? runnableClass
→ executeMethod!thread!runnableClass!run!((runnable)) → Skip
ExecuteInAreaOf \(\triangleq\) \text{val thread : ThreadID} •
\[
\text{var bs : BackingStoreID; \ runnable, object : ObjectID} \quad \text{executeInAreaOf? : (t = thread)?obj?r \rightarrow object, runnable := obj, r;}
\text{FindBackingStore\{object, bs\};}
\text{enterBackingStore\{thread!bs \rightarrow getClassIDOf\{runnable\}?runnableClass}
\rightarrow \text{executeMethod!thread!runnableClass!run!(\{runnable\}) \rightarrow Skip}
\]

ExecuteInOuterArea \(\triangleq\) \text{val thread : ThreadID} •
\[
\text{var ac, obs : BackingStoreID; \ runnable, object : ObjectID} \quad \text{executeInOuterArea? : (t = thread)?r \rightarrow getCurrentAC\{t \rightarrow getClassIDOf\{runnable\}?runnableClass}
\rightarrow \text{executeMethod\{thread!runnableClass!run!(\{runnable\}) \rightarrow Skip}
\]

EnterPerReleaseMemory \(\triangleq\) \text{val thread : ThreadID} •
\[
\text{enterPerReleaseMemory? : (t = thread)?runnable \rightarrow}
\text{if thread \in dom \{perReleaseMemories \rightarrow}
\text{enterBackingStore\{thread!(perReleaseMemories thread)
\rightarrow getClassIDOf\{runnable\}?class}
\rightarrow \text{executeMethod\{thread!class!run!(\{runnable\}) \rightarrow Skip
\text{if}
\]

Suspend \(\triangleq\) \text{suspend \rightarrow SuspendThread ; suspendRet \rightarrow Skip}

Resume \(\triangleq\) \text{resumeThread?thread \rightarrow ResumeThread(thread) ; resumeThreadRet \rightarrow Skip}

ClearPrivateMemory
\[
\Delta \text{MemoryAreaManager}
\Xi \text{SchedulerManager}
\Xi \text{MissionManager}
toClear? : BackingStoreID

toClear? \in dom \{privateMemoryMap \Rightarrow
\quad \text{backingStores}' =
\quad \text{backingStores} \triangleright \{\text{privateMemoryMap toClear}\?\}
\text{toClear?} \notin dom \{privateMemoryMap \Rightarrow
\quad \text{backingStores}' = \text{backingStores}
\text{privateMemoryMap}' = \{\text{toClear}\?\} \triangleleft \text{privateMemoryMap}
\text{missionMemory}' = \text{missionMemory}
\]

ExitMemory \(\triangleq\) \text{val thread : ThreadID} •
\[
\text{executeMethodRet? : (t = thread)?void}
\rightarrow \text{exitBackingStore\{thread \rightarrow exitBackingStoreRet?bsid?isCleared \rightarrow}
\text{if isCleared = True \rightarrow ( ClearPrivateMemory\{bsid/toClear\?\} )}
\text{if isCleared = False \rightarrow Skip}
\text{fi ; continueExecution\{thread \rightarrow Skip}
\]
EnterMemory = val thread : ThreadID • EnterPrivateMemory(thread) □ ExecuteInAreaOf(thread) □ ExecuteInOuterArea(thread) □ EnterPerReleaseMemory(thread)

HandleSpecialMethodsMainLoop = val memoryEntries : ThreadID → N; res retVal : Word •

□ t : ThreadID •
    EnterMemory(t);
    HandleSpecialMethodsMainLoop(memoryEntries ⊕ {t ↦ memoryEntries t + 1}, retVal)
□

□ t : ThreadID •
    (memoryEntries t > 0) & ExitMemory(t);
    HandleSpecialMethodsMainLoop(memoryEntries ⊕ {t ↦ memoryEntries t - 1}, retVal)
□

((Register □ Suspend □ Resume);
    HandleSpecialMethodsMainLoop(memoryEntries, retVal))
□

((∀ t : ThreadID • memoryEntries t = 0) & executeMethodRet? thr : (thr = main)? r → continueExecution! main → retVal := r)

Since there are no entries to memory areas when a method starts executing, memoryEntries initially has every ThreadID mapped to zero. This is specified in the HandleSpecialMethodsMain action below, which behaves as HandleSpecialMethodsMainLoop with the memoryEntries parameter set to a map from each ThreadID to zero. There is a retVal result parameter, which is identified with the result parameter of HandleSpecialMethodsMainLoop.

HandleSpecialMethodsMain = res retVal : Word • HandleSpecialMethodsMainLoop(λ t : ThreadID • 0, retVal)

Additionally, since HandleSpecialMethodsMain is intended for handling special methods while executing methods on the main thread, we also provide an action to handle special methods without waiting for a method to finish. This action is called HandleSpecialMethods and it is used during the execution of the event handler threads, after the setup on the main thread has finished. We omit it here since it is similar in form to the HandleSpecialMethodsMain and HandleSpecialMethodsMain actions, but it omits the final part of the choice (the communication on executeMethodRet), since we are not waiting for a method to return, and the Register action, since that only happens on the main thread during mission setup.

HandleSpecialMethodsLoop = val memoryEntries : ThreadID → N •

□ t : ThreadID •
    EnterMemory(t);
    HandleSpecialMethodsLoop(memoryEntries ⊕ {t ↦ memoryEntries t + 1}))
□

□ t : ThreadID •
    (memoryEntries t > 0) & ExitMemory(t);
    HandleSpecialMethodsLoop(memoryEntries ⊕ {t ↦ memoryEntries t - 1}))
□

((Register □ Suspend □ Resume);
    HandleSpecialMethodsLoop(memoryEntries))

HandleSpecialMethods = HandleSpecialMethodsLoop(λ t : ThreadID • 0)
initialize : MethodID

\[\text{InitializeMission} \triangleq \text{var void : Word } \]
\[\rightarrow \text{getClassIDOf}(\text{currentMission})?\text{currentMissionClass}\]
\[\rightarrow \text{executeMethod}(\text{main!currentMissionClass}!\text{initialize}!(\text{currentMission}))\]
\[\rightarrow \text{HandleSpecialMethodsMain(void)}\]

getBSSize : MethodID
getAreaSize : MethodID
getStackSize : MethodID

\[\text{InitialiseAndStartThreads} \triangleq \]
\[\text{var threadStartInfo : F(} \text{ThreadID} \times \text{BackingStoreID} \times \text{StackID}) \bullet \text{threadStartInfo} := \{};\]
\[\left(\{t : \text{schedulableThreads} \bullet \right\}
\[\text{var threadObj : ObjectID; tid : ThreadID } \bullet \text{threadObj, tid} := t.1, t.2;\]
\[\text{var bsSize, areaSize, stackSize : N; threadClass : ClassID } \bullet \]
\[\text{var bs : BackingStoreID; stack : StackID } \bullet \]
\[\text{getClassIDOf}(\text{threadObj})?\text{class} \rightarrow \text{threadClass} := \text{class};\]
\[\text{ExecuteMethod}(\text{threadClass}, \text{getBSSize}, \langle \text{threadObj} \rangle, \text{bsSize});\]
\[\text{ExecuteMethod}(\text{threadClass}, \text{getAreaSize}, \langle \text{threadObj} \rangle, \text{areaSize});\]
\[\text{ExecuteMethod}(\text{threadClass}, \text{getStackSize}, \langle \text{threadObj} \rangle, \text{stackSize});\]
\[\text{MakeBackingStore}(\text{backingStores missionMemory}, \text{bsSize + areaSize, areaSize, bs});\]
\[\text{newObject!main!perReleaseMemoryClass} \rightarrow \text{newObjectRet}?, \text{perReleaseMemory}\]
\[\rightarrow \text{backingStores} := \text{backingStores} \oplus \{\text{perReleaseMemory} \mapsto \text{bs}\};\]
\[\text{perReleaseMemories} := \text{perReleaseMemories} \oplus \{\text{tid} \mapsto \text{bs}\};\]
\[\text{threadStartInfo} := \text{threadStartInfo} \cup \{\{\text{tid, bs, stack}\}\};\]
\[\text{SstartThreads!threadStartInfo} \rightarrow \text{CheckSReport}\]

WaitForExecution \triangleq \text{SuspendThread} ; \text{HandleSpecialMethods}

\[\text{RunNextMission} \triangleq \text{GetNextMission} ; \text{ResizeMissionMemory} ; \text{InitializeMission} ; \]
\[\text{InitialiseAndStartThreads} ; \text{WaitForExecution}\]

\[\bullet (\text{LauncherInit}) ; \text{Startup} ; \text{RunNextMission}\]
end

B.7 Complete Core Execution Environment

section complete_cee parents launcher, interpreter
channelset CodeAreaInterface ==
  { getInstruction, getInstructionRet, getClass, getClassRet, isSubclassOf }

channelset MemoryInterface ==
  { newObject, newObjectRet, getField, getFieldRet, putField, getClassOf, getClassIDOf, enterBackingStore, exitBackingStore }

channelset InterpreterInterface ==
  { interpreter, interpreterRet, continue }

channelset LauncherInterface ==
  { register, registerRet, enterPrivateMemory, executeInAreaOf, executeInOuterArea, clearCurrentAC, clearCurrentACRet, suspend, suspendRet, resumeThread, resumeThreadRet }

channelset CEEThreadInterface ==
  { CEEstartThread, CEEswitchThread, CEEremoveThread }

process LI ≜ (Launcher [ InterpreterInterface ∪ LauncherInterface ] Interpreter)
\ InterpreterInterface ∪ LauncherInterface

process LIM ≜ (LI [ MemoryInterface ∪ CEEThreadInterface ] Memory) \ MemoryInterface

process CEE ≜ (LIM [ CodeAreaInterface ] CodeArea) \ CodeAreaInterface

B.8 Post-compilation Struct Manager

section struct_manager parents classes, LIchannels

ObjectStruct ::= <classID1>Obj | ... | <classIDn>Obj

...
\[ \text{cast} <\text{classID}_1> : \text{ObjectStruct} \rightarrow <\text{classID}_1> \text{Obj} \]
\[ \vdots \]
\[ \text{cast} <\text{classID}_n> : \text{ObjectStruct} \rightarrow <\text{classID}_n> \text{Obj} \]

\[ \forall x : <\text{classID}_1> \text{Obj} \bullet \text{cast} <\text{classID}_1> (<\text{classID}_1> \text{Con} x) = x \]
\[ \vdots \]

\[ \text{sizeof} <\text{classID}_1> \text{Obj} : \mathbb{N} \]
\[ \vdots \]

\[ \text{sizeof} <\text{classID}_n> \text{Obj} : \mathbb{N} \]

\[ \text{classIDOf} : \text{ObjectStruct} \rightarrow \text{ClassID} \]

\[ \forall x : <\text{classID}_1> \text{Obj} \bullet \text{classIDOf} (<\text{classID}_1> \text{Con} x) = x.\text{classID} \]
\[ \vdots \]

\[ \forall x : <\text{classID}_n> \text{Obj} \bullet \text{classIDOf} (<\text{classID}_n> \text{Con} x) = x.\text{classID} \]

\text{StaticFields}

\[ <\text{classID}_1>._<\text{fieldID}_{1,1}> : \text{Word} \]
\[ \vdots \]

\[ <\text{classID}_1>._<\text{fieldID}_{1,l_1}> : \text{Word} \]
\[ \vdots \]

\[ <\text{classID}_n>._<\text{fieldID}_{n,1}> : \text{Word} \]
\[ \vdots \]

\[ <\text{classID}_n>._<\text{fieldID}_{n,l_n}> : \text{Word} \]

\[ \text{sizeofStaticFields} : \mathbb{N} \]

channel \text{enterBackingStore} : \text{ThreadID} \times \text{BackingStoreID}
channel \text{exitBackingStore} : \text{ThreadID}
channel \text{exitBackingStoreRet} : \text{BackingStoreID} \times \mathbb{B}

channel \text{getCurrentAC} : \text{ThreadID}
channel \text{getCurrentACRet} : \text{BackingStoreID}

channel \text{newObject} : \text{ThreadID} \times \text{ClassID}
channel \text{newObjectRet} : \text{ObjectId}
channel \text{getClassIDOf} : \text{ObjectId} \times \text{ClassID}

channel \text{getObject} : \text{ObjectId}; \text{getObjectRet} : \text{ObjectStruct}
channel \text{putObject} : \text{ObjectId} \times \text{ObjectStruct}
channel \text{getStaticFields} : \text{StaticFields}
channel \text{putStaticFields} : \text{StaticFields}
channel addThreadMemory : ThreadID × BackingStoreID
channel removeThreadMemory : ThreadID

process StructMan_c ≡ begin

StaticFieldsStructID ::= Uninitialised | Initialised[ObjectID]

toSet : StaticFieldsStructID → ℘ ObjectID
toSet Uninitialised = {}  
∀ o : ObjectID • toSet (Initialised o) = { o }

StructManState
objects : ObjectID → ObjectStruct
backingStoreMap : BackingStoreID → ℘ ObjectID
backingStoreStacks : ThreadID → seq¹ BackingStoreID
rootBS : BackingStoreID
staticClassFieldsID : StaticFieldsStructID
staticClassFields : ObjectID → StaticFields

backingStoreMap partition dom objects ∪ toSet staticClassFieldsID 
∪{ t : dom backingStoreStacks • ran (backingStoreStacks t)} = dom backingStoreMap 
rootBS ∈ dom backingStoreMap 
staticClassFieldsID = Uninitialised ⇒ staticClassFields = {} 
toSet staticClassFieldsID ⊆ backingStoreMap rootBS

state StructManState

StructManInit
StructManState'
rootBS? : BackingStoreID

objects' = ∅ 
backingStoreMap' = { rootBS? ↦ ∅ } 
backingStoreStacks' = { main ↦ { rootBS? }, idle ↦ { rootBS? } } 
rootBS' = rootBS? 
staticClassFieldsID = Uninitialised 
staticClassFields = {} 

Init ≡ MMgetRootBackingStore → MMgetRootBackingStoreRet?rootBS → (StructManInit)

staticFieldsSize : ℘(ClassID × FieldID) → ℤ
InitStaticFields

\[\Delta \text{StructManState} \]
\[\text{objectID?} : \text{ObjectID} \]

staticClassFieldsID = Initialised objectID?
staticClassFields = \{ objectID? \rightarrow
\{ <\text{classID}_1, <\text{fieldID}_1> \mapsto null, \]
\[...\]
\[<\text{classID}_n, <\text{fieldID}_n> \mapsto null \}\}

backingStoreMap' rootBS = backingStoreMap rootBS \cup \{ objectID? \}
{rootBS} \leftarrow backingStoreMap' = \{rootBS\} \leftarrow backingStoreMap
objects' = objects
backingStoreStacks' = backingStoreStacks
rootBS' = rootBS

AllocateStaticFields \[\hat{=}\]
\[\text{MMallocateMemory}! (\text{last} (\text{backingStoreStacks \ main}))! (\text{sizeofStaticFields}) \rightarrow \]
\[\text{MMallocateMemoryRet? objectID} \rightarrow \]
\[
(\text{MMreport}? r : (r = \text{MMokay}) \rightarrow (InitStaticFields) \]
\[
\square \text{MMreport}? r : (r = \text{MMokay}) \rightarrow \text{Chaos} \]

EnterBS

\[\Delta \text{StructManState} \]
\[\text{thread?} : \text{ThreadID} \]
\[\text{bsid?} : \text{BackingStoreID} \]

backingStoreStacks' thread? = backingStoreStacks thread? \cap \{bsid?\}
{thread?} \leftarrow backingStoreStacks' = \{thread?\} \leftarrow backingStoreStacks
backingStoreMap' = backingStoreMap \cup (\text{dom backingStoreMap} \leftarrow \{\text{bsid?} \mapsto \emptyset\})
objects' = objects \land rootBS' = rootBS

ExitBS

\[\Delta \text{StructManState} \]
\[\text{thread?} : \text{ThreadID} \]
\[\text{bsid!} : \text{BackingStoreID} \]
\[\text{clear!} : \text{B} \]

\# backingStoreStacks thread? \geq 2
bsid! = last (backingStoreStacks thread?)
backingStoreStacks' thread? = front (backingStoreStacks thread?)
{thread?} \leftarrow backingStoreStacks' = \{thread?\} \leftarrow backingStoreStacks
\exists \text{thread} : \text{dom backingStoreStacks'} \bullet \text{bsid!} \in \text{ran(backingStoreStacks'} \text{thread})
\text{clear!} = \text{False}
backingStoreMap' = backingStoreMap
objects' = objects
rootBS' = rootBS
ExitBS

\[\Delta \text{StructManState}\]
\[
\text{thread}?: \text{ThreadID} \\
\text{bsid}: \text{BackingStoreID} \\
\text{clear}!: \mathbb{B}
\]

\[
\#	ext{backingStoreStacks thread} \geq 2 \\
\text{bsid}! = \text{last}(\text{backingStoreStacks thread})
\]
\[
\text{backingStoreStacks}^{'}, \text{thread} = \text{front}(\text{backingStoreStacks thread})
\]
\[
\{\text{thread}\} \leftarrow \text{backingStoreStacks}' = \{\text{thread}\} \leftarrow \text{backingStoreStacks}
\]
\[
\neg \exists \text{thread} : \text{dom backinStoreStacks} \bullet \text{bsid}! \in \text{ran backinStoreStacks thread}
\]
\[
\text{backingStoreMap}' = \{\text{bsid}!\} \leftarrow \text{backingStoreMap}
\]
\[
\text{objects}' = \text{backingStoreMap bsid}! \leftarrow \text{objects}
\]
\[
\text{clear}! = \text{True}
\]
\[
\text{rootBS}' = \text{rootBS}
\]

AddThread

\[\Delta \text{StructManState}\]
\[
\text{thread}?: \text{ThreadID} \\
\text{bsid}: \text{BackingStoreID} \\
\text{thread}! \notin \text{dom backinStoreStacks}
\]
\[
\text{backingStoreStacks}' = \text{backingStoreStacks} \oplus \{\text{thread}! \mapsto (\text{bsid})\}
\]
\[
\text{backingStoreMap}' = \{\text{bsid}! \mapsto \emptyset\} \oplus \text{backingStoreMap}
\]
\[
\text{objects}' = \text{objects}
\]
\[
\text{rootBS}' = \text{rootBS}
\]

