Verifying Memory Properties of Safety-Critical Java Programs

Chris Marriott

September 2011
Abstract

Java is a common and popular programming language with a wide range of tools designed specifically to aid development or verify specific properties. The introduction of SCJ has seen a novel programming paradigm that is designed specifically for make Java more applicable to safety-critical systems where certification is essential. Memory safety is an important property in any program, however, the failure of a safety-critical system could potentially have more serious consequences. It is not possible to apply the tools and techniques for Java programs to SCJ; therefore, this document discusses what is necessary to reason about memory safety in SCJ programs. We describe a starting point that contains an abstract language and a series of rules that must be obeyed in order to guarantee memory safety.
Contents

1 Introduction 1
  1.1 Motivation .................................................. 1
  1.2 Objectives .................................................. 2
  1.3 Outline ..................................................... 3

2 Safety-Critical Java and Verification Tools 4
  2.1 RTSJ .......................................................... 4
  2.2 SCJ .......................................................... 5
  2.3 Tools ......................................................... 8
     2.3.1 ESC/Java2 ............................................. 9
     2.3.2 Java Pathfinder ..................................... 11
     2.3.3 RJ .................................................... 14
     2.3.4 Perfect Developer .................................. 15
     2.3.5 SCJ Checker .......................................... 17
     2.3.6 Other tools ........................................... 20
  2.4 Final Considerations ...................................... 22

3 SCJ memory safety: a sound static checking technique 23
  3.1 Memory safety .............................................. 23
  3.2 SCJ-M ....................................................... 25
  3.3 Translation from SCJ to SCJ-M ............................ 28
  3.4 Memory-safety rules ....................................... 32
  3.5 Applied example .......................................... 41
  3.6 Final considerations ..................................... 41

4 Conclusion 43
  4.1 Further Work ............................................. 43
  4.1.1 Thesis goal ........................................... 43
  4.1.2 Above and beyond .................................... 44
List of Figures

2.1 Java method with possible null-pointer ................................................. 10
2.2 ESC/Java2 output for null-pointer example ........................................... 11
2.3 Java example for JPF to demonstrate a null-pointer exception ................... 12
2.4 Java example for JPF to demonstrate model checking of concurrent programs .... 13
2.5 RTSJ and SCJ memory representation in $\mathbb{R}_{SJ}$ ............................... 15
2.6 Simple Perfect specification for a list class ........................................... 17
2.7 Automatically generated Java implementation from Perfect specification ......... 18
2.8 Annotated SCJ example to demonstrate restrictions of SCJ Checker ............. 20

3.1 Memory checking process ................................................................. 25
3.2 BNF for SCJ-M .................................................................................. 29
3.3 Example SCJ program ......................................................................... 30
3.4 Example SCJ program expressed in SCJ-M ........................................... 31
3.5 Example application of memory-safety rules .......................................... 42

List of Tables

3.1 SCJ-M meta-variable table ................................................................. 32
3.2 SCJ-M environments table ................................................................. 33
3.3 SCJ-M auxiliary functions table .......................................................... 34
Chapter 1

Introduction

This Chapter introduces the work proposed in this document and tries to establish the motivation for its completion. It gives an overview of the technique and describes what is required in order to fulfil the objectives.

1.1 Motivation

Safety-critical systems are used, and relied upon, by everyone in today’s society. The expectation on systems to automatically keep us safe is forever growing; recent examples include the introduction of pedestrian detection and automatic braking systems in cars [1], and automatic lane-departure warning systems [2]. Systems such as these, whose failure may cause serious injury or even death, are subject to extensive verification and certification processes, especially in the automotive and avionics industries, to try and ensure failure is not an option.

The Java programming language, which we assume the reader is familiar with, is undoubtedly one of the most common and popular programming languages for program developers today. Java is a language with object orientation, which is considered the dominant programming paradigm [3]. The language provides both compile-time and run-time checking; virtual machines make the language architecture independent, whilst features such as Just-In-Time (JIT) compilation gives better performance for specific environments. Java’s ability to express concurrent implementations with threads also gives it appeal over languages such as C.

In the Java memory model, all objects are placed on the heap; local variables are stored on a method’s respective stack. When all references to a particular object are removed from the run-time environment, an automatic garbage collector removes the object from the heap on its next execution. The Java memory model is very much behind-the-scenes to the developer; it is not necessary to think about where and when memory is allocated or deallocated. In fact, the deallocation of memory by the garbage collector happens automatically at potentially random points. This lack of control over the garbage collector presents a potential problem when running time-critical applications. Consider, for example, the automatic braking system previously mentioned; it is not acceptable for the automatic brakes to wait whilst the garbage collector operates.

Verification and certification are timely and costly procedures; methods to automate or make easier these necessities are an interesting topic of research for both academia and industry. One of the more recent attempts to aid the design, verification, and certification of safety-critical systems is the introduction of Safety-Critical Java (SCJ) [4]. An international attempt, by various collaborators, has been made to produce a specification for Java that is suitable for safety-critical and real-time Java programs. It is no surprise that Java is being adapted for use in safety-critical systems as it is already a widely used and popular object-orientated language.

The Real-Time Specification for Java (RTSJ) [5] was designed to make Java more suitable for real-time systems, and provides both timing and behaviour predictability. The guarantees of reliability needed for safety-critical systems, however, were hard to achieve without a further restricted language. SCJ strikes a balance between the popular languages (such as Java and C), and languages already considered suitable for high-integrity systems (such as Ada).
Verifying Java programs is a problem that has been tackled by many, and as such, various publicly available and popular tools already exist. In the context of real-time and safety-critical systems, verification is important to ensure a system is both correct according to some specification that outlines behaviour, and does not contain errors. Unwanted behaviour such as unhandled exceptions or deadlock should not occur, for example.

As Java is not designed for high-integrity applications, and in the case of many applications the guarantees required are not as critical, many of the verification tools available are designed to be bug-hunting tools. These help developers improve the quality of code produced in a timely and cost-effective manner, as opposed to ensuring full program verification. The trade-off between productivity and full verification comes at a cost, namely the soundness and completeness of the tool. There are a few attempts to produce tools for full system verification, which often require extensive user annotations, is discussed in Section 2.3.

Our work is focused specifically on the SCJ language as it is a new and upcoming language that has already received interest from both industry and academia. More specifically, we focus on memory safety of SCJ programs: the SCJ memory model is one of the distinguishing features that sets the language apart from the RTSJ and standard Java languages.

In Java, the programmer does not have to worry about memory allocation or references to different scopes, for example, as it is the job of the garbage collector to maintain the memory automatically in the background. The RTSJ introduces the notion of scoped memory areas that are not subject to the garbage collector, however, the heap remains available for the programmer to create and reference objects with no additional concerns. The SCJ memory model goes one step further by completely removing access to the heap and limiting the entire program to scoped memory areas.

The strict memory model of SCJ introduces the possibility of scoped memory violations that must be checked. It is not enough, like in standard Java, to suggest that the lack of null-pointers and array-out-of-bounds exceptions give a memory safe program. The definition of memory safety in the context of SCJ must be enriched to include the scope rules defined in the language specification. Briefly, the scoped memory areas in SCJ form a hierarchy, and it is not valid to reference an object that is stored in a child memory area as the object may be removed from memory before the reference.

As the SCJ language is relatively new, verification tools and techniques are currently fairly sparse, however, the technique presented in [6] attempts to verify that a given SCJ program is valid according to the rules imposed by user annotations using a static checking tool. These annotations are used to define behavioural and memory properties for particular classes and methods; we focus on the memory properties. Currently, there is no evidence of an underlying formal technique to verify that their approach is both correct and sound; developing these formal foundations is one of our first goals.

The basic memory model of SCJ has been captured formally in the UTP in [7]. The memory model provides a basis to develop a formal representation of the necessary SCJ components required to verify that a given program is memory safe. A formalisation of the full language is not necessarily required in order to verify memory safety; abstractions can be made as will be discussed later in Chapter 3.

Work is also ongoing into the expression of a new variation of the formal language Circus; it is being designed specifically to capture the SCJ programming paradigm independently of the code [8]. Using this language, the authors intend on creating a refinement strategy from abstract models of SCJ programs that do not consider the programming paradigm (ie. missions and the memory model) to a more concrete representation that will facilitate automatic code generation. This work is complementary to that outlined here, and may potentially result in shared techniques applicable to both approaches.