RemoveThread

\[\Delta \text{StructManState}\]
\[
\text{toRemove}?: \text{ThreadID} \\
\text{toRemove}! \in \text{dom backinStoreStacks}
\]
\[
\text{toRemove}! \neq \text{idle}
\]
\[
\text{backingStoreStacks}' = \{\text{toRemove}!\} \leftarrow \text{backingStoreStacks}
\]
\[
\text{backingStoreMap}' = \text{backingStoreMap}
\]
\[
\text{objects}' = \text{objects}
\]
\[
\text{rootBS}' = \text{rootBS}
\]
StructMan<\text{classID}_1> ObjInit

\[\Delta \text{StructMan}\]

\text{objectID}? \rightarrow \text{ObjectID}

\text{thread}? \rightarrow \text{ThreadID}

\text{objectID}? \not\in \text{dom objects}

\text{objects}' = \text{objects} \oplus \{\text{objectID}? \rightarrow \}

\text{classID} = <\text{classID}_1>,

<\text{fieldID}_{1,1}> = \text{null},

\vdots

<\text{fieldID}_{1,m_1}> = \text{null},\}

\exists \text{currentBS} : \text{BackingStoreID} | \text{currentBS} = \text{last} (\text{backingStoreStacks thread}? ) \bullet

\text{backingStoreMap}' = \text{backingStoreMap} \oplus \{\text{currentBS} \rightarrow \text{backingStoreMap currentBS} \cup \{\text{objectID}?}\}\}

\text{backingStoreStacks}' = \text{backingStoreStacks}

\text{rootBS}' = \text{rootBS}

\text{staticClassFieldsID}' = \text{staticClassFieldsID}

\text{staticClassFields}' = \text{staticClassFields}

:

StructMan<\text{classID}_n> ObjInit

\[\Delta \text{StructMan}\]

\text{objectID}? \rightarrow \text{ObjectID}

\text{thread}? \rightarrow \text{ThreadID}

\text{objectID}? \not\in \text{dom objects}

\text{objects}' = \text{objects} \oplus \{\text{objectID}? \rightarrow \}

\text{classID} = <\text{classID}_n>,

<\text{fieldID}_{n,1}> = \text{null},

\vdots

<\text{fieldID}_{n,m_n}> = \text{null},\}

\exists \text{currentBS} : \text{BackingStoreID} | \text{currentBS} = \text{last} (\text{backingStoreStacks thread}? ) \bullet

\text{backingStoreMap}' = \text{backingStoreMap} \oplus \{\text{currentBS} \rightarrow \text{backingStoreMap currentBS} \cup \{\text{objectID}?}\}\}

\text{backingStoreStacks}' = \text{backingStoreStacks}

\text{rootBS}' = \text{rootBS}

\text{staticClassFieldsID}' = \text{staticClassFieldsID}

\text{staticClassFields}' = \text{staticClassFields}
NewObject \equiv \texttt{var objectID : ObjectID •}
\quad \texttt{newObject?thread?classID\rightarrow}
\quad \texttt{if classID = <classID_1> \rightarrow}
\quad \quad \texttt{AllocateObject(thread, sizeof <classID_1> Obj, objectID); (StructMan <classID_1> ObjInit)}
\quad \quad \quad \texttt{classID = <classID_2> \rightarrow}
\quad \quad \quad \texttt{AllocateObject(thread, sizeof <classID_2> Obj, objectID); (StructMan <classID_2> ObjInit)}
\quad \quad \vdots
\quad \quad \texttt{classID = <classID_n> \rightarrow}
\quad \quad \texttt{AllocateObject(thread, sizeof <classID_n> Obj, objectID); (StructMan <classID_n> ObjInit)}
\quad \texttt{fi; newObjectRet!objectID \rightarrow Skip}

GetObject \equiv
\quad \texttt{getObject?oid : (oid ∈ dom objects) \rightarrow getobjectRet!(objects oid) \rightarrow Skip}
\quad \quad \texttt{if object \notin dom objects \rightarrow Chaos}

PutObject \equiv
\quad \texttt{putObject?oid?struct : (oid ∈ dom objects) \rightarrow objects := objects \oplus \{oid \mapsto struct\}}
\quad \quad \quad \texttt{if object \notin dom objects \rightarrow Chaos}

GetStaticFields \equiv
\quad \texttt{getStaticFields!(staticClassFields ((Initialised ~) staticClassFieldsID)) \rightarrow Skip}

PutStaticFields \equiv
\quad \texttt{putStaticFields!staticFields \rightarrow}
\quad \quad \texttt{staticClassFields := \{(Initialised ~) staticClassFieldsID \mapsto staticFields\}}

GetClassIDOf \equiv \texttt{getClassIDOf?oid!(classIDOf (objects oid)) \rightarrow Skip}

GetCurrentAC \equiv \texttt{getCurrentAC?thread \rightarrow}
\quad \texttt{if thread \in dom backingStoreStacks ∧ backingStoreStacks thread \neq \varnothing \rightarrow}
\quad \quad \texttt{getCurrentACRet!(last (backingStoreStacks thread)) \rightarrow Skip}
\quad \quad \texttt{if thread \notin dom backingStoreStacks ∨ backingStoreStacks thread = \varnothing \rightarrow Chaos fi}

EnterBackingStore \equiv \texttt{var bsid : BackingStoreID •}
\quad \texttt{enterBackingStore?thread?bs → (EnterBS)}

CheckMMReport \equiv \texttt{MMreport?r \rightarrow if r = MMokay \rightarrow Skip ∧ r \neq MMokay \rightarrow Chaos fi}
ExitBackingStore ≜ \text{var} \ thread: \ \text{ThreadID}; \ bsid: \ \text{BackingStoreID}; \ clear: \ \mathbb{B}.
\begin{align*}
\text{exitBackingStore}?t & \rightarrow \ thread := t; \\
(\text{ExitBS}1 \land \text{ExitBS}2); & \\
\text{if} \ clear = \text{True} & \rightarrow \ MMclearBackingStore!bsid \rightarrow \ CheckMMReport \\
\text{fi}; & \text{exitBackingStoreRet!bsid!clear} \rightarrow \text{Skip}
\end{align*}

AddThreadMemory ≜ \text{addThreadMemory}?\text{thread}?\text{bsid} \rightarrow \text{AddThread}

RemoveThreadMemory ≜ \text{removeThreadMemory}?\text{toRemove} \rightarrow \text{RemoveThread}

Loop ≜
(\text{NewObject} \square \text{GetObject} \square \text{PutObject} \square \text{GetStaticFields} \square \text{PutStaticFields} \square \text{GetClassIDOf} \\
\square \text{EnterBackingStore} \square \text{ExitBackingStore} \square \text{AddThreadMemory} \square \text{RemoveThreadMemory} \square \text{GetCurrentAC}); \ \text{Loop}

\begin{itemize}
\item \text{Init}; \ \text{AllocateStaticFields}; \ \text{Loop}
\end{itemize}
\text{end}
Appendix C

Compilation Rules

Rule [Bytecode Expansion]. For a given \( bc \)

\[
HandleInstruction_{bc} \subseteq_A \text{ if } \[ i, pc = i \rightarrow handleAction(bc i) \] fi
\]

where \( handleAction \) is a syntactic function defined by Table 5.1.

Rule [Sequence introduction]. If \( i \neq j \) and

\[
\{frameStack \neq \emptyset\}; A = \{frameStack \neq \emptyset\}; A; \{frameStack \neq \emptyset\}
\]

then,

\[
\begin{align*}
\mu X \bullet & \quad \text{if } frameStack = \emptyset \rightarrow \text{Skip} \\
& \quad \text{if } frameStack \neq \emptyset \rightarrow \\
& \quad \quad \text{if } \dots \\
& \quad \quad \quad \text{if } pc = i \rightarrow A; pc := j \subseteq_A \\
& \quad \quad \quad \text{if } \dots \\
& \quad \quad \quad \text{if } pc = j \rightarrow B \\
& \quad \quad \quad \text{if } \dots \\
& \quad \quad \quad \text{fi; Poll; } X \\
& \quad \text{fi}
\end{align*}
\]

Rule [if conditional introduction]. If \( i \neq j, i \neq k \), and

\[
\begin{align*}
\{frameStack \neq \emptyset\}; A &= \{frameStack \neq \emptyset\}; A; \{frameStack \neq \emptyset\}
\end{align*}
\]
then

\[ \mu X \bullet \]

if frameStack = \emptyset \rightarrow Skip

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = i \rightarrow A; \\
\quad \text{pc} := \text{if } b \text{ then } j \text{ else } k \quad \sqsubseteq_A \\
\quad \cdots \\
\quad \text{pc} = k \rightarrow B; \text{ pc} := j \\
\quad \cdots \\
\quad \text{fi}; \text{ Poll}; \quad X \\
\end{array}
\end{array} \]

fi

\[ \mu X \bullet \]

if frameStack = \emptyset \rightarrow Skip

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = i \rightarrow A; \\
\quad \text{pc} := \text{if } b \text{ then } j \text{ else } k \\
\quad \text{if } b \rightarrow \text{Skip} \\
\quad \text{fi}; \quad \text{pc} := j \\
\quad \cdots \\
\end{array}
\end{array} \]

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = k \rightarrow B; \text{ pc} := j \\
\quad \cdots \\
\quad \text{fi}; \text{ Poll}; \quad X \\
\end{array}
\end{array} \]

\[ \mu X \bullet \]

if frameStack = \emptyset \rightarrow Skip

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = i \rightarrow A; \text{ Poll}; \\
\quad \text{pc} := \text{if } b \text{ then } j \text{ else } k \\
\quad \text{if } b \rightarrow B \\
\quad \text{fi}; \quad \text{pc} := x \\
\quad \cdots \\
\end{array}
\end{array} \]

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = j \rightarrow B; \text{ pc} := x \\
\quad \cdots \\
\quad \text{fi}; \text{ Poll}; \quad X \\
\end{array}
\end{array} \]

\[ \mu X \bullet \]

if frameStack = \emptyset \rightarrow Skip

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = k \rightarrow C; \text{ pc} := x \\
\quad \cdots \\
\quad \text{fi}; \text{ Poll}; \quad X \\
\end{array}
\end{array} \]

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = k \rightarrow C; \text{ pc} := x \\
\quad \cdots \\
\quad \text{fi}; \text{ Poll}; \quad X \\
\end{array}
\end{array} \]

Rule [if-else conditional introduction]. If \( i \neq j, i \neq k, \) and

\( \{ \text{frameStack} \neq \emptyset \}; \ A = \{ \text{frameStack} \neq \emptyset \}; \ A \ ; \ \{ \text{frameStack} \neq \emptyset \} \)

then

\[ \mu X \bullet \]

if frameStack = \emptyset \rightarrow Skip

\[ \begin{array}{l}
\text{if } \cdots \\
\quad \begin{array}{l}
\quad \text{pc} = i \rightarrow A; \\
\quad \text{pc} := \text{if } b \text{ then } j \text{ else } k \quad \sqsubseteq_A \\
\quad \cdots \\
\quad \text{pc} = j \rightarrow B; \text{ pc} := x \\
\quad \cdots \\
\quad \text{fi}; \text{ Poll}; \quad X \\
\end{array}
\end{array} \]

fi

Rule [Conditional introduction]. If \( i \neq j, i \neq k, \) and

\( \{ \text{frameStack} \neq \emptyset \}; \ A = \{ \text{frameStack} \neq \emptyset \}; \ A \ ; \ \{ \text{frameStack} \neq \emptyset \} \)
then

\[
\mu X \bullet \\
\text{if } \text{frameStack} = \emptyset \longrightarrow \text{Skip} \\
\quad \text{if } \text{frameStack} \neq \emptyset \longrightarrow \\
\quad \quad \text{if } \ldots \\
\quad \quad \quad \quad \quad \quad \quad \text{pc} = i \longrightarrow A; \\
\quad \quad \quad \quad \quad \quad \quad \text{pc} := \text{if } b \text{ then } j \text{ else } k \\
\quad \quad \ldots \\
\quad \quad \quad \text{pc} = j \longrightarrow B \\
\quad \quad \ldots \\
\quad \quad \quad \text{pc} = k \longrightarrow C \\
\quad \quad \ldots \\
\quad \quad \text{fi}; \text{Poll} ; \ X \\
\text{fi}
\]

\textbf{Rule [while loop introduction 1].} If } i \neq j,

\{\text{frameStack} \neq \emptyset \}; \ A \\
= \\
\{\text{frameStack} \neq \emptyset \}; \ A; \ \{\text{frameStack} \neq \emptyset \}

then

\[
\mu X \bullet \\
\text{if } \text{frameStack} = \emptyset \longrightarrow \text{Skip} \\
\quad \text{if } \text{frameStack} \neq \emptyset \longrightarrow \\
\quad \quad \text{if } \ldots \\
\quad \quad \quad \quad \quad \quad \quad \text{pc} = i \longrightarrow A; \\
\quad \quad \quad \quad \quad \quad \quad \text{pc} := \text{if } b \text{ then } j \text{ else } k \\
\quad \quad \ldots \\
\quad \quad \quad \text{pc} = j \longrightarrow B \\
\quad \quad \ldots \\
\quad \quad \quad \text{pc} = k \longrightarrow C ; \  pc := i \\
\quad \quad \ldots \\
\quad \quad \text{fi}; \text{Poll} ; \ X \\
\text{fi}
\]

\textbf{Rule [while loop introduction 2].} If } i \neq j,

\{\text{frameStack} \neq \emptyset \}; \ A \\
= \\
\{\text{frameStack} \neq \emptyset \}; \ A; \ \{\text{frameStack} \neq \emptyset \}
then

\[ \mu X \bullet \]

\[ \text{if frameStack} = \emptyset \longrightarrow \text{Skip} \]
\[ \text{if frameStack} \neq \emptyset \longrightarrow \]
  \[
  \text{if } \cdots
  \]
  \[
  \text{pc} = i \longrightarrow A; \quad \text{pc} := \text{if } b \text{ then } j \text{ else } k
  \]
  \[
  \cdots
  \]
  \[
  \text{pc} = j \longrightarrow B; \quad \text{pc} := i
  \]
  \[
  \cdots
  \]
  \[
  \text{pc} = k \longrightarrow C
  \]
  \[
  \text{fi}; \quad \text{Poll}; \quad X
  \]
\[ \text{fi} \]

\[ \mu X \bullet \]

\[ \text{if frameStack} = \emptyset \longrightarrow \text{Skip} \]
\[ \text{if frameStack} \neq \emptyset \longrightarrow \]
  \[
  \text{if } \cdots
  \]
  \[
  \text{pc} = i \longrightarrow \mu Y \bullet A; \quad \text{pc} := \text{if } b \text{ then } j \text{ else } k; \quad \text{Poll};
  \]
  \[
  \text{if } b \longrightarrow B; \quad \text{pc} := i; \quad \text{Poll}; \quad Y
  \]
  \[
  \text{fi}
  \]
  \[
  \cdots
  \]
  \[
  \text{pc} = j \longrightarrow B; \quad \text{pc} := i
  \]
  \[
  \cdots
  \]
  \[
  \text{pc} = k \longrightarrow C
  \]
  \[
  \cdots
  \]
  \[
  \text{fi}; \quad \text{Poll}; \quad X
  \]
\[ \text{fi} \]

**Rule [do-while loop introduction].** If \( i \neq j \),

\[
\{ \text{frameStack} \neq \emptyset \}; \quad A = \{ \text{frameStack} \neq \emptyset \}; \quad A; \quad \{ \text{frameStack} \neq \emptyset \}
\]
then

\[ \mu X \bullet \]

\[ \text{if frameStack} = \emptyset \longrightarrow \text{Skip} \]
\[ \text{if frameStack} \neq \emptyset \longrightarrow \]
  \[
  \text{if } \cdots
  \]
  \[
  \text{pc} = i \longrightarrow A; \quad \text{pc} := \text{if } b \text{ then } i \text{ else } j \quad \subseteq_A
  \]
  \[
  \cdots
  \]
  \[
  \text{pc} = j \longrightarrow B
  \]
  \[
  \cdots
  \]
  \[
  \text{fi}; \quad \text{Poll}; \quad X
  \]
\[ \text{fi} \]

**Rule [Infinite loop introduction].** If

\[
\{ \text{frameStack} \neq \emptyset \}; \quad A = \{ \text{frameStack} \neq \emptyset \}; \quad A; \quad \{ \text{frameStack} \neq \emptyset \}
\]
then

\[ \mu X \bullet \]

\[ \text{if frameStack} = \emptyset \longrightarrow \text{Skip} \]
\[ \text{if frameStack} \neq \emptyset \longrightarrow \]
  \[
  \text{if } \cdots
  \]
  \[
  \text{pc} = i \longrightarrow \mu Y \bullet A; \quad \text{pc} := \text{if } b \text{ then } i \text{ else } j; \quad \text{Poll};
  \]
  \[
  \text{if } b \longrightarrow Y
  \]
  \[
  \text{fi}
  \]
  \[
  \cdots
  \]
  \[
  \text{pc} = j \longrightarrow \text{Skip}
  \]
  \[
  \cdots
  \]
  \[
  \text{fi}; \quad \text{pc} := j
  \]
  \[
  \cdots
  \]
  \[
  \text{fi}; \quad \text{Poll}; \quad X
  \]
\[ \text{fi} \]

\[205\]
Rule [Method call resolution]. If an action $M$ is such that

\[
\{(\text{head frameStack}).\text{storedPC} = i \land \text{frameStack} = fs\}; \ M
= \{(\text{head frameStack}).\text{storedPC} = i \land \text{frameStack} = fs\}; \ M;
\{(pc = i \land \text{frameStack} = \text{tail fs})\}
\]

and $i \neq j$,

\[
\mu X \bullet
\begin{cases}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \rightarrow \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{fi} ; \text{Poll} ; \ X
\end{cases}
\]

\[
\mu X \bullet
\begin{cases}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \rightarrow \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{fi} ; \text{Poll} ; \ X
\end{cases}
\]