1.2 Objectives

In order to check memory properties of SCJ programs, it is important to understand what memory safety is and what memory rules are imposed by the SCJ specification. Part of this work aims at analysing existing tools and techniques for variants of the Java programming language to try and determine what properties are being checked, and what techniques are used to check them.

The technique we aim to use should be sound and facilitate static checking. A sound checking technique must be built on an underlying formal theory that is proved to be correct. Our goal, therefore, is to develop a formal basis on which SCJ programs can be represented, and a series of formal rules that,
when true, guarantee memory safety. Soundness is an essential part of our work, as existing techniques rarely justify their approach is sound, as will be shown in Chapter 2.

Our technique will use an abstract language to represent SCJ programs; by using an abstraction technique, we can focus on parts of SCJ program required to verify memory safety and present this is a consistent and structured format. We will use the Unified Theories of Programming (UTP) [9] to give a formal semantics to our abstract language. By using the UTP, we can use existing theories about object orientation and the memory model of SCJ, for example, to give a meaning to our new language.

It is not unreasonable to suggest that a sound static checking technique for Level 1 SCJ programs can be developed. This document describes a representation of the SCJ programming language, and the basic rules required to express memory safety. The representation and corresponding rules are a work-in-progress and will be continually enriched to facilitate a greater set of SCJ programs. Formal proofs of the memory safety rules are an essential part of this work; however, they are not yet complete. The ability to take any SCJ program as input to our technique is another distinguishing factor as other techniques often rely on limited programming constructs or idealised programming styles. It is also our intention to automate the translation from SCJ to our new abstract language.

1.3 Outline

Chapter 2 gives an introduction to RTSJ and SCJ and describes the differences between the languages. It justifies the need for real-time and safety-critical variations of Java, and how these are more suitable for use in their setting. A detailed comparison of the memory models associated with each language is included. This section also looks at the existing verification tools for Java programs and evaluates their applicability to SCJ programs. A wide range of tools and techniques with different model checkers and theorem provers are considered.

Chapter 3 describes ways in which memory properties are verified in existing techniques. It also describes an abstract language for SCJ and a series of formal rules designed to verify the memory safety of a given SCJ program. Examples of how the abstract language is used and the application of memory safety rules are demonstrated.

Finally, Chapter 4 draws the document to a close and describes in more detail the future work to be completed as part of the thesis. It also describes longer term possibilities that may be investigated, however, these are not considered goals of the thesis.
Chapter 2

Safety-Critical Java and Verification Tools

This Chapter gives an introduction to the RTSJ and SCJ programming languages, which are variation of the Java programming language specifically designed for real-time and safety-critical applications, and the differences between them. Section 2.3 describes existing verification tools for Java, and how these can be applied to SCJ programs.

2.1 RTSJ

The RTSJ is a variation of the popular Java language that was designed to address the limitations found in standard Java when developing real-time programs. The idea was to create a language that imposes as few limitations on the developer as possible, whilst also giving them the functionality required to express real-time properties. The main additions to standard Java, found in the RTSJ, are discussed individually below [10].

**Time** The standard concept of calendar time provided by Java is not enough for time-critical systems. RTSJ introduces high resolution time, which has granularity of a nanosecond. This concept of time is then extended into three categories: relative, absolute, and rational time. Relative time is a simple duration from one point in time to another. Absolute time defines an exact fixed point in time. Rational time inherits from relative time, however, it also has a frequency to capture the rate of occurrence for periodic events, for example.

**Scheduling** In standard Java, the user has no guarantees about scheduling in the JVM; this is not acceptable for systems with priority-based schedulables. The RTSJ facilitates the implementation of user-specific scheduling algorithms, for example, earliest deadline first; however, priority-based scheduling is the most common. All schedulable objects in RTSJ have three parameters: a release requirement, a memory requirement, and a scheduling requirement. The release requirement defines when the schedulable object is ready to run. The memory requirement defines the rate at which the object allocates memory. Finally, the scheduling requirement defines the priority of the object.

Schedulable objects can be periodic, aperiodic, or sporadic events. Periodic events have a fixed arrival frequency whereas aperiodic and sporadic events occur as and when triggered by the system. The difference between aperiodic and sporadic is the minimum inter-arrival time found in sporadic events, which specifies the minimum time that must pass before the object can be released again.

**Memory management** As mentioned previously, it is not desirable for real-time applications to be interrupted by the garbage collector. To overcome this problem, the RTSJ introduces immortal and scoped memory areas, which are not subject to garbage collection. Memory areas, like the heap, store dynamically created objects. Immortal memory exists for the entire length of the program, and objects stored there cannot be removed once stored. In contrast, scoped memory areas have a lifetime and objects...
stored there can be added and removed. Schedulable objects can use scoped memory areas, however, use of either immortal or scoped memory areas must be explicit, as the heap is used by default. Once all schedulable objects have finished inside a particular scoped memory area, it is cleared out.

Scoped memory areas can be created as and when required within a program. The current allocation context of a program can also change during execution, therefore, the scope structure of an RTSJ program is a tree-like structure of memory areas. This can potentially cause reference problems at run-time; the following rules are defined to ensure no dangling references occur:

- Objects in the heap cannot reference scoped memory areas.
- Objects in immortal memory cannot reference scoped memory areas.
- Objects in scoped memory areas can only reference other scoped memory areas if the target area is down the scope tree, that is, towards the root.
- Scoped memory areas must have only one parent.

The introduction of scoped memory areas has avoided the concerns raised with garbage collection; however, work is currently ongoing into the use of garbage collection algorithms with real-time systems [10].

**Threads** Java contains threads; the RTSJ introduces real-time threads which inherit the same requirements of a schedulable object. Real-time no-heap threads are also real-time threads, but they guarantee not to reference or allocate any objects on the heap; this makes them independent of the garbage collector.

**Asynchronous events** Threads are often used to perform tasks that are not waiting for some specific event; for this we use asynchronous event handlers. It is true that a thread could wait for an event to occur, however, when scaled up for all events in a system, it could lead to a massive number of threads running concurrently.

It is also important to consider that a schedulable object may need to be interrupted when a particular asynchronous event has occurred. The RTSJ includes the notion of Asynchronous Transfer of Control (ATC), which allows a schedulable object to be interrupted, as long as a flag allowing it to be interrupted has been set. This transfers the program execution from the current schedulable object to the new event handler that has just been released. Once a transfer has been made, it is not possible to go back to the original execution point. Synchronised methods can never be interrupted.

### 2.2 SCJ

The Safety-Critical Java specification [4] is based on the Java reference language and the RTSJ. It is designed to be more suited to safety-critical systems, and in particular their certifiability. It is true that Java and the RTSJ could be used for such systems, however, restrictions are often imposed to ensure that programs are suitable for certification. An important example of this is garbage collection, which is not considered suitable for use in real-time and safety-critical systems. The memory model of SCJ is one of its distinguishing features from RTSJ, and is discussed in more detail below.

SCJ programs that conform to the SCJ specification and use safety-critical libraries are certifiable to the Level A of the DO-178B [11] avionics standard. Level A systems are defined as those whose failure could cause catastrophic failure and subsequently prevent an aircraft from continuing safely. It is not true, however, that SCJ programs written to the specification will automatically be certifiable by a standard. The SCJ specification is simply designed to make certification easier by providing a programming paradigm.

**SCJ Paradigm** The SCJ programming paradigm is focused around the concept of missions, where each mission has a number of event handlers. When released, a mission goes through three phases: *initialisation*, *execution*, and *cleanup*. Objects in each mission are created during the initialisation phase before being used in the execution phase; new objects are not allocated once the mission has started executing. Finally, once the mission has finished executing all its event handlers, the cleanup phase is entered to perform any final tasks before the mission finishes.
Missions are controlled by the mission sequencer class; missions are released for execution based on the user implementation of its `getNextMission()` method. The safelet class is the top level of any SCJ program and defines the outer-most mission sequencer. The safelet, like the mission, also has initial and final phases of execution called `setUp` and `tearDown` respectively.