Rule [Dynamic method call resolution]. If actions $M_1, \ldots, M_n$ are such that

\[
\{(\text{returnAddress} = i \land \text{frameStack} = fs); \ M_k
= \{(\text{returnAddress} = i \land \text{frameStack} = fs); \ M_k; \ \{pc = i \land \text{frameStack} = \text{tail fs}\}
\]

for $k \in \{1, \ldots, n\}$ and $i \neq j$,

\[
\mu X \bullet
\begin{cases}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \rightarrow \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{fi} ; \text{Poll} ; \ X
\end{cases}
\]

\[
\mu X \bullet
\begin{cases}
\text{if } \text{frameStack} = \emptyset \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset \rightarrow \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{if } \ldots \\
\text{fi} ; \text{Poll} ; \ X
\end{cases}
\]

Rule [Main Action Refinement]. If $entry(c_i, m_i) = j_i$ for $i \in \{1, \ldots, n\}$ and

\[
\{(\# \text{frameStack} = 1); \ M_i
= \{(\# \text{frameStack} = 1); \ M_i; \ \{\text{frameStack} = \emptyset\}
\]

206
\{ \# \text{frameStack} = 1 \\
\land \text{pc} = \text{entry}(\text{cid}, \text{mid}) \};

\mu X \bullet
\begin{align*}
\text{if } \text{frameStack} = \emptyset & \rightarrow \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset & \\
\text{if } \text{pc} = j_1 & \rightarrow M_1 \\
& \cdots \\
\text{if } \text{pc} = j_n & \rightarrow M_n \\
\text{fi; Poll; } X
\end{align*}

\begin{align*}
\{ \# \text{frameStack} = 1 \}; \\
\text{if}(\text{cid}, \text{mid}) = (c_1, m_1) & \rightarrow M_1 \\
\cdots \\
\text{if}(\text{cid}, \text{mid}) = (c_n, m_n) & \rightarrow M_n \\
\text{fi}
\end{align*}
Appendix D

Z/Eves Theorems and Proofs

D.1 Additional Toolkit Lemmas

section additional_lemmas parents standard_toolkit

D.1.1 Sets

D.1.1.1 Empty Set

theorem grule nullComprehension [X]
\( \forall P : X \cdot \{ x : \emptyset [X] \cdot P \} = \{ \} \)

D.1.1.2 Union

theorem disabled rule cupElementLeft [X]
\( \forall x : X ; S : P X \mid x \in S \cdot \{ x \} \cup S = S \)

theorem disabled rule cupElementRight [X]
\( \forall x : X ; S : P X \mid x \in S \cdot S \cup \{ x \} = S \)

D.1.1.3 Set Difference

theorem disabled rule diffDisjoint [X]
\( \forall A, B : P X \cdot A \cap B = \{ \} \Leftrightarrow A \setminus B = A \)

theorem rule cupDiff [X]
\( \forall A, B : P X \cdot A \cup (B \setminus A) = A \cup B \)

theorem rule capDiff [X]
\( \forall A, B : P X \cdot A \cap (B \setminus A) = \{ \} \)
theorem capDiffDisjoint [X]
∀ A, B, C : P X • A ∩ B = {} ⇒ A ∩ (B \ C) = {}

theorem splitDisjoint [X]
∀ A, B, C : P X • A = B ∪ C ∧ B ∩ C = {} ⇒ A \ B = C ∧ A \ C = B

D.1.2 Relations

D.1.2.1 Composition

theorem grule compIsRel [X, Y, Z]
∀ R : X ↔ Y; Q : Y ↔ Z • R ⊤ Q ∈ P(X × Z)

theorem rule compCup [X, Y, Z]
∀ R : X ↔ Y; S, T : Y ↔ Z • R ⊤ (S ∪ T) = (R ⊤ S) ∪ (R ⊤ T)

theorem rule cupComp [X, Y, Z]
∀ R, S : X ↔ Y; T : Y ↔ Z • (R ∪ S) ⊤ T = (R ⊤ T) ∪ (S ⊤ T)

D.1.2.2 Domain and Range Anti-restriction

theorem grule ndresIsRel [X, Y]
∀ R : X ↔ Y; S : P Y • R ⊤ S ∈ P(X × Y)

theorem grule nrresIsRel [X, Y]
∀ R : X ↔ Y; S : P Y • R ⊤ S ∈ P(X × Y)

theorem rule rresSelf [X]
∀ R : X ↔ X • R ⊢ X = R

theorem disabled rule moveNrres [X]
∀ Q, R : X ↔ X; S : P X • (Q ⊤ R) ⊢ S = Q ⊤ (R ⊢ S)

theorem disabled rule moveNrresBackwards [X]
∀ Q, R : X ↔ X; S : P X • Q ⊤ (R ⊢ S) = (Q ⊤ R) ⊢ S

theorem disabled rule flipNrres [X]
∀ Q, R : X ↔ X; S : P X • (Q ⊢ S) ⊤ R = (Q ⊤ S) ⊣ R
D.1.2.3 Relational Image

**Theorem** grule imageType \([X, Y]\)
\[
\forall R : X \leftrightarrow Y; ~ S : \mathbb{P} X \bullet R \upharpoonright \{S\} \in \mathbb{P} Y
\]

**Theorem** grule nrresCompImageIsRel \([X, Y]\)
\[
\forall R : X \leftrightarrow Y; ~ Q : Y \leftrightarrow Y; ~ S : \mathbb{P} X \bullet R \upharpoonright ((R \upharpoonright Q) \upharpoonright \{S\}) \in \mathbb{P}(X \times Y)
\]

**Theorem** disabled rule inImageUnit \([X]\)
\[
\forall R : X \leftrightarrow X; ~ x : x \in R \upharpoonright \{x\} \iff (x, y) \in R
\]

**Theorem** disabled rule inImageUnitBackwards \([X]\)
\[
\forall R : X \leftrightarrow X; ~ x : x \in R \iff y \in R \upharpoonright \{x\}\]

**Theorem** disabled rule ranIsBigCupImage \([X, Y]\)
\[
\forall R : X \leftrightarrow Y \bullet \text{ran } R = \bigcup \{x : X \bullet R \upharpoonright \{x\}\}
\]

D.1.2.4 Transitive Closure

**Theorem** grule plusType \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R^+ \in X \leftrightarrow X
\]

**Theorem** starContainsPlus \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R^+ \subseteq R^*
\]

**Theorem** plusTransRight \([X]\)
\[
\forall R : X \leftrightarrow X; ~ y : X \bullet (x, y) \in R^+ \land (y, z) \in R \Rightarrow (x, z) \in R^+
\]

**Theorem** starTransRight \([X]\)
\[
\forall R : X \leftrightarrow X; ~ y : X \bullet (x, y) \in R^* \land (y, z) \in R \Rightarrow (x, z) \in R^*
\]

**Theorem** disabled rule plusStarComp \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R^+ \upharpoonright R^* \upharpoonright = R^+
\]

**Theorem** disabled rule plusStarCompBackwards \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R^+ \upharpoonright R^* \upharpoonright = R^+
\]

**Theorem** disabled rule starPlusComp \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R^* \upharpoonright R^+ = R^+
\]
\textbf{theorem} \ relIntoPlus \ [X] \\
\forall \ R : X \leftrightarrow X; \ y : X \Rightarrow (x, y) \in R \land (y, z) \in R^+ \Rightarrow (x, z) \in R^+

\textbf{theorem} \ relCompPlusInPlus \ [X] \\
\forall \ R : X \leftrightarrow X \Rightarrow R \subseteq R^+

\textbf{theorem} \ selfInSelfCompStar \ [X] \\
\forall \ R : X \leftrightarrow X \Rightarrow R \subseteq R^*_R

\textbf{theorem} \ selfInStarCompSelf \ [X] \\
\forall \ R : X \leftrightarrow X \Rightarrow R \subseteq R^*_R

\textbf{D.1.2.5} \ Functions

\textbf{theorem} \ grule \ applyType \ [X, Y] \\
\forall \ f : X \rightarrow Y \Rightarrow \forall \ x : \text{dom} \ f \Rightarrow f x \in Y

\textbf{D.1.3} \ Numbers and Finiteness

\textbf{D.1.3.1} \ Natural Numbers

\textbf{theorem} \ frule \ natCases \\
\forall \ x : \mathbb{N} \Rightarrow x = 0 \lor x \geq 1

\textbf{theorem} \ rule \ natSumInNat \\
\forall \ x, y : \mathbb{N} \Rightarrow x + y \in \mathbb{N}

\textbf{D.1.3.2} \ Relational Iteration and Further Transitive Closure Theorems

\textbf{theorem} \ iterIsRelForallNat1Set \ [X] \\
\forall \ R : X \leftrightarrow X \Rightarrow \mathbb{N}_1 \subseteq \{ n : \mathbb{N}_1 | \text{iter} \ n \ R \in \mathcal{P}(X \times X) \}

\textbf{theorem} \ grule \ iterMaxType \ [X] \\
\forall \ R : X \leftrightarrow X; \ n : \mathbb{Z} \Rightarrow \text{iter} \ n \ R \in \mathcal{P}(X \times X)

\textbf{theorem} \ grule \ iterIsRel \ [X] \\
\forall \ R : X \leftrightarrow X; \ n : \mathbb{Z} \Rightarrow \text{iter} \ n \ R \in X \leftrightarrow X

\textbf{theorem} \ grule \ bigcupIter1IsPowerCross \ [X] \\
\forall \ R : X \leftrightarrow X \Rightarrow \bigcup \{ n : \mathbb{N}_1 \Rightarrow \text{iter} \ n \ R \} \in \mathcal{P}(X \times X)
\textbf{theorem} grule bigcupIterIsPowerCross \([X]\)
\[
\forall R : X \leftrightarrow X \bullet \bigcup \{n : \mathbb{N} \bullet \text{iter } n \ R\} \in \mathcal{P}(X \times X)
\]

\textbf{theorem} selfInBigcupIter \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R \subseteq \bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R\}
\]

\textbf{theorem} bigcupIterCompBigcupIterInBigcupIter \([X]\)
\[
\forall R : X \leftrightarrow X \bullet (\bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R\}) \subseteq (\bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R\})
\]

\textbf{theorem} disabled rule plusIsBigcupIter \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R + = \bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R\}
\]

\textbf{theorem} disabled rule starIsBigcupIter \([X]\)
\[
\forall R : X \leftrightarrow X \bullet R * = \bigcup \{n : \mathbb{N} \bullet \text{iter } n \ R\}
\]

\textbf{theorem} disabled rule iterUnfoldStart \([X]\)
\[
\forall R : X \leftrightarrow X ; n : \mathbb{N} \bullet \text{iter } (n + 1) \ R = R \uparrow (\text{iter } n \ R)
\]

\textbf{theorem} plusIterAbsorb \([X]\)
\[
\forall R : X \leftrightarrow X ; n : \mathbb{N}_1 \bullet (R ^+) \uparrow (\text{iter } n \ R) \subseteq R ^+
\]

\textbf{theorem} inclusion1CasesSet \([X]\)
\[
\forall R : X \leftrightarrow X ; S : \mathcal{P} X \bullet \mathbb{N}_1 \subseteq \{n : \mathbb{N}_1 \mid \text{iter } n \ R \triangleright R + \{S\} \subseteq \text{iter } n \ (R \triangleright R + \{S\})\}
\]

\textbf{theorem} disabled rule nrresBigcupIterDistrib \([X]\)
\[
\forall R : X \leftrightarrow X ; S : \mathcal{P} X \bullet \bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R\} \triangleright S = \bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R \triangleright S\}
\]

\textbf{theorem} disabled rule nrresBigcupIterDistribBackwards \([X]\)
\[
\forall R : X \leftrightarrow X ; S : \mathcal{P} X \bullet \bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R \triangleright S\} = \bigcup \{n : \mathbb{N}_1 \bullet \text{iter } n \ R \triangleright S\}
\]

\textbf{theorem} inclusion1Cases \([X]\)
\[
\forall R : X \leftrightarrow X ; S : \mathcal{P} X ; n : \mathbb{N}_1 \bullet \text{iter } n \ R \triangleright R + \{S\} \subseteq \text{iter } n \ (R \triangleright R + \{S\})
\]

\textbf{theorem} inclusion1 \([X]\)
\[
\forall R : X \leftrightarrow X ; S : \mathcal{P} X \bullet R ^+ \triangleright R ^+ \{S\} \subseteq (R \triangleright R ^+ \{S\}) ^+
\]
\textbf{D.1.3.3} Ranges

\textbf{Theorem} rule \texttt{diffRangeStart}
\[ \forall a, b, c : \mathbb{Z} \mid c \leq b \land a \leq c \bullet (a \ldots b) \setminus (a \ldots c) = c + 1 \ldots b \]

\textbf{D.1.3.4} Finiteness

\textbf{Theorem} \texttt{grule diffFinite [X]}
\[ \forall A, B : \mathbb{F} X \bullet A \setminus B \in \mathbb{F} X \]

\textbf{Theorem} disabled rule \texttt{diffCupCupCupDiff [X]}
\[ \forall A, B, C, D : \mathbb{F} X \mid C \in \mathbb{P} A \land D \in \mathbb{P} (B \cup C) \bullet ((A \setminus C) \cup D) \cup ((B \cup C) \setminus D) = A \cup B \]

\textbf{Theorem} disabled rule \texttt{finiteDef [X]}
\[ S \in \mathbb{F} X \Leftrightarrow S \in \mathbb{P} X \land \exists n : \mathbb{N} \bullet \exists f : 1 \ldots n \rightarrow S \bullet \text{ran} f = S \]

\textbf{Theorem} \texttt{finiteRelPlusImageIsFinite [X]}
\[ \forall R : X \leftrightarrow X ; S : \mathbb{P} X \bullet \text{ran} R \in \mathbb{F} X \Rightarrow R + \{S\} \in \mathbb{F} X \]

\textbf{Theorem} \texttt{finiteSubsetIsFinite [X]}
\[ \forall A : \mathbb{F} X \bullet \forall B : \mathbb{P} A \bullet B \in \mathbb{F} X \]

\textbf{D.1.3.5} Cardinality

\textbf{Theorem} disabled rule \texttt{cupCard [X]}
\[ \forall S, T : \mathbb{F} X \bullet \#(S \cup T) = \# S + \# T - \#(S \cap T) \]
\textbf{theorem} disabled rule \text{cardCupDisjoint} [X]
\[\forall A, B : \mathbb{F} X \mid A \cap B = \{\} \implies \#(A \cup B) = \# A + \# B\]

\textbf{theorem} disabled rule \text{plusCardDisjoint} [X]
\[\forall A, B : \mathbb{F} X \mid A \cap B = \{\} \implies \# A + \# B = \#(A \cup B)\]

\textbf{D.1.3.6} Finite Function Spaces

\textbf{theorem} grule \text{ndresInFfun} [X, Y]
\[\forall S : \mathbb{P} X ; f : X \Rightarrow Y \implies S \triangleleft f \in X \Rightarrow Y\]

\textbf{theorem} grule \text{oplusInFfun} [X, Y]
\[\forall f, g : X \Rightarrow Y \implies f \oplus g \in X \Rightarrow Y\]

\textbf{theorem} grule \text{singletonInFfun} [X, Y]
\[\forall x : X ; y : Y \implies \{\{x \mapsto y\}\} \in X \Rightarrow Y\]

\textbf{D.2} Proofs of Additional Toolkit Lemmas

\textbf{D.2.1} Sets

\section{additional_lemmas_sets_proofs parents additional_lemmas}

\textbf{D.2.1.1} Empty Set

\textbf{proof} [nullComprehension]
prove by reduce;
apply extensionality;
prove;

\textbf{D.2.1.2} Union

\textbf{proof} [cupElementLeft]
use \text{cupSubsetLeft}[X][S := \{x\}, T := S];
prove;

\textbf{proof} [cupElementRight]
use \text{cupSubsetLeft}[X][S := \{x\}, T := S];
prove;
D.2.1.3 Set Difference

**proof**[diffDisjoint]
apply extensionality to predicate \((\_ \cap \_ \_)[X] (A, B) = \{\_\};
prove by reduce;
cases;
rewrite;
apply extensionality to predicate \((\_ \_ \_ \_)[X] (A, B) = A;
rewrite;
instantiate \_ \_ \_ \_ == \_;
prove;
next;
apply extensionality to predicate \((\_ \_ \_ \_)[X] (A, B) = A;
prenex;
prove;
instantiate \_ \_ \_ \_ == \_;
prove;
next;

**proof**[cupDiff]
apply extensionality to predicate \((\_ \_ \_ \_)[X] (A, \_ \_ \_ \_ [X] (B, A)) = (\_ \_ \_ \_ \_)[X] (A, B);
prove;

**proof**[capDiff]
apply extensionality;
prove;

**proof**[capDiffDisjoint]
apply extensionality to predicate \((\_ \_ \_ \_ \_ \_ \_)[X] (A, \_ \_ \_ \_ \_ \_ \_ [X] (B, C)) = \{\_\};
prove;
apply extensionality to predicate \((\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_)[X] (A, B) = \{\_\};
prove;
instantiate \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ == \_;
rewrite;

**proof**[splitDisjoint]
equality substitute;
apply distributeDiffOverCupLeft;
rewrite;
apply diffSuperset to expression \((\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ [X] (B, B));
rewrite;
apply diffSuperset to expression \((\_ \_ \_ \_ \_ \_ \_ \_ \_ \_ \_ [X] (C, C));
rewrite;
use diffDisjoint[X][A := B, B := C];
prove;
use diffDisjoint[X][A := C, B := B];
prove;

D.2.2 Relations

section additional_lemmas_relations_proofs parents additional_lemmas

D.2.2.1 Composition

**proof**[compIsRel]
prove;
proof[compCup]
apply extensionality;
prove;
apply compDef;
prove;
instantiate y___4 == y___0;
prove;
instantiate y___3 == z;
prove;
split z___l ∈ X ∧
            y___0 ∈ Y ∧
            z___0 ∈ Z ∧
            y___1 = (x___1, z___0) ∧
            (x___1, y___0) ∈ R ∧
            (y___0, z___0) ∈ S;
cases;
instantiate y == y___0;
prove;
next;
instantiate y == y___2;
prove;
next;

proof[compCup]
apply extensionality;
prove;
apply compDef;
prove;
instantiate y___4 == y___0;
prove;
instantiate y___3 == y;
prove;
split z___l ∈ X ∧
            y___0 ∈ Y ∧
            z___0 ∈ Z ∧
            y___1 = (x___1, z___0) ∧
            (x___1, y___0) ∈ R ∧
            (y___0, z___0) ∈ T;
cases;
instantiate y == y___0;
prove;
next;
instantiate y == y___2;
prove;
next;