**Compliance levels** SCJ implementations also have a compliance-level, which is used to define the complexity of a program. For example, hard real-time applications are often likely to contain a single thread of execution with simple timing properties to ensure deadlines are not missed. Alternatively, more complex programs may be highly concurrent with multiple threads executing at the same time. SCJ has three compliance-levels: 0, 1 and 2. Level 0 programs refer to the most simple programs described above, whilst Level 2 programs have increased complexity, as also illustrated.

Level 0 programs are cyclic executive programs. Missions contain only periodic event handlers which have fixed periods, priorities, and release times (in relation to a cycle). There is no concurrency at this level. Only Sequential missions are allowed at this level.

Level 1 programs introduce aperiodic event handlers. These, along with periodic event handlers, are executed concurrently in each mission. Schedulable objects are controlled by a fixed-priority pre-emptive scheduler.

Level 2 programs are the most complex and introduce real-time no-heap threads. They also allow concurrent missions and nesting of missions. Methods that may cause blocking, such as `Object.wait` and `Object.notify` are also allowed at this level.

As Level 2 programs are so complex, existing research on SCJ has focused on Level 1 applications, which is considered similar in complexity to Ravenscar Ada [7,12], for instance.

**Annotations** The SCJ specification includes specific annotations to express constraints on classes and methods. These annotations allow static analysis to be performed to ensure an implementation conforms to the specification rules. They are also maintained in compiled bytecode to allow checks at class load-time. The annotations are split into three categories: compliance-level, memory safety, and behavioural restrictions.

Compliance-level annotations are used to ensure classes and methods are only used at the correct level; for example, a Level 1 implementation could not use a method defined as Level 2 compliant. Memory safety annotations are used to ensure no dangling references exist. Finally, behavioural annotations are used to restrict properties such as blocking and allocation. The SCJ annotations, and rules to accompany them, are discussed in more detail in Section 2.3.5.

**Memory Model** The SCJ specification takes the memory restrictions one step further than the RTSJ by completely removing the heap and garbage collection from the memory model. Two types of memory area are defined within the memory model, these are immortal and scoped memory areas. Immortal memory is the same as that in the RTSJ. Scoped memory areas are used for individual aspects of the SCJ programming model, for example missions have their own scoped memory, as do event handlers; these are called mission memory and per-release memory respectively. Temporary private scoped memory areas are also used during the initialisation phase of a mission, and by individual handlers; these are organised in a stack-based structure. Once a particular handler release or mission finishes executing, its associated memory area(s) and contents are removed. In practice, this creates a hierarchical structure of memory areas from the immortal memory down to the private scoped memory areas for event handlers. To avoid the possibility of dangling references, the SCJ memory structure has strict rules about references within the hierarchical memory areas, as will be discussed below.

As the programmer has more control of the memory management, there is need for tools and techniques to check the memory safety for programs. Currently only one tool exists to check memory safety for SCJ programs; it is discussed in Section 2.3.5. This technique uses the inbuilt SCJ annotations to define memory properties of a particular program: these annotations are then checked statically to ensure the SCJ memory rules are adhered to. This technique does not, however, rely on or is justified by any formal representation of the underlying memory model, which is what Cavalcanti et al. introduce in [7].

**SCJ Memory Model in the UTP** Cavalcanti et al. present a formalisation of the SCJ memory model using the UTP. By using the UTP, it allows the memory model to be integrated with existing
theories about object-orientation and time, both of which are relevant to SCJ. In the UTP, relations are
made up of a series of predicates and are used to define the behaviour of a program. These predicates
define healthiness conditions which must be obeyed by all programs.

The SCJ memory structure contains individual stacks for the program, mission sequencer, and event
handlers. In the memory model presented, stacks contain frames, which separate the individual execution
contexts for a particular method. Each frame contains a set of variables and associated values; these values
are defined to be either a primitive value, the special value null, or a reference. Values that hold references
point to objects stored in memory areas; these objects contain a set of fields and respective values. Once
again, these values may be primitive, or contain further references to other objects.

From the alphabet that describes the memory model, a series of healthiness conditions are created to
ensure a given program satisfies the SCJ memory rules. Below is a summary of the healthiness conditions.

| HSCJ1 | The immortal memory can only grow, that is, elements can be added but not removed. |
| HSCJ2 | References contained in the program stack may only point to objects in the immortal memory area. |
| HSCJ3 | References contained in the immortal memory area may only point to objects in the immortal memory area. |
| HSCJ4 | References contained in the mission sequencer stack may only point to objects in the immortal memory area. |
| HSCJ5 | References contained in the mission memory area may only point to objects in the immortal memory area. |
| HSCJ6 | References contained in a handler’s stack may only point to objects in one of its own temporary private memory areas, its per-release memory area, the corresponding mission memory area, or the immortal memory. |
| HSCJ7 | References contained in a handler’s per-release memory area may only point to objects in the same per-release memory area, the corresponding mission memory, or immortal memory. |
| HSCJ8 | References contained in one of a handler’s private temporary memory areas may only point to objects in the same private temporary memory area, the handlers per-release memory area, the corresponding mission memory, or immortal memory. |
| HSCJ9 | The number of frames on an individual handler’s stack must be strictly greater than the number of temporary private memory areas associated with the handler. |
| HSCJ10 | All memory areas are disjoint, that is, no object can reside in more than one memory area. |

As the values assigned to variables can contain references to objects in other memory areas, it is necessary
to define additional healthiness conditions to ensure that each variable has a valid path to a value in a
specific memory area. These are described below:

| HV1 | Values of variables in the program stack have a valid path from the program stack to immortal memory. |
| HV2 | Values of variables in the mission sequencer stack have a valid path from the mission sequencer stack to mission memory or immortal memory. |
| HV3 | Values of variables in a specific handler’s stack have a valid path from the handler stack to any corresponding temporary private memory, the associated per-release memory, mission memory, or the immortal memory. |

As defined previously, it is only possible to add items to the immortal memory; mission memory areas
are similar as they too can only increase in size until the mission has finished executing, at which point
the entire memory area is removed. However, it is not valid to say that the previous state of mission
memory is a subset of the current state. This is because the current state of the mission memory may be
a new instance of the same mission, whilst the previous state defines the old mission. This same property
applies to the per-release memory and temporary private memory areas for handlers. To formally capture
this behaviour, a history of the previous memory structure is kept in a number of sequences for the
mission, per-release, and temporary private memory areas. The following healthiness conditions capture
properties about memory history:

**HH1** If there is no current mission executing, the mission memory must be empty and there can be no event handlers active.

**HH2** The sequence of mission memory history can only be extended with a valid mission.

**HH3** Whilst a mission is executing, the mission memory area can only grow in size. The set of handlers remains the same, and the release history of the handlers can only be extended with a valid release.

**HH4** All handlers that are currently not released must have an empty per-release memory area and have created no temporary private memory areas.

**HH5** All temporary private memory areas created by a handler must be defined with a unique identifier in the temporary private memory area history sequence, even if a previous memory area identifier is not in use and has been removed from the stack.

**HH6** The number of elements in the last stack of the sequence that defines the history of temporary private memory areas for a particular handler must be equal to the number of temporary private memory areas created.

**HH7** Whilst a particular handler is executing, the associated per-release memory area can only be increased. The history of temporary private memory areas created can only be extended. Also, the contents of all temporary private memory areas for the handler can only increase.

As far as the authors are aware, this paper currently gives the only formal definition of the SCJ memory model.

The term memory safety is often used in verification to refer to the absence of null pointers in a program. This means there can be no variable in a stack that points to an uninitialised object, for example. This is not sufficient in SCJ as memory safety also needs to consider the structure of the memory model. As well as having no null pointers, it is important to check that a variable in a stack does not point to an object in a memory area that is considered a child in the memory structure. For example, a variable in the program stack cannot point to an object in the mission memory area; this is because the program has a life longer than the mission memory area and, therefore, an illegal reference might be created when the mission area is reclaimed. To capture memory safety formally, three more healthiness conditions are required:

**HMS1** There are no dangling references from the program stack to the immortal memory area.

**HMS2** There are no dangling references from the mission sequence stack to the immortal or mission memory areas.

**HMS3** There are no dangling references from the handler stacks to the immortal, mission, per-release, or temporary private memory areas.

These healthiness conditions give a foundation to allow formal reasoning about memory safety of SCJ programs. A program that conforms to these conditions is guaranteed to be memory safe. Sound approaches to verifying memory properties can be investigated using these conditions.