D.2.2.2 Domain and Range Anti-restriction

proof[ndresIsRel]
prove;

proof[nrresIsRel]
prove;
proof[rresSelf]
  apply rresDef;
  prove;
  apply extensionality;
  prove by reduce;
  invoke (_ \leftrightarrow _);
  apply inPower;
  instantiate e == y;
  prove;

proof[moveNrres]
  use relationExtensionality[X, X][
  Q := (_ \rightarrow \leftarrow)_X \cdot [X, X] \cdot [X, X] \cdot (Q, R), S),
  R := (_ \rightarrow \leftarrow)_X \cdot [X, X] \cdot [X, X] \cdot (Q, (_ \rightarrow \leftarrow)_X \cdot [X, X] \cdot (R, S))];
  prove;

proof[moveNrresBackwards]
  apply moveNrres;
  prove;

proof[flipNrrres]
  apply extensionality;
  prove by reduce;
  apply compDef;
  apply nrresDef;
  apply ndresDef;
  prove;
  invoke (_ \leftrightarrow _);
  prove;
  cases;
  instantiate y.2 == y.0;
  prove;
  next;
  instantiate y.3 == y.1;
  prove;
  next;

D.2.2.3 Relational Image

proof[imageType]
  prove;

proof[nrresCompImageIsRel]
  prove;

proof[inImageUnit]
  apply inImage;
  prove;

proof[inImageUnitBackwards]
  apply inImageUnit;
  prove;
proof[ranIsBigcupImage]
  apply extensionality;
  prove;
  apply inRan;
  prove;
  instantiate B == \( (\_ \_ \_ \{ \_ \} [X, Y] (R, \{z_0\})) \);
  prove;
  instantiate z_1 == z;
  prove;
  apply inImage;
  prove;

D.2.2.4 Transitive Closure

proof[plusType]
  prove;

proof[starContainsPlus]
  apply starDef2;
  prove;

proof[plusTransRight]
  prove by reduce;
  use plusContainsSelf[X];
  prove;
  apply inPower;
  instantiate e == \((y, z)\);
  rewrite;
  use plusIsTransitive[X];
  prove;
  apply inPower;
  instantiate e_0 == \((x, z)\);
  apply pairInComp;
  prove;
  instantiate y_0 == y;
  prove;

proof[starTransRight]
  split x = y;
  cases;
  use starContainsSelf[X];
  prove;
  next;
  use plusTransRight[X];
  apply starDef2;
  prove;
  next;

proof[plusStarComp]
  apply starDef2;
  simplify;
  apply compCup;
  simplify;
  apply idInRel;
  simplify;
  apply inPowerSelf;
  prove;
  apply cupSubsetRight;
  prove;
  use plusIsTransitive[X];
  prove;

proof[plusStarCompBackwards]
  apply plusStarComp;
  prove;
proof[starPlusComp]
apply starDef2;
simplify;
apply cupComp;
simplify;
apply idInRel;
simplify;
apply inPowerSelf;
prove;
apply cupSubsetRight;
prove;
use plusIsTransitive[X];
prove;

proof[relIntoPlus]
use plusContainsSelf[X];
use plusIsTransitive[X];
prove;
apply inPower;
instantiate e_0 == (x, y);
prove;
instantiate e == (x, z);
prove;
instantiate y_0 == y;
prove;

proof[relCompPlusInPlus]
prove;
apply compDef;
apply inPower;
prove;
use relIntoPlus[X];
prove;

proof[selfInSelfCompStar]
prove;
use compMonotone[X, X, X]|Q := R, Q' := R, R' := id X, R := (\ast) [X] R];
prove;
apply inPower to predicate id X \in \mathcal{P} (\ast) [X] R;
prove;
apply starDef2;
prove;

proof[selfInStarCompSelf]
prove;
use compMonotone[X, X, X]|R := R, R' := R, Q' := id X, Q := (\ast) [X] R];
prove;
apply inPower to predicate id X \in \mathcal{P} (\ast) [X] R;
prove;
apply starDef2;
prove;
apply dresElimination to expression (\ast \triangleleft \ast) [X] X (X, R);
prove;

D.2.2.5 Functions

proof[applyType]
prove;

D.2.3 Numbers and Finiteness
D.2.3.1 Natural Numbers

\[\text{proof[natCases]}\]
prove by reduce;

\[\text{proof[natSumInNat]}\]
prove;

D.2.3.2 Relational Iteration and Further Transitive Closure Theorems

\[\text{proof[iterIsRelForallNat1Set]}\]
apply nat1Induction;
use iterPositive[X][R \_ 1 := R, n \_ 0 := x];
prove;
split iter [X] (1 + x) R = (_ \uplus \_)[X, X, X] (R, iter [X] x R);
prove;

\[\text{proof[iterMaxType]}\]
split n ≥ 1;
cases;
use iterIsRelForallNat1Set[X];
prove;
apply inPower;
 instantiate e == n;
apply inNat1;
prove;
 instantiate e \_ 0 == n;
apply inNat1;
next;
split n = 0;
cases;
next;
prove;
split n < 0;
cases;
prove;
use iterNegative[X][R \_ 0 := R];
prove;
use iterIsRelForallNat1Set[X][R := (_\sim)[X, X] R];
prove;
apply inPower to predicate N 1 ∈ P \{n \_ 0 : N 1 \mid iter [X] n \_ 0 ((_\sim)[X, X] R) ∈ P (X × X)\};
instantiate e == (−1 \ast n);
apply inNat1;
prove;
instantiate e == (−1 \ast n);
apply inNat1;
next;
prove by reduce;
next;

\[\text{proof[iterIsRel]}\]
prove;

\[\text{proof[bigcupIter1IsPowerCross]}\]
invoke (_\leftrightarrow _\);\[\text{apply bigcupInPower};\]
prove;
apply inPower to predicate \{n : N 1 \mid iter [X] n R \}\ ∈ P P (X × X);
prove;
use iterIsRel[X];
prove;
\textbf{proof}[\texttt{bigcupIterIsPowerCross}]
\begin{itemize}
\item invoke \((\_ \leftrightarrow \_);\)
\item apply \texttt{bigcupInPower};
\item apply \texttt{inPower} to predicate \(\{ n : \mathbb{N} \bullet \text{iter} [X] n R \} \in \mathcal{P} (X \times X);\)
\item use \texttt{iterIsRel}[X];
\item prove;
\end{itemize}

\textbf{proof}[\texttt{selfInBigcupIter}]
\begin{itemize}
\item prove by reduce;
\item apply \texttt{inPower};
\item apply \texttt{inBigcup};
\item prove;
\item cases;
\item instantiate \(B \equiv \text{iter}[X] 1 R;\)
\item prove;
\item instantiate \(n \equiv 1;\)
\item prove;
\item next;
\item apply \texttt{inPower};
\item prove;
\item next;
\end{itemize}

\textbf{proof}[\texttt{bigcupIterCompBigcupIterInBigcupIter}]
\begin{itemize}
\item prove;
\item apply \texttt{inPower};
\item prove;
\item apply \texttt{compDef};
\item prove;
\item apply \texttt{inBigcup};
\item prove;
\item split \(\{ n_1 : \mathbb{N} \bullet \text{iter} [X] n_1 R \} \in \mathcal{P} (X \times X);\)
\item cases;
\item instantiate \(B_{\_1} \equiv \text{iter}[X] (n + n_0) R;\)
\item prove;
\item instantiate \(n_{\_1} \equiv n + n_0;\)
\item prove;
\item use \texttt{composePositiveIterates}[X][Y := X, k := n_0];
\item prove;
\item apply \texttt{extensionality} to predicate \(\text{iter} [X] (n + n_0) R = (\_ \leftrightarrow \_) [X, X, X] (\text{iter} [X] n R, \text{iter} [X] n_0 R);\)
\item apply \texttt{compDef} to expression \((\_ \leftrightarrow \_) [X, X, X] (\text{iter} [X] n R, \text{iter} [X] n_0 R);\)
\item prove;
\item split \text{iter} \[X] n_0 R \in X \leftrightarrow X;
\item cases;
\item prove;
\item instantiate \(y_{\_1} \equiv (x, z);\)
\item prove;
\item instantiate \(y_{\_2} \equiv y;\)
\item prove;
\item next;
\item use \texttt{iterIsRel}[X][n := n_0];
\item prove;
\item next;
\item apply \texttt{inPower};
\item prove;
\end{itemize}
\textbf{proof}[\textit{plusIsBigcupIter}]

apply extensionality;
prove;
cases;
apply plusDef;
prove;
apply inBigcap;
prove;
split \(z \in X \times X\);
cases;
next;
prove by reduce;
next;
prove;
\begin{align*}
\text{instantiate } B &= \bigcup_{X \times X} \{ n : N | \bullet \text{iter}[X] n R \}; \\
\text{use selfInBigcupIter}[X]; \\
\text{use bigcupIterCompBigcupIterInBigcupIter}[X]; \\
\text{prove}; \\
\text{next};
\end{align*}
apply inBigcup;
prove;
\begin{align*}
split y &\in X \times X \land \{ n_0 : N | \bullet \text{iter}[X] n_0 R \} \in \mathcal{P} \mathcal{P} (X \times X); \\
cases;
\end{align*}
prove;
\begin{align*}
\text{use iterInPlus}[X]; \\
\text{prove}; \\
\text{apply inPower}; \\
\text{instatiate } e_0 &= y; \\
\text{prove}; \\
\text{next};
\end{align*}
prove by reduce;
apply inPower;
prove;
next;

\hfill \blacksquare
proof[starIsBigcupIter]
use starDef[2][X];
use plusIsBigcupIter[X];
prove;
apply extensionality;
prove;
cases;
split x ∈ id X;
cases;
apply inBigcup to predicate x ∈ \bigcup [X \times X] \{ n_3 : \mathbb{N} \bullet \text{iter} [X] n_3 \} R ;
prove;
cases;
 instantiate B == id X;
prove;
instantiate n == 0;
prove;
next;
apply inPower to predicate \{ n_0 : \mathbb{N} \bullet \text{iter} [X] n_0 \} R \in \mathbb{P} (X \times X);
prove;
next;
apply inBigcup to predicate x ∈ \bigcup [X \times X] \{ n_1 : \mathbb{N} \bullet \text{iter} [X] n R \} ;
apply inBigcup to predicate x ∈ \bigcup [X \times X] \{ n_6 : \mathbb{N} \bullet \text{iter} [X] n_6 \} R ;
prove;
split z ∈ X \times X \land \{ n_4 : \mathbb{N} \bullet \text{iter} [X] n_4 \} R \in \mathbb{P} (X \times X);
cases;
 split \{ n_5 : \mathbb{N} \bullet \text{iter} [X] n_5 \} R \in \mathbb{P} (X \times X);
cases;
instantiate B_2 == B;
prove;
instantiate n_0 == n;
prove;
next;
apply inPower to predicate \{ n_0 : \mathbb{N} \bullet \text{iter} [X] n_0 \} R \in \mathbb{P} (X \times X);
prove;
next;
apply inBigcup to predicate y ∈ \bigcup [X \times X] \{ n : \mathbb{N} \bullet \text{iter} [X] n R \} ;
split y ∈ X \times X \land \{ n : \mathbb{N} \bullet \text{iter} [X] n R \} \in \mathbb{P} (X \times X);
cases;
apply inBigcup to predicate y ∈ \bigcup [X \times X] \{ n_5 : \mathbb{N} \bullet \text{iter} [X] n_5 \} R ;
prove;
split \{ n_5 : \mathbb{N} \bullet \text{iter} [X] n_5 \} R \in \mathbb{P} (X \times X);
cases;
instantiate B_2 == B;
prove;
instantiate n_0 == n;
prove;
split n = 0;
cases;
prowe;
next;
prowe;
apply inPower to predicate \{ n_0 : \mathbb{N} \bullet \text{iter} [X] n_0 \} R \in \mathbb{P} (X \times X);
prowe;
use iterIsRel[X][n := n_0];
prowe;
prowe;
apply inPower to predicate \{ n_4 : \mathbb{N} \bullet \text{iter} [X] n_4 \} R \in \mathbb{P} (X \times X);
prowe;
proof[iterUnfoldStart]
prove;
apply extensionality;
prove;
use composePositiveIterates[\{X\}[n := 1, k := n, Y := X]];
prove;

proof[plusIterAbsorb]
use compMonotone[\{X, X, X\}]
\[ Q := \text{iter}[X] R, Q' := \text{iter}[X] R; \]
\[ R := \text{iter}[X] R, R' := \text{iter}[X] n R; \]
use iterInPlus[\{X\}];
use plusIsTransitive[\{X\}];
prove;

proof[inclusion1CasesSet]
apply nat1Induction;
prove;
use iterIsRel[\{X\}[n := 1 + x, R := \{\text{iter} \} X X \{X, X, X\} R S]];
prove;
apply inPower to predicate
\[ \text{iter} [X] (1 + x) \{\text{iter} \} X X \{X, X, X\} R S; \]
use iterUnfoldStart[\{X\}[n := x]];
\textbf{proof} \text{[nresBigcupIterDistrib]}

apply extensionality;

prove;
cases;
split \(z \in X \times X\);
cases;
apply \text{inBigcup};
prove;
cases;
prove;
split \(\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R\} \in P \ P (X \times X)\);
cases;
prove;
instantiate \(B \_0 == \_[] \_[] [X, X] (\text{iter} [X] n R, S)\);
prove;
instantiate \(n \_0 == n\);
prove;
next;
prove;
apply \text{inPower};
prove;
next;
prove;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
prove;
next;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
instantiate \(B \_0 == \text{iter} [X] n R\);
instantiate \(n \_0 == n\);
prove;
next;
prove;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
instantiate \(B \_0 == \_[] \_[] [X, X] (\text{iter} [X] n \_1 R, S)\);
instantiate \(n \_0 == n\);
prove;
next;
prove;
apply \text{inBigcup}\text{[nresBigcupIterDistrib]};

\[\text{split} \ y \in X \times X \wedge \{n \_0: N \_1 \bullet (\_[] \_[] [X, X] (\text{iter} [X] n \_0 R, S)\} \in P \ P (X \times X)\];
cases;
prove;
next;
prove;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
apply \text{inBigcup};

\[\text{split} \ x \in X \times X \wedge \{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
cases;
prove;
next;
prove;
apply \text{inPower} to predicate
\[\{n \_2: N \_1 \bullet (x, x) \in \text{iter} [X] n \_2 R, S\} \in P \ P (X \times X)\];
prove;
next;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
apply \text{inPower} to predicate
\[\{n \_1: N \_1 \bullet (x, x) \in \text{iter} [X] n \_1 R, S\} \in P \ P (X \times X)\];
prove;
next;
proof[arresBigcupIterDistribBackwards]
apply arresBigcupIterDistrib;
prove;

proof[inclusion1Cases]
use inclusion1CasesSet[X];
prove;
cases;
apply inPower;
prove;
 instantiate e⁻¹ == n;
split n ∈ N;
prove;
next;
apply inPower to predicate
iter [X] n (\_ braid \_ [X, X] (R, (\_ \# \_)) [X, X] ((\_ +) [X] R, S)) ∈ P (X × X);
prove;
prove;
next;
■
proof[binclusion1]
prove;
apply inPower to predicate
\((\Delta \triangleright \omega) [X, X] (\Delta^T) [X, R, (\omega \uplus \omega) [X, X] ((\omega^+ \triangleright) [X, X] R))\)
\(\in \mathbb{P} (\Delta^T) [X, X] (\Delta \triangleright \omega) [X, X] ((\omega^+ \triangleright) [X, X] R)\));
apply plusIsBigcupIter;
with disabled (inNRes) prove;
apply nresBigcupIterDistrib;
prove;
apply inBigcup;
cases;
split \(e \in X \times X \wedge \)
\(\{n \in \mathbb{N}, (\omega \uplus \omega) [X, X] (\omega \uplus \omega) [X, X] \} \in \mathbb{P} (X \times X)\);
simplify;
prenex;
rewite;
prenex;
instantate \(B = 0\) \(\Rightarrow \)
\(\omega^T [X] (\omega \uplus \omega) [X, X] \wedge (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
simplify;
rewite;
use inclusion1Cases[X];
rearrange;
rewite;
apply inPower;
instantiate \(e = \omega^T\) \(\Rightarrow \)
equality substitute;
apply plusIsBigcupIter;
prove;
next;
prove;
apply inPower to predicate
\(\{n \in \mathbb{N}, \omega \uplus \omega [X, X] (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
prove;
next;
split \(e \in X \times X \wedge \)
\(\{n \in \mathbb{N}, \omega \uplus \omega [X, X] (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
simplify;
rewite;
apply inPower to predicate
\(\{n \in \mathbb{N}, \omega \uplus \omega [X, X] (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
prove;
apply iterIsRel[X] \(\{n \in \mathbb{N}, e \uplus e [X, X] (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
use iterIsRel[X] \(\{n \in \mathbb{N}, e \uplus e [X, X] (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
prove;
next;
prove;
apply inPower to predicate
\(\{n \in \mathbb{N}, \omega \uplus \omega [X, X] (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
prove;
use iterIsRel[X] \(\{n \in \mathbb{N}, \omega \uplus \omega [X, X] (\omega \uplus \omega) [X, X] \}
\(\in \mathbb{P} (X \times X)\);
apply nresBigcupIterDistribBackwards;
prove;
next;

proof[iterUnfoldEnd]
use composePositiveIterates[X][Y := X, n := n - 1, k := 1];
prove;

227
proof[plusUnfoldEnd]
use plusIsBigcupIter[X];
use starIsBigcupIter[X];
prove;
apply extensionality;
prove;
cases;
prove;
apply compDef;
prove;
apply inBigcup;
prove;
split \( x \in X \times X \land \{ n : N \bullet \iter[X] n \in R \} \in \mathcal{P}(X \times X) \);
cases;
prove;
assert \( \{ n_{\_1} : N \bullet \iter[X] n_{\_1} \in R \} \in \mathcal{P}(X \times X) \);
cases;
prove;
instatiate \( B_{\_2} \equiv \iter[X](n-1) \);
prove;
instatiate \( n_{\_0} \equiv n-1 \);
prove;
apply iterUnfoldEnd to expression \( \iter[X] n \in R \);
prove;
apply compDef;
prove;
use \( \iterIsRel[X][n \equiv n-1] \);
prove;
assert \( \{ n : N \bullet \iter[X] n \in R \} \in \mathcal{P}(X \times X) \);
cases;
prove;
next;
prove;
apply inPower;
prove;
apply inPower;
prove;
use \( \iterIsRel[X][n \equiv n_0] \);
prove;
invoke \( \_ \leftrightarrow \_ \);
apply inPower to predicate \( \iter[X] n_0 \in R \in \mathcal{P}(X \times X) \);
assert \( \epsilon_{\_3} \equiv \epsilon_{\_0} \);
prove;
assert \( \mathcal{P}(X \times X) \);
prove;
assert \( \mathcal{P}(X \times X) \);
prove;
apply compDef;
prove;
apply inBigcup;
prove;
split \( \{ n_{\_0} : N \bullet \iter[X] n_{\_0} \in R \} \in \mathcal{P}(X \times X) \);
cases;
prove;
split \( \{ n_{\_3} : N \bullet \iter[X] n_{\_3} \in R \} \in \mathcal{P}(X \times X) \);
cases;
prove;
apply compDef;
prove;
apply inBigcup;
prove;
assert \( \{ n_{\_2} : N \bullet \iter[X] n_{\_2} \in R \} \equiv \iter[X](n+1) \);
prove;
assert \( n_{\_0} \equiv n+1 \);
prove;
apply iterUnfoldEnd to expression \( \iter[X] (1+n) \);
prove;
assert \( y_{\_3} \equiv y \);
prove;
next;
apply inPower;
prove;
next;
apply inPower;
prove;
next;
\qed
proof[inclusion2]
prove;
apply inPower to predicate
\[\bigcup \{ \bigcup X \mid (R, \bigcup \{ \bigcup \}) [X, X] (\bigcup \{ \bigcup \}) [X, R, S]) \}\in P \bigcup \{ \bigcup X \mid (R, \bigcup \{ \bigcup \}) [X, X] (\bigcup \{ \bigcup \}) [X, R, S])\};
prove;
apply plusUnfoldEnd to expression \(\bigcup \{ \bigcup X \mid (R, \bigcup \{ \bigcup \}) [X, X] (\bigcup \{ \bigcup \}) [X, R, S])\);
prove;
use compMonotone \[X, X, X]\]
prove;
use inclusion1[X];
prove;
instantiate \(e_0 \equiv e\);
prove;
apply moveNrresBackwards;
prove;
apply plusUnfoldEnd;
prove;

proof[nrresPlusDistrib]
apply extensionality2;
use inclusion1[X];
inclusion2[X];
prove;

D.2.3.3 Ranges

proof[diffRangeStart]
apply rangeDef;
prove;
apply diffSuperset to expression \(\{ k: Z \mid a \leq k \land k \leq b \} \setminus \{ k_0: Z \mid a \leq k_0 \land k_0 \leq c \}\);;
prove;
cases;
apply extensionality to predicate \(\);;
prove;
next;
apply inPower;
prove;
instantiate \(e_0 \equiv e\);
rewrite;
next;
apply extensionality;
prove;

D.2.3.4 Finiteness

proof[diffFinite]
prove;
proof[diffCupCupCupDiff]
prove by reduce;
apply cupPermutes;
simplify;
apply cupDiff;
rewrite;
apply cupPermutes;
simplify;
apply cupDiff;
rewrite;
apply cupCommutes to expression \((\_ \cup \_)[X](C ,D)\);
simplify;
apply cupPermutes to expression \((\_ \cup \_)[X](B , (\_ \cup \_)[X](D ,C))\);
simplify;
apply cupSubsetLeft to expression \((\_ \cup \_)[X](D , (\_ \cup \_)[X](B ,C))\);
rewrite;
apply cupPermutes;
simplify;
apply cupSubsetRight to expression \((\_ \cup \_)[X](A ,C)\);
rewrite;

proof[finiteDef]
split \(S \in F X\);
cases;
invoke \(F _{\_}\);
prove;
instantiate \(n_{\_0} == n\);
instantiate \(f_{\_0} == f\);
prove;
next;
prove;
invoke \(F _{\_}\);
trivial rewrite;
instantiate \(S_{\_0} == S\);
instantiate \(n_{\_0} == n\);
instantiate \(f_{\_0} == f\);
prove;
next;

proof[finiteRelPlusImageIsFinite]
prove;

proof[finiteSubsetIsFinite]
prove;

D.2.3.5 Cardinality

proof[cupCard]
use cardCup[X];
prove;

proof[cardCupDisjoint]
use cardCup[X][S := A , T := B];
prove;

proof[plusCardDisjoint]
apply cardCupDisjoint;
prove;
D.2.3.6 Finite Function Spaces

proof[finiteFunctionDom]
use functionFinite[X, Y][A := X, B := Y];
prove;

proof[ndresInFfun]
prove;

proof[oplusInFfun]
apply oplusDef;
prove;

proof[singletonInFfun]
prove;

D.3 Memory Manager Theorems

section memorymanager_theorems parents memorymanager, additional_lemmas

| bsidWitness : BackingStoreID

theorem BackingStoreID_vc_fsb_given_para
¬ BackingStoreID = {}

theorem BackingStoreIDElement
∃ x : BackingStoreID • true

theorem grule MemoryAddressMaxType
∀ x : MemoryAddress • x ∈ Z

theorem disabled rule MemoryAddressDef
x ∈ MemoryAddress ⇔ x ∈ Z ∧ x ≥ 0

theorem grule addMemoryAddress
∀ x, y : MemoryAddress • x + y ∈ MemoryAddress

theorem rule decMemoryAddress
∀ x : MemoryAddress • x ≥ 1 ⇔ x − 1 ∈ MemoryAddress

theorem grule finiteMemoryAddressMaxType
∀ x : F MemoryAddress • x ∈ ℤ
\textbf{theorem} grule finiteMemoryAddressIsFiniteNum
\begin{align*}
\forall x : \mathbb{F} \text{MemoryAddress} \Rightarrow x \in \mathbb{F} \mathbb{Z}
\end{align*}

\textbf{theorem} grule ContiguousMemoryMaxType
\begin{align*}
\forall x : \text{ContiguousMemory} \Rightarrow x \in \mathbb{P} \mathbb{Z}
\end{align*}

\textbf{theorem} grule ContiguousMemoryIsPowerMemoryAddress
\begin{align*}
\forall x : \text{ContiguousMemory} \Rightarrow x \in \mathbb{P} \text{MemoryAddress}
\end{align*}

\textbf{theorem} grule ContiguousMemoryIsFiniteMemoryAddress
\begin{align*}
\forall x : \text{ContiguousMemory} \Rightarrow x \in \mathbb{F} \text{MemoryAddress}
\end{align*}

\textbf{theorem} grule ContiguousMemoryIsFiniteNum
\begin{align*}
\forall x : \text{ContiguousMemory} \Rightarrow x \in \mathbb{F} \mathbb{Z}
\end{align*}

\textbf{theorem} disabled rule inContiguousMemory
\begin{align*}
x \in \text{ContiguousMemory} \iff (x \in \mathbb{P} \text{MemoryAddress} \land (\exists a, b : \text{MemoryAddress} \Rightarrow x = a \ldots b))
\end{align*}

\textbf{theorem} rule rangeInContiguousMemory
\begin{align*}
\forall a, b : \text{MemoryAddress} \Rightarrow a \ldots b \in \text{ContiguousMemory}
\end{align*}

\textbf{theorem} grule nullInContiguousMemory
\begin{align*}
\{\} \in \text{ContiguousMemory}
\end{align*}

\textbf{theorem} disabled splitContiguousMemory
\begin{align*}
\forall A : \text{ContiguousMemory}; s : \mathbb{N} \mid s \leq \# A \Rightarrow (\exists B, C : \text{ContiguousMemory} \Rightarrow A = B \cup C \land B \cap C = \{\} \land \# B = s)
\end{align*}

\textbf{MemoryBlockFSBSig}

\begin{itemize}
\item MemoryBlock
\item MemoryBlock
\end{itemize}

\textbf{theorem} MemoryBlock\_vc\_fsb\_state
\begin{align*}
\exists \text{MemoryBlockFSBSig} \mid \text{true} \cdot \text{true}
\end{align*}

\textbf{MemoryBlockInitFSBSig}

\begin{itemize}
\item MemoryBlock'
\item addresses? : \text{ContiguousMemory}
\end{itemize}

\textbf{true}

\textbf{theorem} MemoryBlockInit\_vc\_fsb\_state
\begin{align*}
\exists \text{MemoryBlockInitFSBSig} \mid \text{true} \cdot \text{true}
\end{align*}
\begin{align*}
\text{MBAllocateFSBSig} & \quad \text{MemoryBlock} \\
& \quad \text{size?} : \mathbb{N} \\
& \quad \text{size?} \leq \# \text{ free}
\end{align*}

\textbf{theorem} MBAllocate\_vc\_fsb\_pre
\begin{align*}
\forall \text{MBAllocateFSBSig} \mid \text{true} \bullet \text{pre MBAllocate}
\end{align*}

\begin{align*}
\text{MBClearFSBSig} & \quad \text{MemoryBlock} \\
& \quad \text{true}
\end{align*}

\textbf{theorem} MBClear\_vc\_fsb\_pre
\begin{align*}
\forall \text{MBClearFSBSig} \mid \text{true} \bullet \text{pre MBClear}
\end{align*}

\begin{align*}
\text{MBResizeFSBSig} & \quad \text{MBClearFSBSig} \\
& \quad \text{newAddresses?} : \text{ContiguousMemory} \\
& \quad \# \text{ newAddresses?} \geq \# \text{ total} - \# \text{ free} \\
& \quad \text{used = } \emptyset
\end{align*}

\textbf{theorem} MBResize\_vc\_fsb\_pre
\begin{align*}
\forall \text{MBResizeFSBSig} \mid \text{true} \bullet \text{pre MBResize}
\end{align*}

\begin{align*}
\text{MBContentsResizeFSBSig} & \quad \text{MBClearFSBSig} \\
& \quad \text{newSize?} : \mathbb{N} \\
& \quad \text{oldAddresses?} : \text{ContiguousMemory} \\
& \quad \text{oldAddresses?} \subseteq \text{used} \\
& \quad \text{newSize?} \leq \#(\text{free} \cup \text{oldAddresses?}) \\
& \quad \text{free} \cup \text{oldAddresses?} \in \text{ContiguousMemory}
\end{align*}

\textbf{theorem} MBContentsResize\_vc\_fsb\_pre
\begin{align*}
\forall \text{MBContentsResizeFSBSig} \mid \text{true} \bullet \text{pre MBContentsResize}
\end{align*}
\( \text{MBGetTotalSizeFSBSig} \)
\( \text{MBClearFSBSig} \)
\( \text{true} \)

**theorem** MBGetTotalSize vc fsb pre
\( \forall \text{MBGetTotalSizeFSBSig} \mid \text{true} \bullet \text{pre MBGetTotalSize} \)

\( \text{MBGetUsedSizeFSBSig} \)
\( \text{MBClearFSBSig} \)
\( \text{true} \)

**theorem** MBGetUsedSize vc fsb pre
\( \forall \text{MBGetUsedSizeFSBSig} \mid \text{true} \bullet \text{pre MBGetUsedSize} \)

\( \text{MBGetFreeSizeFSBSig} \)
\( \text{MBClearFSBSig} \)
\( \text{true} \)

**theorem** MBGetFreeSize vc fsb pre
\( \forall \text{MBGetFreeSizeFSBSig} \mid \text{true} \bullet \text{pre MBGetFreeSize} \)

\( \text{SuccessFSBSig} \)
\( \text{Success} \)
\( \text{Success} \)

**theorem** Success vc fsb state
\( \exists \text{SuccessFSBSig} \mid \text{true} \bullet \text{true} \)

\( \text{MBOOutOfMemoryFSBSig} \)
\( \text{MBClearFSBSig} \)
\( \text{size} \? : \mathbb{N} \)
\( \neg \text{size} \? \leq \# \text{free} \)

**theorem** MBOOutOfMemory vc fsb pre
\( \forall \text{MBOOutOfMemoryFSBSig} \mid \text{true} \bullet \text{pre MBOOutOfMemory} \)
\_\text{MBNotEmptyFSBSig}\_ \\
\text{MBClearFSBSig} \\
used \neq \emptyset \\

\textbf{theorem} \text{MBNotEmpty}_\text{vc}_\text{fsb}_\text{pre} \\
\forall \text{MBNotEmptyFSBSig} \mid true \bullet \text{pre MBNotEmpty} \\

\_\text{MBResizeBellowOverheadFSBSig}\_ \\
\text{MBClearFSBSig} \\
newAddresses? : \text{ContiguousMemory} \\
\neg \# newAddresses? \geq \# total - \# free \\

\textbf{theorem} \text{MBResizeBellowOverhead}_\text{vc}_\text{fsb}_\text{pre} \\
\forall \text{MBResizeBellowOverheadFSBSig} \mid true \bullet \text{pre MBResizeBellowOverhead} \\

\_\text{MBNonexistentAllocationFSBSig}\_ \\
\text{MBClearFSBSig} \\
oldAddresses? : \text{ContiguousMemory} \\
\neg (oldAddresses? \subseteq used) \\

\textbf{theorem} \text{MBNonexistentAllocation}_\text{vc}_\text{fsb}_\text{pre} \\
\forall \text{MBNonexistentAllocationFSBSig} \mid true \bullet \text{pre MBNonexistentAllocation} \\

\_\text{MBInsufficientFreeSizeFSBSig}\_ \\
\text{MBClearFSBSig} \\
newSize? : \mathbb{N} \\
oldAddresses? : \text{ContiguousMemory} \\
\neg newSize? \leq \#(\text{free} \cup \text{oldAddresses}?) \\

\textbf{theorem} \text{MBInsufficientFreeSize}_\text{vc}_\text{fsb}_\text{pre} \\
\forall \text{MBInsufficientFreeSizeFSBSig} \mid true \bullet \text{pre MBInsufficientFreeSize}
\[\text{MBNotContiguousWithFreeFSBSig} \]
\[\text{MBClearFSBSig} \]
\[\text{oldAddresses} : \text{ContiguousMemory} \]
\[\text{free} \cup \text{oldAddresses} \notin \text{ContiguousMemory} \]

**Theorem**  \[\text{MBNotContiguousWithFree}_\text{vc}_\text{fsb}_\text{pre} \]  
\[\forall \text{MBNotContiguousWithFreeFSBSig} | \text{true} \cdot \text{pre MBNotContiguousWithFree} \]

\[\text{RMBAllocateFSBSig} \]
\[\text{MBClearFSBSig} \]
\[\text{size} : \mathbb{N} \]
\[\text{size} : \mathbb{N} \]
\[\text{true} \]

**Theorem**  \[\text{RMBAllocate}_\text{vc}_\text{fsb}_\text{pre} \]  
\[\forall \text{RMBAllocateFSBSig} | \text{true} \cdot \text{pre RMBAllocate} \]

\[\text{RMBClearFSBSig} \]
\[\text{MBClearFSBSig} \]
\[\text{true} \]

**Theorem**  \[\text{RMBClear}_\text{vc}_\text{fsb}_\text{pre} \]  
\[\forall \text{RMBClearFSBSig} | \text{true} \cdot \text{pre RMBClear} \]

\[\text{RMBResizeFSBSig} \]
\[\text{MBClearFSBSig} \]
\[\text{newAddresses} : \text{ContiguousMemory} \]
\[\text{newAddresses} : \text{ContiguousMemory} \]
\[\text{true} \]

**Theorem**  \[\text{RMBResize}_\text{vc}_\text{fsb}_\text{pre} \]  
\[\forall \text{RMBResizeFSBSig} | \text{true} \cdot \text{pre RMBResize} \]
\textbf{theorem} RMBContentsResize\_vc\_fsb\_pre
\[\forall\ RMBContentsResizeFSBSig \mid true \bullet \text{pre} \ RMBContentsResize\]

\textbf{theorem} RMBGetFreeSize\_vc\_fsb\_pre
\[\forall\ RMBGetFreeSizeFSBSig \mid true \bullet \text{pre} \ RMBGetFreeSize\]

\textbf{theorem} RMBGetTotalSize\_vc\_fsb\_pre
\[\forall\ RMBGetTotalSizeFSBSig \mid true \bullet \text{pre} \ RMBGetTotalSize\]

\textbf{theorem} RMBGetUsedSize\_vc\_fsb\_pre
\[\forall\ RMBGetUsedSizeFSBSig \mid true \bullet \text{pre} \ RMBGetUsedSize\]

\textbf{theorem} BackingStore\_vc\_fsb\_pre
\[\forall\ BackingStoreFSBSig \mid true \bullet \text{pre} \ BackingStore\]
**Theorem** BackingStore\_init\_pre
\[\forall \text{addresses}?: \text{ContiguousMemory} \mid \text{backingStoreOverhead} \leq \# \text{addresses}? \bullet \exists \text{BackingStore} \bullet \text{true}\]

**Theorem** BackingStore\_init\_pre
\[\forall \text{addresses}?: \text{ContiguousMemory} \mid \text{backingStoreOverhead} \leq \# \text{addresses}? \bullet \exists \text{BackingStore} \bullet \text{true}\]

**BSAllocateChild\_FSBSig**

\[\text{Backin\_Store}\]
\[\text{size}?: \mathbb{N}\]

**pre MBAllocate**
\[\text{size}?: \leq \text{backingStoreOverhead}\]
\[\exists \text{childID}?: \text{BackingStoreID} \bullet \text{childID}? \notin \text{children} \land \text{childID}? \neq \text{self}\]

**Theorem** BSAllocateChild\_vc\_fsb\_pre
\[\forall \text{BSAllocateChild\_FSBSig} \mid \text{true} \bullet \text{pre BSAllocateChild}\]

**BSClear\_FSBSig**

\[\text{Backin\_Store}\]
\[\text{true}\]