### 2.3 Tools

The analysis of individual tools below described what the tools are designed to achieve and how this is made possible. The tools discussed are a selection of those more commonly used by developers when verification tools are required, and those specifically designed for RTSJ and SCJ; less common tools are discussed at the end of this section. The analysis is focused heavily on memory properties, and how each tool addresses the issue of null-pointer exceptions. The applicability of these tools to SCJ programs is also evaluated, however, it is not expected that a tool specifically designed for Java will be able to check similar properties for SCJ.
2.3.1 ESC/Java2

The Extended Static Checking tool for Java (ESC/Java2) [13] is designed to find errors in JML annotated Java programs, and is arguably the most common verification tool used by Java developers. The JML annotation language is a behavioural interface specification language for Java [14]. The language is used to express additional information about the interface and behaviour of Java programs using annotations. Interface properties describe the names and static information about Java declarations; behavioural properties describe how a declaration acts when called. The behavioural annotations are used to define pre and post conditions, and invariants.

JML annotations are ignored by regular Java compilers, as they are prefixed with the Java comment string. ESC/Java2 is presented as a half-way house between static checking, such as type checking, and full program verification using formal proofs. This is achieved by adding a series of intuitive annotations to the original program in a language as close to Java as possible; a subset of JML.

The aim of ESC/Java2 is to provide a cost-effective tool that is capable of catching more errors than a standard static checker without the overheads of formal verification. The authors explain that the need to provide annotations in code is independent of the checker being used because manual checking requires similar annotations [13]. This is a justification by the authors on the annotation burden, which is inevitable for annotation-based checkers.

In ESC/Java2, guarded commands that express pre conditions under which each routine body is constrained are generated automatically from an abstract syntax tree representing the program. The strongest post conditions and weakest pre conditions are taken from the guarded commands, and are verified using the Simplify theorem prover [15].

Two ideal properties of a static checker are the ability to find all errors in the original program, i.e. ensure it is complete, and to ensure every error found is in fact a real error, i.e. soundness. The tool is neither sound nor complete and is targeted at finding as many errors as possible in a reasonable amount of time with minimal effort placed on the end user. Flanagan et al. acknowledge that soundness is a desirable property, however, their approach is more focused on the cost-effectiveness of using the tool, which is achieved through compromises in soundness, completeness, annotation overheads and performance. For this reason alone, the tool is not suited to full system verification, however, ESC/Java2 is able to check specific properties about Java programs that may be applicable to SCJ. Specifically, it may be possible to catch memory violations such as null dereferences and index out of bounds.

ESC/Java2 uses a modular approach to verification within a guess and verify technique. This means the tool checks an individual input file rather than an entire system. This gives it the advantage of being able to check smaller parts of the system without having to use time and resources to check the whole system. A disadvantage with this approach comes from the additional class definitions required to check the current class. This increases the overhead required to check files, and does not scale well to large systems.

When a class calls a method in another class, ESC/Java2 relies on specification of the method in the static class definition for verification rather than an instance of the class, in which the method may be overridden with stricter conditions. This is based on the theory of behavioural subtyping where a proven property in a supertype holds in a subtype [16].

The underlying theorem prover Simplify automatically checks a series of verification conditions written in first-order predicate calculus. These are automatically generated from the JML annotations provided by the user in the Java implementation. Annotations are required to define more specific checks that cannot be automatically generated. For example, it is not possible to establish automatically specific conditions such as the range for a variable before and after a method call. Where assertions to ensure variables do not point to null do not exist, ESC/Java2 automatically warns the user about the possibility of a null-pointer at every dereference, whether the possibility is valid or not.

The logic used by the Simplify theorem prover is untyped; therefore the individual variables in Java are defined to be of type type. A set of acceptable values for the type are then defined to ensure the theorem prover understands the restrictions. This is important because Java is a typed language, and removing the definition of types also removes the assurance gained through such a language. Without a type-safe language, it is possible that memory safety could be compromised; for example, an untyped language may allow an arbitrary integer as a pointer.