**Theorem** BSClear\_vc\_fsb\_pre
\[\forall \text{BSClear\_FSBSig} \mid \text{true} \bullet \text{pre BSClear}\]

**BSResizeChild\_FSBSig**

\[\text{BSClear\_FSBSig}\]
\[\text{newSize}?: \mathbb{N}\]
\[\text{oldAddresses}?: \text{ContiguousMemory}\]
\[\text{oldID}?: \text{BackingStoreID}\]

**MBContentsResize\_FSBSig**
\[\text{oldID}?: \in \text{children}\]

**Theorem** BSResizeChild\_vc\_fsb\_pre
\[\forall \text{BSResizeChild\_FSBSig} \mid \text{true} \bullet \text{pre BSResizeChild}\]

**BSAllocate\_FSBSig**

\[\text{BSClear\_FSBSig}\]
\[\text{size}?: \mathbb{N}\]

\[\exists \text{actualSize}?: \mathbb{N} \mid \text{actualSize} = \text{size}? + \text{allocationOverhead}\]

\[\text{pre MBAllocate}[\text{actualSize} / \text{size}?]\]
\textbf{theorem} BSAllocate\_ vc\_ fsb\_ pre
\[ \forall \text{BSAllocateFSBSig} \mid \text{true} \bullet \text{pre BSAllocate} \]

\begin{align*}
\text{BSResizeFSBSig} \\
\text{BSClearFSBSig} \\
\text{newAddresses? : ContiguousMemory} \\
\text{pre MBResize} \\
\# \text{newAddresses?} \geq \text{backingStoreOverhead} \\
\text{children} = \emptyset
\end{align*}

\textbf{theorem} BSResize\_ vc\_ fsb\_ pre
\[ \forall \text{BSResizeFSBSig} \mid \text{true} \bullet \text{pre BSResize} \]

\begin{align*}
\text{BSGetTotalSizeFSBSig} \\
\text{BSClearFSBSig} \\
\text{true}
\end{align*}

\textbf{theorem} BSGetTotalSize\_ vc\_ fsb\_ pre
\[ \forall \text{BSGetTotalSizeFSBSig} \mid \text{true} \bullet \text{pre BSGetTotalSize} \]

\begin{align*}
\text{BSGetFreeSizeFSBSig} \\
\text{BSClearFSBSig} \\
\text{true}
\end{align*}

\textbf{theorem} BSGetFreeSize\_ vc\_ fsb\_ pre
\[ \forall \text{BSGetFreeSizeFSBSig} \mid \text{true} \bullet \text{pre BSGetFreeSize} \]

\begin{align*}
\text{BSGetUsedSizeFSBSig} \\
\text{BSClearFSBSig} \\
\text{true}
\end{align*}

\textbf{theorem} BSGetUsedSize\_ vc\_ fsb\_ pre
\[ \forall \text{BSGetUsedSizeFSBSig} \mid \text{true} \bullet \text{pre BSGetUsedSize} \]
theorem BSNotEmpty_{vc_fsb_pre}
\forall BSNotEmptyFSBSig \mid \text{true} \bullet \text{pre } BSNotEmpty

theorem BSNonexistentAllocation_{vc_fsb_pre}
\forall BSNonexistentAllocationFSBSig \mid \text{true} \bullet \text{pre } BSNonexistentAllocation

theorem BSSizeTooSmall_{vc_fsb_pre}
\forall BSSizeTooSmallFSBSig \mid \text{true} \bullet \text{pre } BSSizeTooSmall

theorem BSNoFreeIdentifier_{vc_fsb_pre}
\forall BSNoFreeIdentifierFSBSig \mid \text{true} \bullet \text{pre } BSNoFreeIdentifier

theorem BSOutOfMemory_{vc_fsb_pre}
\forall BSOutOfMemoryFSBSig \mid \text{true} \bullet \text{pre } BSOutOfMemory
theorem BSOutOfMemory vc fsb pre
\( \forall BSOutOfMemoryFSBSig \mid true \bullet pre BSOutOfMemory \)

\( _RBSAllocateChildFSBSig \)
BSClearFSBSig
size? : N
size? : N
true

theorem RBSAllocateChild vc fsb pre
\( \forall RBSAllocateChildFSBSig \mid true \bullet pre RBSAllocateChild \)

\( _RBSClearFSBSig \)
BSClearFSBSig
true

theorem RBSClear vc fsb pre
\( \forall RBSClearFSBSig \mid true \bullet pre RBSClear \)

\( _RBSResizeChildFSBSig \)
BSClearFSBSig
newSize? : N
newSize? : N
oldAddresses? : ContiguousMemory
oldAddresses? : ContiguousMemory
oldAddresses? : ContiguousMemory
oldAddresses? : ContiguousMemory
oldID? : BackingStoreID
oldID? : BackingStoreID
true

theorem RBSResizeChild vc fsb pre
\( \forall RBSResizeChildFSBSig \mid true \bullet pre RBSResizeChild \)

\( _RBSAllocateFSBSig \)
BSClearFSBSig
size? : N
size? : N
true
\textbf{theorem} \textit{RBSAllocate} \_ \textit{vc} \_ \textit{fsb} \_ \textit{pre}
\[ \forall \text{RBSAllocateFSBSig} \mid \text{true} \bullet \text{pre RBSAllocate} \]

\textbf{RBSResizeFSBSig}
\begin{itemize}
  \item \textit{BSClearFSBSig}
  \item \textit{newAddresses?} : \textit{ContiguousMemory}
  \item \textit{newAddresses?} : \textit{ContiguousMemory}
  \item \text{true}
\end{itemize}

\textbf{theorem} \textit{RBSResize} \_ \textit{vc} \_ \textit{fsb} \_ \textit{pre}
\[ \forall \text{RBSResizeFSBSig} \mid \text{true} \bullet \text{pre RBSResize} \]

\textbf{RBSGetFreeSizeFSBSig}
\begin{itemize}
  \item \textit{BSClearFSBSig}
  \item \text{true}
\end{itemize}

\textbf{theorem} \textit{RBSGetFreeSize} \_ \textit{vc} \_ \textit{fsb} \_ \textit{pre}
\[ \forall \text{RBSGetFreeSizeFSBSig} \mid \text{true} \bullet \text{pre RBSGetFreeSize} \]

\textbf{RBSGetTotalSizeFSBSig}
\begin{itemize}
  \item \textit{BSClearFSBSig}
  \item \text{true}
\end{itemize}

\textbf{theorem} \textit{RBSGetTotalSize} \_ \textit{vc} \_ \textit{fsb} \_ \textit{pre}
\[ \forall \text{RBSGetTotalSizeFSBSig} \mid \text{true} \bullet \text{pre RBSGetTotalSize} \]

\textbf{RBSGetUsedSizeFSBSig}
\begin{itemize}
  \item \textit{BSClearFSBSig}
  \item \text{true}
\end{itemize}

\textbf{theorem} \textit{RBSGetUsedSize} \_ \textit{vc} \_ \textit{fsb} \_ \textit{pre}
\[ \forall \text{RBSGetUsedSizeFSBSig} \mid \text{true} \bullet \text{pre RBSGetUsedSize} \]

\textbf{GlobalMemoryManagerFSBSig}
\begin{itemize}
  \item \textit{GlobalMemoryManager}
  \item \textit{GlobalMemoryManager}
\end{itemize}

\textbf{theorem} \textit{GlobalMemoryManager} \_ \textit{vc} \_ \textit{fsb} \_ \textit{state}
\[ \exists \text{GlobalMemoryManagerFSBSig} \mid \text{true} \bullet \text{true} \]

242
D.4 Proofs of Memory Manager Theorems

section memorymanager_proofs parents memorymanager_theorems

proof[BackingStoreID vc fsb given para]
apply extensionality to predicate BackingStoreID = {};
prove;
  instantiate z == bsidWitness;
  rewrite;

proof[BackingStoreIDElement]
  instantiate z == bsidWitness;
  prove;

proof[MemoryAddressMaxType]
  prove by reduce;

proof[MemoryAddressDef]
  prove by reduce;

proof[addMemoryAddress]
  prove by reduce;

proof[decMemoryAddress]
  prove by reduce;

proof[finiteMemoryAddressMaxType]
  prove by reduce;

proof[finiteMemoryAddressIsFiniteNum]
  prove by reduce;

proof[ContiguousMemoryMaxType]
  prove by reduce;

proof[ContiguousMemoryIsPowerMemoryAddress]
  prove by reduce;

proof[ContiguousMemoryIsFiniteMemoryAddress]
  prove by reduce;

proof[ContiguousMemoryIsFiniteNum]
  prove by reduce;
proof\[\text{inContiguousMemory}\]
prove by reduce;

proof\[\text{rangeInContiguousMemory}\]
prove by reduce;
instantiate \(a_0\) == \(a\), \(b_0\) == \(b\);
prove;

proof\[\text{nullInContiguousMemory}\]
apply \text{inContiguousMemory};
instantiate \(a\) == 2, \(b\) == 1;
prove by reduce;

proof\[\text{splitContiguousMemory}\]
\(s = 0\);
cases;
instantiate \(B\) == \(\emptyset\), \(C\) == \(A\);
prove by reduce;
instantiate \(a_0\) == 2, \(b_0\) == 1;
rewrite;
next;
apply \text{inContiguousMemory};
\text{prenex};
use \text{rangeSplits}\[a := a, b := a + s - 1, c := b;\]
rewrite;
instantiate \(B\) == \(a \ldots a + s - 1\), \(C\) == \(a + s \ldots b\);
rewrite;
equality substitute;
apply \text{sizeRange to expression} \# (a \ldots b);
rewrite;
\text{rearrange};
apply \text{MemoryAddressDef};
rewrite;
apply \text{rangeInContiguousMemory};
prove;
apply \text{MemoryAddressDef};
rewrite;
apply \text{extensionality};
prove;
next;

D.5 Memory Blocks

proof\[\text{MemoryBlock\_vc\_fsb\_state}\]
instantiate total == \(\emptyset\), free == \(\emptyset\), used == \(\emptyset\);
prove by reduce;
instantiate \(a\) == 1, \(b\) == 0;
prove;

proof\[\text{MemoryBlockInit\_vc\_fsb\_state}\]
instantiate free' == \(\emptyset\), addresses' == \(\emptyset\), total' == \(\emptyset\), used' == \(\emptyset\);
prove by reduce;
instantiate \(a\) == 1, \(b\) == 0;
prove;

proof\[\text{MBAllocate\$domainCheck}\]
prove by reduce;

244
proof[MBAllocateFSBSig\$domainCheck]
prove by reduce;

proof[MBAllocate\_ wc\_ fsb\_ pre]
invoke MBAllocate;
invoke MBAllocateFSBSig;
prove by reduce;
invoke MemoryBlock;
prove;
use splitContiguousMemory[A := free, s := size?];
prove;
rewrite;
prove;
instantiate allocated != B;
prove;
apply inPower to predicate B ∈ P free;
prove;
apply distributeDiffOverCupLeft to expression B ∪ C \ B;
prove;
split C ∈ P Z;
cases;
prove;
apply diffSuperset to expression B \ C;
rewrite;
split B ∈ P Z;
cases;
prove;
split used ∈ P Z;
cases;
use diffDisjoint[2][A := C, B := B];
prove;
apply distributeCupOverCupRight to expression C ∩ (used ∪ B);
rewrite;
apply distributeCupOverCupRight to expression used ∩ (B ∪ C);
rewrite;
next;
prove by reduce;
next;
prove by reduce;
next;
prove by reduce;
next;

proof[MBClear\_ wc\_ fsb\_ pre]
instantiate total' == total, free' == used ∪ free, used' == {};
prove;
invoke MBClear;
invoke Δ MemoryBlock;
invoke MemoryBlock;
prove;
invoke MBClearFSBSig;
invoke MemoryBlock;
prove;

proof[MBResize\$domainCheck]
prove by reduce;

proof[MBResizeFSBSig\$domainCheck]
prove by reduce;
proof[MBResize vc fsb pre]
use splitContiguousMemory[A := newAddresses?, s := # total − # free];
prenex ;
rearrange;
invoke MBResizeFSBSig;
invoke MBClearFSBSig;
invoke MemoryBlock ;
rewrite;
use sizeOfSubset[Z][S := free, T := total];
rearrange;
rewrite;
split total ∈ F Z;
cases;
rewrite;
instantiate free 0 == C ;
invoke MBResize;
invoke ∆ MemoryBlock ;
invoke MemoryBlock ;
prove;
apply cupSubset to predicate B ∪ C ∈ P Z;
apply subsetCup to predicate C ∈ P (B ∪ C );
apply cupNullLeft to expression {} ∪ free;
apply capNullLeft to expression {} ∩ free;
rearrange;
rewrite;
split free ∈ P Z;
cases;
split B ∈ P Z;
cases;
rewrite;
apply cupCard to expression # (B ∪ C );
split B ∈ F Z;
cases;
split C ∈ F Z;
cases;
rewrite;
split B ∈ P Z;
cases;
split C ∈ P Z;
cases;
prove;
next;
prove by reduce;
next;
prove by reduce;
next;
prove by reduce;
next;
prove by reduce;
next;
prove by reduce;
next;
prove by reduce;
next;
prove by reduce;
next;

proof[MBContentsResize$domainCheck ]
prove by reduce;

proof[MBContentsResizeFSBSig$domainCheck ]
prove by reduce;

246


proof[MBContentsResize\_vc\_fsb\_pre]
use MBAllocate\_vc\_fsb\_pre
[free := free \cup oldAddresses?, used := used \setminus oldAddresses?, size := newSize?];
prove;
imvoke MBContentsResizeFSBSig;
imvoke MBAllocateFSBSig;
imvoke MBClearFSBSig;
imvoke MemoryBlock;
prove;
apply cupPermutes to expression used \setminus oldAddresses? \cup (free \cup oldAddresses?)
rewrite;
split oldAddresses? \in \mathbb{P} Z;
cases;
rewrite;
apply cupSubsetLeft to expression oldAddresses? \cup used;
rewrite;
apply distributeCapOverCupRight to expression (used \setminus oldAddresses?) \cap (free \cup oldAddresses?)
rewrite;
split free \in \mathbb{P} Z;
cases;
rewrite;
use capDiffDisjoint\[Z\][A := free, B := used, C := oldAddresses?];
rearrange;
rewrite;
split oldAddresses? \in \mathbb{P} total;
cases;
prove;
instantiate free' \_0' == free', newAddresses! == allocated!, total' \_0' == total', used' \_0' == used';
imvoke MBContentsResize;
imvoke MBAllocate;
imvoke Δ MemoryBlock;
imvoke MemoryBlock;
prove;
next;
apply inPower to predicate oldAddresses? \in \mathbb{P} total;
apply inPower to predicate oldAddresses? \in \mathbb{P} used;
apply inPower to predicate used \in \mathbb{P} total;
prove;
instantiate e\_1 \_1 == e;
instantiate e\_2 \_2 == e;
prove;
next;
prove by reduce;
next;
prove by reduce;
next;

proof[MBGetTotalSize\_\$domainCheck]
prove by reduce;

proof[MBGetUsedSize\_\$domainCheck]
prove by reduce;

proof[MBGetFreeSize\_\$domainCheck]
prove by reduce;

proof[MBGetTotalSize\_\_vc\_fsb\_pre]
instantiate free' \_0 == free, used' \_0 == used, total' \_0 == total, size! == \# total;
imvoke MBGetTotalSizeFSBSig;
imvoke MBGetTotalSize;
imvoke MBClearFSBSig;
imvoke Ξ MemoryBlock;
imvoke MemoryBlock;
prove by reduce;
proof MBGetUsedSize vc fsb pre
  instantiate free' == free, used' == used, total' == total, size! == # used;
  invoke MBGetUsedSizeFSBSig;
  invoke MBGetUsedSize;
  invoke Ξ MemoryBlock;
  invoke MemoryBlock;
  use cardIsNonNegative[\mathbb{Z}]; S := used];
  prove;

proof MBGetFreeSize vc fsb pre
  instantiate free' == free, used' == used, total' == total, size! == # free;
  invoke MBGetFreeSizeFSBSig;
  invoke MBGetFreeSize;
  invoke MBClearFSBSig;
  invoke Ξ MemoryBlock;
  invoke MemoryBlock;
  prove by reduce;

proof Success vc fsb state]
  instantiate report! == MMokay;
  prove by reduce;

proof MBOutOfMemory$domainCheck]
  prove by reduce;

proof MBOutOfMemoryFSBSig$domainCheck]
  prove by reduce;

proof MBOutOfMemory vc fsb pre]
  invoke MBOutOfMemory;
  invoke Ξ MemoryBlock;
  invoke MBOutOfMemoryFSBSig;
  invoke MBClearFSBSig;
  invoke MemoryBlock;
  prove;

proof MBNotEmpty vc fsb pre]
  invoke MBNotEmpty;
  invoke Ξ MemoryBlock;
  invoke MBNotEmptyFSBSig;
  invoke MBClearFSBSig;
  invoke MemoryBlock;
  prove;

proof MBResizeBellowOverhead$domainCheck]
  prove by reduce;

proof MBResizeBellowOverheadFSBSig$domainCheck]
  prove by reduce;
proof[MBResizeBelowOverhead\_\_vc\_fsb\_pre]  
invoke MBResizeBelowOverhead;  
invoke MBResizeBelowOverheadFSBSig;  
invoke MBClearFSBSig;  
invoke MemoryBlock;  
prove;

proof[MBNonexistentAllocation\_\_vc\_fsb\_pre]  
invoke MBNonexistentAllocation;  
invoke MBNonexistentAllocationFSBSig;  
invoke MBClearFSBSig;  
invoke MemoryBlock;  
prove;

proof[MBInsufficientFreeSize\_\_domainCheck]  
prove by reduce;

proof[MBInsufficientFreeSizeFSBSig\_\_domainCheck]  
prove by reduce;

proof[MBInsufficientFreeSize\_\_vc\_fsb\_pre]  
invoke MBInsufficientFreeSize;  
invoke MBInsufficientFreeSizeFSBSig;  
invoke MBClearFSBSig;  
invoke MemoryBlock;  
prove;

proof[MBNotContiguousWithFree\_\_vc\_fsb\_pre]  
invoke MBNotContiguousWithFreeFSBSig;  
invoke MBClearFSBSig;  
invoke MBNotContiguousWithFree;  
invoke MemoryBlock;  
instantiate free' == free, report! == MMnotContiguousWithFree, used' == used, total' == total;  
prove;

proof[RMBAllocate\_\_vc\_fsb\_pre]  
use RMBAllocate;  
use RMBAllocateFSBSig;  
invoke RMBAllocate;  
invoke RMBAllocateFSBSig;  
invoke MBOutOfMemoryFSBSig;  
invoke MBClearFSBSig;  
invoke MBOutOfMemory;  
invoke MemoryBlock;  
prove;  
split size? $\leq$ \# free;  
cases:  
invoke Success;  
instantiate allocated\_0 == allocated\_0;  
prove;  
next;  
instantiate allocated\_1 == \{\};  
prove;  
next;
\textbf{proof[RMBClear\_vc\_fsb\_pre]} 
use MBClear\_vc\_fsb\_pre; 
prove; 
\text{instantiate free\_0' == free', total\_0' == total', used\_0' == used', report! == MMokay}; 
prove by reduce; 

\textbf{proof[RMBResize\_vc\_fsb\_pre]} 
invoke RMBResize; 
use MBResize\_vc\_fsb\_pre; 
use MBNotEmpty\_vc\_fsb\_pre; 
use MBResizeBellowOverhead\_vc\_fsb\_pre; 
prove; 
invoke MBResizeBellowOverheadFSBSig; 
invoke MBNotEmptyFSBSig; 
invoke MBResizeFSBSig; 
invoke RMBResizeFSBSig; 
\text{split used = \{\}}; 
cases; 
\text{split # newAddresses? \geq -1 * # free + # total}; 
cases; 
\text{instantiate free\_3' == free', oldAddresses\_0! == oldAddresses!, report\_1! == MMokay, total\_3' == total\_1', used\_3' == used\_1'}; 
prove; 
\text{next}; 
\text{instantiate free\_3' == free', oldAddresses\_1! == \{\}, report\_2! == report!, total\_3' == total', used\_3' == used'; 
prove; 
\text{next}; 
\text{instantiate free\_3' == free\_0', oldAddresses\_1! == \{\}, report\_2! == report\_0', 
\text{total\_3' == total\_0', used\_3' == used\_0'}; 
prove; 
\text{next};
proof[RMBContentsResize vc fsb pre]
  invoke RMBContentsResize;
  use MBContentsResize vc fsb pre;
  use MBNonexistentAllocation vc fsb pre;
  use MBInsufficientFreeSize vc fsb pre;
  use MBNotContiguousWithFree vc fsb pre;
  prove;
  invoke MBNotContiguousWithFreeFSBSig;
  invoke MBInsufficientFreeSizeFSBSig;
  invoke MBNonexistentAllocationFSBSig;
  invoke MBContentsResizeFSBSig;
  invoke RMBContentsResizeFSBSig;
  invoke MBClearFSBSig;
  split oldAddresses ? ⊆ used;
  cases;
  split newSize ? ≤ # (free ∪ oldAddresses ?);
  cases;
  instantiate free _0' == free', newAddresses _0! == newAddresses!, report _0! == MMokay,
  total _0' == total _0', used _0' == used _2';
  prove;
  next;
  instantiate free _0' == free', newAddresses _1! == {}, report _0! == report!,
  total _0' == total _0', used _0' == used _2';
  split free ∪ oldAddresses ? ∈ P Z;
  cases;
  split ContiguousMemory ∈ P P Z;
  cases;
  prove;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next;
  instantiate free _0' == free _0', newAddresses _0! == {}, report _0! == report _0',
  total _0' == total _0', used _0' == used _0';
  prove;
  next;
  instantiate free _0' == free _0', newAddresses _1! == {}, report _0! == report _1',
  total _0' == total _1', used _0' == used _1';
  prove;
  next;
proof[RMBGetFreeSize vc fsb pre]
  use MBGetFreeSize vc fsb pre;
  invoke RMBGetFreeSizeFSBSig;
  invoke MBGetFreeSizeFSBSig;
  invoke MBClearFSBSig;
  prove;
  instantiate free _0' == free _0', used _0' == used _0', total _0' == total _0',
  size _0! == size _0', report _0! == MMokay;
  invoke RMBGetFreeSize;
  invoke Success;
  invoke
proof[RMBGetTotalSize vc fsb pre]
  use MBGetTotalSize vc fsb pre;
  invoke RMBGetTotalSizeFSBSig;
  invoke MBGetTotalSizeFSBSig;
  invoke MBClearFSBSig;
  prove;
  instantiate free _0' == free _0', used _0' == used _0', total _0' == total _0',
  size _0! == size _0', report _0! == MMokay;
  invoke RMBGetTotalSize;
  invoke Success;
  prove;
  ■
D.6 Backing Stores

prove BackingStore$\text{domainCheck}$
prove by reduce;

prove BackingStore$\text{state}$
use BackingStoreIDElement;
prenex;
split backingStoreOverhead = 0;
cases:
instantiate children == {}, free == {}, used == {}, self == x, total == {};
prove by reduce;
instantiate a == 2, b == 1;
prove;
next:
instantiate children_0 == {}, free_0 == {}, used_0 == {},
self_0 == x, total_0 == 1 .. backingStoreOverhead;
use natCases(x := backingStoreOverhead);
invoke BackingStoreFSBSig;
invoke BackingStore;
invoke MemoryBlock;
prove;
prove by reduce;
next;

prove BackingStoreInit$\text{pre}$
use BackingStoreIDElement;
prove;
use splitContiguousMemory[A := addresses?, s := backingStoreOverhead];
prove;
instantiate free' == C, total' == addresses?, used' == {}, self' == x, children' == {};
invoke BackingStore;
invoke MemoryBlock;
prove;
apply subsetCup to predicate C ∈ P (B ∪ C);
apply inPowerSelf;
prove;
use ContiguousMemoryMaxType[x := B];
prove;
apply cardCupDisjoint;
simplify;
use ContiguousMemoryIsFiniteNum[x := B];
rearrange;
split B ∩ C = {};
simplify;

proof[RMBGetUsedSize\_vb\_fsb\_pre]
use MBGetUsedSize\_vb\_fsb\_pre;
invoke RMBGetUsedSizeFSBSig;
invoke MClearFSBSig;
prove;
instantiate free' == free', used_0' == used', total_0' == total',
size_0' == size!, report! == MMokay;
invoke RMBGetUsedSize;
invoke Success;
prove;

proof[MBGetUsedSize\_vb\_fsb\_pre]
use MBGetUsedSize\_vb\_fsb\_pre;
invoke MBGetUsedSizeFSBSig;
prove;
instantiate free' == free', used_0' == used', total_0' == total',
size_0' == size!, report! == MMokay;
invoke MBGetUsedSizeFSBSig;
invoke MBGetUsedSize;
invoke Success;
prove;
proof[BSAllocateChild\_vc\_fsb\_pre]
invoke BSAllocateChildFSBSig;
prenex;
 instantiate allocated\_0' == allocated!, free\_0' == free', total\_0' == total',
 used\_0' == used', childID\_0' == childID!, children' == children \cup \{childID\}, self' == self;
prove;
 invoke BSAllocateChild;
 invoke \Delta BackingStore;
 invoke BackingStore;
 invoke MAllocate;
 invoke \Delta MemoryBlock;
 invoke MemoryBlock;
 prove;
 use cardCap[Z][S := allocated! \cup used, T := free \setminus allocated!];
 split allocated! \cup used \in F Z \land free \setminus allocated! \in F Z; cases;
 rewrite;
 equality substitute;
 rewrite;
 apply cupPermutes to expression allocated! \cup (used \cup (free \setminus allocated!));
 use cupDiff[Z][A := allocated!, B := used];
 prove;
 apply cupSubsetLeft to expression allocated! \cup (free \cup used); prove;
 apply cupCupDisjoint to expression # (free \cup used);
 prove;
 next;
 prove by reduce;
 next;

proof[BSClear\_vc\_fsb\_pre]
 use MBClear\_vc\_fsb\_pre;
 invoke BSClearFSBSig;
 invoke MBClearFSBSig;
 invoke MBClear;
 invoke \Delta MemoryBlock;
 invoke MemoryBlock;
 prove;
 instantiate children' == \{}, self' == self, free\_0' == free',
 total\_0' == total', used\_0' == used';
 invoke BSClear;
 invoke \Delta BackingStore;
 invoke BackingStore;
 invoke MBClear;
 invoke \Delta MemoryBlock;
 invoke MemoryBlock;
 prove;
 apply cupCupDisjoint;
 rewrite;

proof[BSResizeChild vc fsb pre]
  use MBContentsResize vc fsb pre;
  pre;
  instantiate children' == children, newID! == oldID?, self' == self, free_0' == free',
  total_0' == total', used_0' == used', newAddresses_0' == newAddresses!;
  invoke BSResizeChildFSBSig;
  invoke MBContentsResizeFSBSig;
  invoke BSClearFSBSig;
  invoke MBClearFSBSig;
  invoke BSResizeChild;
  invoke MBAllocate;
  invoke ∆ BackingStore;
  invoke ∆ MemoryBlock;
  invoke BackingStore;
  invoke MemoryBlock;
  prove;
  apply cupElementRight;
  rewrite;
  split (used \ oldAddresses? \ newAddresses!) ∩ (free ∪ oldAddresses? \ newAddresses!) = {};
  cases;
  prove;
  apply plusCardDisjoint to expression
  (# (used \ oldAddresses? \ newAddresses!) + # (free ∪ oldAddresses? \ newAddresses!));
  rewrite;
  split (used \ oldAddresses? \ newAddresses! ∈ F Z ∧ free ∪ oldAddresses? \ newAddresses! ∈ F Z);
  cases;
  rewrite;
  apply cupPermutates to expression
  free ∪ oldAddresses? \ newAddresses! ∪ (used \ oldAddresses? \ newAddresses!);
  simplify;
  apply cupCommutes to expression
  free ∪ oldAddresses? \ newAddresses! ∪ newAddresses!;
  rewrite;
  split used \ oldAddresses? ∈ P Z;
  cases;
  rewrite;
  apply cupAssociates;
  rewrite;
  split free ∈ P Z ∧ oldAddresses? ∈ P Z;
  cases;
  rewrite;
  apply cupSubsetLeft to expression oldAddresses? ∪ used;
  rewrite;
  apply cupPermutates to expression free ∪ (newAddresses! ∪ used);
  simplify;
  apply cupSubsetLeft to expression newAddresses! ∪ (free ∪ used);
  rewrite;
  split newAddresses! ∈ P (free ∪ used);
  cases;
  prove;
  apply cardCupDisjoint to expression # (free ∪ used);
  prove by reduce;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next;
proof[BSAllocate \_vc\_fsb\_pre]
invoke BSAllocate\_FSBSig;
prove;
  use MBA\_Allocate \_vc\_fsb\_pre[\text{size} := \text{size} + \text{allocation\_Overhead}];
invoke MBA\_Allocate\_FSBSig;
invoke BSC\_Clear\_FSBSig;
invoke MBA\_Allocate;
invoke \Delta \_Memory\_Block;
prove;
  instantiate allocated\_! == allocated!, children' == children, free\_! == free',
  self' == self, total\_! == total', used\_! == used';
invoke BS\_Allocate;
invoke MBA\_Allocate;
invoke \Delta \_Backing\_Store;
invoke BSC\_Clear\_Store;
invoke MBA\_Allocate;
invoke Memory\_Block;
prove;
  apply plus\_Card\_Disjoint to expression \# (allocated! \cup used) + \# (free \setminus allocated!);
  rewrite;
  split allocated! \in \mathbb{F} \mathbb{Z};
cases;
  rewrite;
  apply cup\_Permutes to expression allocated! \cup (used \cup (free \setminus allocated!));
  simplify;
  split allocated! \in \mathbb{P} \mathbb{Z} \land free \setminus allocated! \in \mathbb{P} \mathbb{Z};
cases;
  simplify;
  apply cup\_Diff;
  split free \in \mathbb{P} \mathbb{Z};
cases;
  rewrite;
  apply cup\_Commutes to expression (free \cup used);
  simplify;
  apply cup\_Permutes to expression allocated! \cup (used \cup free);
  simplify;
  apply cup\_Subset\_Left to expression allocated! \cup free;
  rewrite;
  apply plus\_Card\_Disjoint;
  split free \in \mathbb{P} \mathbb{Z};
cases;
  prove;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next;
  prove by reduce;
  next
proof[BSResize\_domain\_Check]
prove by reduce;

proof[BSResize\_FSBSig\_domain\_Check]
prove by reduce;
proof[BSResize\_vc\_fsb\_pre]
use MBResize\_vc\_fsb\_pre;
invoke BSResizeFSBSig;
invoke MBResizeFSBSig;
invoke MBClearFSBSig;
invoke BackingStore;
invoke MBResize;
invoke \Delta MemoryBlock;
prove;

proof[BSGetTotalSize\_vc\_fsb\_pre]
use MBGetTotalSize\_vc\_fsb\_pre;
prove;
instantiate children' == children, self' == self, free_1' == free', oldAddresses_1' == oldAddresses!,
total_1' == total', used_1' == used';
invoke BSResize;
invoke MBResize;
invoke \Delta BackingStore;
invoke BackingStore;
invoke \Delta MemoryBlock;
invoke MemoryBlock;
prove;

proof[BSGetFreeSize\_vc\_fsb\_pre]
use MBGetFreeSize\_vc\_fsb\_pre;
prove;
instantiate children' == children, self' == self, free_0' == free', size_0! == size!,
total_0' == total', used_0' == used';
prove by reduce;

proof[BSGetUsedSize\_vc\_fsb\_pre]
use MBGetUsedSize\_vc\_fsb\_pre;
prove;
instantiate children' == children, self' == self, free_0' == free', size_0! == size!,
total_0' == total', used_0' == used';
prove by reduce;

proof[BSNotEmpty\_vc\_fsb\_pre]
instantiate total' == total, free' == free, used' == used, self' == self, children' == children,
report! == MMNotEmpty;
invoke BSNotEmptyFSBSig;
invoke BSNotEmpty;
invoke BackingStore;
invoke BackingStore;
invoke MemoryBlock;
prove;

proof[BSNonexistentAllocation\_vc\_fsb\_pre]
instantiate children' == children, self' == self, free' == free, used' == used, total' == total,
report! == MMNonexistentAllocation;
invoke BSNonexistentAllocationFSBSig;
invoke BSNonexistentAllocation;
invoke BackingStore;
invoke BackingStore;
invoke MemoryBlock;
prove by reduce;
proof[BSSizeTooSmall$\text{domainCheck}] 
prove by reduce;  

proof[BSSizeTooSmallFSBSig$\text{domainCheck}] 
prove by reduce;  

proof[BSSizeTooSmall$_{vc\_fsb\_pre}$] 
instantiate children' == children, self' == self, free' == free, used' == used, total' == total,  
report! == MMsizeTooSmall;  
invoke BSSizeTooSmallFSBSig;  
invoke BSSizeTooSmall;  
invoke $\Xi$ BackingStore;  
invoke MemoryBlock;  
prove;  

proof[BSNoFreeIdentifier$_{vc\_fsb\_pre}$] 
instantiate free' == free, used' == used, total' == total, self' == self, children' == children,  
report! == MMnoFreeIdentifier;  
invoke BSNoFreeIdentifierFSBSig;  
invoke BSNoFreeIdentifier;  
invoke $\Xi$ BackingStore;  
invoke BackingStore;  
invoke MemoryBlock;  
prove;  

proof[BSOutOfMemory$\text{domainCheck}] 
prove by reduce;  

proof[BSOutOfMemoryFSBSig$\text{domainCheck}] 
prove by reduce;  

proof[BSOutOfMemory$_{vc\_fsb\_pre}$] 
instantiate children' == children, free' == free, report! == MMoutOfMemory, self' == self,  
total' == total, used' == used;  
invoke BSOutOfMemory;  
invoke BSOutOfMemoryFSBSig;  
invoke BSClearFSBSig;  
invoke $\Xi$ BackingStore;  
prove by reduce;  

257
proof[RBSAllocateChild vc fsb pre]
prove;
invoke RBSAllocateChildFSBSig;
prove;
split size? ≤ # free;
cases;
split ∃ childID : BackingStoreID • childID! ∉ children ∧ childID! ≠ self;
cases;
prove;
use BSAllocateChild vc fsb pre;
use MBAllocate vc fsb pre;
invoke BSAllocateChildFSBSig;
invoke MBAllocateFSBSig;
invoke BSClearFSBSig;
prove;
instantiate allocated? == allocated!, free?′ == free′, total?′ == total′, used?′ == used′;
instantiate childID?′ == childID!
children?′ == children′, free?′ == free′, self?′ == self′;
total?′ == total′, used?′ == used′, report! == MMokay;
prove by reduce;
next;
prove;
use BSNoFreeIdentifier vc fsb pre;
invoke BSNoFreeIdentifierFSBSig;
invoke BSClearFSBSig;
prove;
instantiate childID?′ == childID!
prove;
instantiate childID?′ == self, children?′ == children′,
free?′ == free′, report?′! == MMnoFreeIdentifier, self?′ == self′,
total?′ == total′, used?′ == used′, allocated? == {);
prove by reduce;
next;
prove;
invoke MBOutOfMemory vc fsb pre;
invoke MBOutOfMemoryFSBSig;
invoke MBClearFSBSig;
invoke BackingStore;
prove;
instantiate free?′ == free′, report?′! == MMoutOfMemory, total?′ == total′,
used?′ == used′, self?′ == self, children?′ == children, childID?′ == self, allocated? == {);
prove by reduce;
next;

proof[RBSClear vc fsb pre]
use BSClear vc fsb pre;
prove;
instantiate children?′ == children′, free?′ == free′, self?′ == self′, total?′ == total′,
used?′ == used′, report! == MMokay;
prove by reduce;
■
proof[RBSResizeChild[vc, fsh, pre]]
invoke RBSResizeChildFSBSig;
split oldAddresses? ∈ used;
cases;
split newSize? ≤ # (free ∪ oldAddresses?)?
cases;
split free ∪ oldAddresses? ∈ ContiguousMemory;
cases;
split oldID? ∈ children;
cases;
use RSResizeChild[vc, fsh, pre];
prove;
instantiate children' − 0? = children', free' − 0? = free',
newAddresses' − 0? = newAddresses!, newID' − 0? = newID!,
self' − 0? = self', total' − 0? = total', used' − 0? = used',
report? = MMokay;
prove by reduce;
next;
use BSNonexistentAllocation[vc, fsh, pre];
instantiate children' − 0? = children, free' − 0? = free, newAddresses! = {}
newID! = oldID!, report! = MMnonexistentAllocation, self' − 0? = self,
total' − 0? = total, used' − 0? = used;
prove by reduce;
next;
use MBNotContiguousWithFree[vc, fsh, pre];
instantiate children' − 0? = children, free' − 0? = free, newAddresses! = {}
newID! = oldID!, report! = MBnotContiguousWithFree, self' − 0? = self,
total' − 0? = total, used' − 0? = used;
prove by reduce;
next;
use MBInsufficientFreeSize[vc, fsh, pre];
instantiate children' − 0? = children, free' − 0? = free, newAddresses! = {}
newID! = oldID!, report! = MBOutOfMemory, self' − 0? = self,
total' − 0? = total, used' − 0? = used;
prove by reduce;
next;
proof[RBSAllocate[vc, fsh, pre]]
split size? + allocationOverhead ≤ # free;
cases;
use BSAllocate[vc, fsh, pre];
prove;
instantiate allocated − 0? = allocated!, children' − 0? = children', free' − 0? = free',
report! = MMokay, self' − 0? = self', total' − 0? = total', used' − 0? = used';
use MBAlocate[vc, fsh, pre][size? := size? + allocationOverhead];
invoke BSAllocateFSBSig;
prove;
instantiate allocated' − 1? = allocated − 0?, free' − 1? = free − 0?,
used' − 1? = used − 0?, total' − 1? = total − 0?;
prove by reduce;
next;
use BSOOutOfMemory[vc, fsh, pre];
prove;
instantiate allocated! = {}, children' − 0? = children, free' − 0? = free,
report− 0? = MBOutOfMemory, self' − 0? = self, total' − 0? = total, used' − 0? = used;
prove by reduce;
next;
\begin{proof}\[\text{RBSResize}_{\text{vc}}_{\text{fsb}}_{\text{pre}}\]
    \begin{align*}
    &\text{split used} = \{\}; \\
    &\text{cases}; \\
    &\text{split children} = \{\}; \\
    &\text{cases}; \\
    &\text{split \# newAddresses?} \geq \text{backingStoreOverhead}; \\
    &\text{cases}; \\
    &\text{use } \text{BSSizeTooSmall}_{\text{vc}}_{\text{fsb}}_{\text{pre}}; \\
    &\text{prove}; \\
    &\text{instantiate children}_0' == \text{children }, \text{free}_0' == \text{free'}, \text{oldAddresses}_0! == \text{total}; \\
    &\text{report!} == \text{MMokay}, \text{self}_0' == \text{self}, \text{total}_0' == \text{total'}, \text{used}_0' == \text{used}; \\
    &\text{use } \text{MBResize}_{\text{vc}}_{\text{fsb}}_{\text{pre}}; \\
    &\text{prove}; \\
    &\text{invoke } \text{BSResizeFSBSig}; \\
    &\text{prove}; \\
    &\text{instantiate free}_1' == \text{free}_0', \text{oldAddresses}_1! == \text{oldAddresses}_0!; \\
    &\text{total}_1' == \text{total}_0', \text{used}_1' == \text{used}_0'; \\
    &\text{prove by reduce}; \\
    &\text{next}; \\
    &\text{use } \text{BSSizeTooSmall}_{\text{vc}}_{\text{fsb}}_{\text{pre}}[\text{addresses?} == \text{newAddresses}]; \\
    &\text{prove}; \\
    &\text{instantiate children}_0' == \text{children'}, \text{free}_0' == \text{free'}, \text{oldAddresses!} == \{\}; \\
    &\text{report}_0' == \text{MMNotEmpty}, \text{self}_0' == \text{self'}, \text{total}_0' == \text{total'}, \text{used}_0' == \text{used'}; \\
    &\text{prove by reduce}; \\
    &\text{next}; \\
    &\text{use } \text{MBNotEmpty}_{\text{vc}}_{\text{fsb}}_{\text{pre}}; \\
    &\text{prove}; \\
    &\text{instantiate children'} == \text{children}, \text{free}_0' == \text{free'}, \text{oldAddresses!} == \{\}; \\
    &\text{report}_0' == \text{MMNotEmpty}, \text{self}_0' == \text{self'}, \text{total}_0' == \text{total'}, \text{used}_0' == \text{used'}; \\
    &\text{prove by reduce}; \\
    &\text{next}; \\
\end{align*}
\end{proof}

\begin{proof}\[\text{RBSGetFreeSize}_{\text{vc}}_{\text{fsb}}_{\text{pre}}\]
    \begin{align*}
    &\text{use } \text{BSSizeTooSmall}_{\text{vc}}_{\text{fsb}}_{\text{pre}}; \\
    &\text{prove}; \\
    &\text{instantiate children}_0' == \text{children'}, \text{free}_0' == \text{free'}, \text{report!} == \text{MMokay}, \\
    &\text{self}_0' == \text{self'}, \text{size}_0! == \text{size!}, \text{total}_0' == \text{total'}, \text{used}_0' == \text{used'}; \\
    &\text{prove by reduce}; \\
\end{align*}
\end{proof}

\begin{proof}\[\text{RBSGetTotalSize}_{\text{vc}}_{\text{fsb}}_{\text{pre}}\]
    \begin{align*}
    &\text{use } \text{BSSizeTooSmall}_{\text{vc}}_{\text{fsb}}_{\text{pre}}; \\
    &\text{prove}; \\
    &\text{instantiate children}_0' == \text{children'}, \text{free}_0' == \text{free'}, \text{report!} == \text{MMokay}, \\
    &\text{self}_0' == \text{self'}, \text{size}_0! == \text{size!}, \text{total}_0' == \text{total'}, \text{used}_0' == \text{used'}; \\
    &\text{prove by reduce}; \\
\end{align*}
\end{proof}

\begin{proof}\[\text{RBSGetUsedSize}_{\text{vc}}_{\text{fsb}}_{\text{pre}}\]
    \begin{align*}
    &\text{use } \text{BSSizeTooSmall}_{\text{vc}}_{\text{fsb}}_{\text{pre}}; \\
    &\text{prove}; \\
    &\text{instantiate children}_0' == \text{children'}, \text{free}_0' == \text{free'}, \text{report!} == \text{MMokay}, \\
    &\text{self}_0' == \text{self'}, \text{size}_0! == \text{size!}, \text{total}_0' == \text{total'}, \text{used}_0' == \text{used'}; \\
    &\text{prove by reduce}; \\
\end{align*}
\end{proof}

\begin{proof}\[\text{GlobalStoresManager}^\$\text{domainCheck}\]
    \begin{align*}
    &\text{prove}; \\
\end{align*}
\end{proof}

\begin{proof}\[\text{GlobalMemoryManager}^\$\text{domainCheck}\]
    \begin{align*}
    &\text{prove}; \\
\end{align*}
\end{proof}
proof(GlobalMemoryManager VC fsb state)
  use BackingStore VC fsb state;
  prove;
  invoke BackingStoreFSBSig;
  invoke BackingStore;
  invoke MemoryBlock;
  instantiate rootBackingStore = self, childRelation = {}, threadACs = {},
  stores = (self -> θ BackingStore[children = {}, self = self]);
  prove;
  invoke GlobalMemoryManagerFSBSig;
  invoke GlobalMemoryManager;
  prove;
  invoke GlobalThreadACManager;
  prove;
  invoke GlobalStoresManager;
  prove;
  invoke BackingStore;
  invoke MemoryBlock;
  prove;
  instantiate childMemory = {};
  prove;
  split {childID : } • (childID, ((self, θ (BackingStore [children = {}])) childID) . total) = {};
  cases;
  prove;
  next;
  apply extensionality to predicate
  {childID : } • (childID, ((self, θ (BackingStore [children = {}])) childID) . total) = {};
  prove;
  next;

proof(GlobalMakeBS$domainCheck)
  prove;

proof(GlobalClearBS$domainCheck)
  prove;
Appendix E

Compilation Rules Proofs

Proof [Sequence introduction] We seek to apply Law C.129, taking the RHS of the rule as $Y$, and the LHS as $\mu X \cdot F(X)$, and proving $F(Y) \subseteq A$:

\[
\begin{align*}
\text{if } \text{frameStack} = \emptyset & \quad \longrightarrow \quad \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset & \quad \longrightarrow \\
\quad \text{\quad if } \cdots \quad \text{pc} = i & \quad \longrightarrow \quad A ; \quad \text{pc} := j \cdots \quad \text{pc} = j & \quad \longrightarrow \quad B \cdots \quad \text{Poll} ; \\
\mu X & \quad \longrightarrow \quad \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset & \quad \longrightarrow \\
\quad \text{\quad if } \cdots \quad \text{pc} = i & \quad \longrightarrow \quad A ; \quad \text{pc} := j \cdots \quad \text{pc} = j & \quad \longrightarrow \quad B \cdots \quad \text{Poll} ; \quad X
\end{align*}
\]

\[\subseteq_A\]

[Law[alt-seq-dist] (needs to be extended to handle divergent case)]

\[
\begin{align*}
\text{if } \text{frameStack} = \emptyset & \quad \longrightarrow \quad \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset & \quad \longrightarrow \\
\quad \text{\quad if } \cdots \quad \text{pc} = i & \quad \longrightarrow \quad A ; \quad \text{pc} := j \quad \text{Poll} ; \\
\mu X & \quad \longrightarrow \quad \text{Skip} \\
\text{if } \text{frameStack} \neq \emptyset & \quad \longrightarrow \\
\quad \text{\quad if } \cdots \quad \text{pc} = i & \quad \longrightarrow \quad A ; \quad \text{pc} := j \quad \text{Poll} ; \quad B \cdots \quad \text{pc} = j & \quad \longrightarrow \quad B \cdots \quad \text{Poll} ; \quad X
\end{align*}
\]

\[\subseteq_A\]

[Law[alt-assump-intro]]
\[ \text{Law[assump-alt-dist-move]} \]

\[ \subseteq_A \]

\[ \text{if frameStack} = \emptyset \rightarrow \{ \text{frameStack} = \emptyset \} \]

\[ \text{if } \text{frameStack} \neq \emptyset \rightarrow \]

\[ \begin{align*}
\lbrack 1 \rbrack & \text{ pc } = i \rightarrow \{ \text{frameStack} \neq \emptyset \}; A; pc := j; Poll; \\
\lbrack 2 \rbrack & \text{ if frameStack} = \emptyset \rightarrow \text{ Skip} \\
& \text{ if frameStack} \neq \emptyset \rightarrow \\
& \quad \text{ if } \ldots \text{ pc } = i \rightarrow A; pc := j; Poll; B; \ldots \text{ pc } = j \rightarrow B; \ldots; Poll; X \\
\end{align*} \]

\[ \ldots \]

\[ \text{ if } \ldots \text{ pc } = j \rightarrow \{ \text{frameStack} \neq \emptyset \}; B; Poll; \\
\lbrack 3 \rbrack & \text{ if frameStack} = \emptyset \rightarrow \text{ Skip} \\
& \text{ if frameStack} \neq \emptyset \rightarrow \\
& \quad \text{ if } \ldots \text{ pc } = i \rightarrow A; pc := j; Poll; B; \ldots \text{ pc } = j \rightarrow B; \ldots; Poll; X \\
\end{align*} \]

\[ \ldots \]

\[ \subseteq_A \]

\[ \text{Law[assump-assign-dist]} \]

\[ \text{if frameStack} = \emptyset \rightarrow \{ \text{frameStack} = \emptyset \} \]

\[ \text{if } \text{frameStack} \neq \emptyset \rightarrow \]

\[ \begin{align*}
\lbrack 1 \rbrack & \text{ pc } = i \rightarrow \{ \text{frameStack} \neq \emptyset \}; A; \{ \text{frameStack} \neq \emptyset \}; pc := j; Poll; \\
\lbrack 2 \rbrack & \text{ if frameStack} = \emptyset \rightarrow \text{ Skip} \\
& \text{ if frameStack} \neq \emptyset \rightarrow \\
& \quad \text{ if } \ldots \text{ pc } = i \rightarrow A; pc := j; Poll; B; \ldots \text{ pc } = j \rightarrow B; \ldots; Poll; X \\
\end{align*} \]

\[ \ldots \]

\[ \text{ if } \ldots \text{ pc } = j \rightarrow \{ \text{frameStack} \neq \emptyset \}; B; Poll; \\
\lbrack 3 \rbrack & \text{ if frameStack} = \emptyset \rightarrow \text{ Skip} \\
& \text{ if frameStack} \neq \emptyset \rightarrow \\
& \quad \text{ if } \ldots \text{ pc } = i \rightarrow A; pc := j; Poll; B; \ldots \text{ pc } = j \rightarrow B; \ldots; Poll; X \\
\end{align*} \]

\[ \ldots \]

\[ \subseteq_A \]

\[ \text{Poll doesn’t affect frameStack (needs proof)} \]
if frameStack = ∅ \rightarrow \{ frameStack = ∅ \}
if frameStack ≠ ∅ \rightarrow 
\[ \begin{align*}
& pc = i \rightarrow \{ frameStack ≠ ∅ \}; \ A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ { frameStack ≠ ∅ \}; \\
& if frameStack = ∅ \rightarrow Skip \\
& if \cdots \ pc = i \rightarrow A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ B \cdots \ pc = j \rightarrow B \cdots \ fi ; \ Poll \ ^\bullet \ X \\
& fi \\
& \cdots \\
& pc = j \rightarrow \{ frameStack ≠ ∅ \}; \ B \ ^\bullet \ Poll \ ^\bullet \ \\
& if frameStack = ∅ \rightarrow Skip \\
& if \cdots \ pc = i \rightarrow A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ B \cdots \ pc = j \rightarrow B \cdots \ fi ; \ Poll \ ^\bullet \ X \\
& fi \\
& \cdots \\
& fi
\end{align*} \]
[Law C.128]

\[\text{\texttt{A}}\]
if frameStack = ∅ \rightarrow \{ frameStack = ∅ \}
if frameStack ≠ ∅ \rightarrow 
\[ \begin{align*}
& pc = i \rightarrow \{ frameStack ≠ ∅ \}; \ A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ { frameStack ≠ ∅ \}; \\
& if frameStack = ∅ \rightarrow Skip \\
& if \cdots \ pc = i \rightarrow A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ B \cdots \ pc = j \rightarrow B \cdots \ fi ; \ Poll \ ^\bullet \ X \\
& fi \\
& \cdots \\
& pc = j \rightarrow \{ frameStack ≠ ∅ \}; \ B \ ^\bullet \ Poll \ ^\bullet \ \\
& if frameStack = ∅ \rightarrow Skip \\
& if \cdots \ pc = i \rightarrow A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ B \cdots \ pc = j \rightarrow B \cdots \ fi ; \ Poll \ ^\bullet \ X \\
& fi \\
& \cdots \\
& fi
\end{align*} \]
[collapse conditional with assumption]

\[\text{\texttt{A}}\]
if frameStack = ∅ \rightarrow \{ frameStack = ∅ \}
if frameStack ≠ ∅ \rightarrow 
\[ \begin{align*}
& pc = i \rightarrow \{ frameStack ≠ ∅ \}; \ A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ { frameStack ≠ ∅ \}; \\
& if \cdots \ pc = i \rightarrow A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ B \cdots \ pc = j \rightarrow B \cdots \ fi ; \ Poll \ ^\bullet \ X \\
& fi \\
& \cdots \\
& pc = j \rightarrow \{ frameStack ≠ ∅ \}; \ B \ ^\bullet \ Poll \ ^\bullet \ \\
& if frameStack = ∅ \rightarrow Skip \\
& if \cdots \ pc = i \rightarrow A \ ^\bullet \ pc := j \ ^\bullet \ ; \ Poll \ ^\bullet \ B \cdots \ pc = j \rightarrow B \cdots \ fi ; \ Poll \ ^\bullet \ X \\
& fi \\
& \cdots \\
& fi
\end{align*} \]
[Law assign-assump-intro]
if frameStack = ∅ → (frameStack = ∅)
  if frameStack ≠ ∅ →
    if pc = i → {frameStack ≠ ∅}; A; pc := j; \{pc = j\}; Poll; \{frameStack ≠ ∅\};
      \µX
        if frameStack = ∅ → Skip
          if frameStack ≠ ∅ →
            pc = j → (frameStack ≠ ∅); B; Poll;
            \µX
              if frameStack = ∅ → Skip
                if frameStack ≠ ∅ →
                  pc = i → A; pc := j; Poll; B ... pc = j → B ... fi; Poll; X
                fi
              fi
            fi
          fi
        fi
      fi
    fi
  fi

\equiv_A

{Poll doesn’t affect pc}

if frameStack = ∅ → (frameStack = ∅)
  if frameStack ≠ ∅ →
    if pc = i → {frameStack ≠ ∅}; A; pc := j; Poll; \{pc = j\}; \{frameStack ≠ ∅\};
      \µX
        if frameStack = ∅ → Skip
          if frameStack ≠ ∅ →
            pc = i → A; pc := j; Poll; B ... pc = j → B ... fi; Poll; X
          fi
        fi
      fi
    fi
  fi

\equiv_A

{swap disjoint assumptions}

if frameStack = ∅ → (frameStack = ∅)
  if frameStack ≠ ∅ →
    if pc = i → {frameStack ≠ ∅}; A; pc := j; Poll; \{frameStack ≠ ∅\}; \{pc = j\};
      \µX
        if frameStack = ∅ → Skip
          if frameStack ≠ ∅ →
            pc = i → A; pc := j; Poll; B ... pc = j → B ... fi; Poll; X
          fi
        fi
      fi
    fi
  fi

\equiv_A

{use assumption to collapse conditional}
So we have $F(Y) \subseteq_A Y$ and so, by Law C.129, $\mathcal{LX} \bullet F(X) \subseteq Y$. $\Box$