Part of the incompleteness found in the ESC/Java2 tool comes from the limitations in the theorem
class C {

    int n;

    static int m(C a, C b) {
        if (a != null) {
            return a.n;
        } else {
            return b.n;
        }
    }
}

Figure 2.1: Java method with possible null-pointer

prover. For example, there is no semantics for multiplication and it cannot handle mathematical induction. Also, Simplify is incomplete regarding the possibility of integer overflow; this is a common problem and ESC/Java2 uses a technique to try and reduce the likelihood of overflow occurring. This is achieved by representing any value greater than one million as a symbolic value whose ordering is known, but the exact value is not. Also, the semantics for floating-point operations are very weak; the theorem prover cannot prove that two different integers are not equal, for example. Similarly, whilst the prover can confirm that a valid string is not null, it cannot verify the value of a specific character.

Loops are modelled with an approximation of their semantics by unrolling the loop a given number of times; the remaining iterations are modelled with code that terminates without error. This approach does not find errors that only occur in later stages of a loop as the default behaviour is to unroll loops 1.5 times. In a large scale example, unrolling loops twice produced a 20% time increase, unrolling 5 times gave a 100% increase. Flanagan et al. mention that sufficiently strong loop invariants are difficult to provide, even by experts.

As mentioned above, the theorem prover is also not sound; this is shown further through some of the errors raised during the checking process. The theorem prover produces an error whenever a potential counterexample is found; it will not continue to evaluate the verification condition to determine if the counterexample is false, or whether the entire verification condition can be evaluated to true. Similarly, the checking technique throws an error for every run-time exception whether this exception is handled in the code or not.

Evaluation The tool is readily available on all major platforms and is easy to use. It comes complete with comprehensive documentation on how the tool is implemented, and how the pre and post conditions are represented in the underlying theorem prover, Simplify. The tool is invoked from the command line through a script, which can be customised based on the individual user’s requirements; this method of separate configuration is well suited to safety-critical systems where traceability is important. The latest version of the tool is compatible with Java 1.3 and 1.4; the latest versions 1.5 and 1.6 are not supported.

In its original state, the ESC/Java2 tool is not capable of verifying SCJ implementations; this is due to the absence of the necessary javax.realtime and javax.safetycritical libraries. With the addition of these libraries, ESC/Java2 can be used in the same way for SCJ programs as Java programs to highlight possible null-pointer exceptions, for example. It does not, however, have knowledge of the SCJ memory model and therefore cannot check for the lack of dangling references between scope memory areas. With no comprehensive library implementation currently available, a small example in Figure 2.1 shows a Java method that could potentially contain a null-pointer. The reference a.n is protected by the if statement, however, the reference b.n has no guard and could point to a field in the class C that has not been initialised.

Figure 2.2 shows the ESC/Java2 output when the example is checked. As shown, the tool returns the source of the error in the source code with a description of the error; the tool provides useful feedback to the user as its primary focus is to be an effective debugging tool.

To overcome these errors, additional JML annotations are required in the source code to specify
properties about the program. In the example shown, one solution would be to add another conditional statement that checks if the value of \( b \) is null; another would be to add a JML annotation that states that the method \( m \) cannot be called unless either \( a \) or \( b \) are not null:

\[
//@ requires a != null | b != null;
\]

The authors estimate that for every thousand lines of code, a further 40-100 annotations are required. They also estimate that a programmer could annotate at a rate of 300-600 lines of original source code every hour.

ESC/Java2 is a good debugging tool for Java programs that does not require large amounts of additional knowledge from the end-user. The main effort required to make the tool effective is the addition of JML annotations to the implementation to check specific properties. The tool is capable of checking SCJ programs given the library definitions and can prove useful in the debugging and verification process of behavioural properties.

### 2.3.2 Java Pathfinder

Java Pathfinder (JPF) is a model-checking tool for Java programs designed to analyse automatically a multi-threaded Java program using the SPIN model checker [17]. This is achieved by translating a Java program, with a finite state space, into Promela, the modelling language used by SPIN. In conjunction with the model produced, JPF uses a custom JVM that executes concurrent Java programs in every possible way, ensuring that every possible execution path is explored from a particular decision or instance of nondeterminism point (called choice points). This is achieved using a state graph to represent the choice points in the program execution.

Choice points occur when an input value is recorded or a particular thread is chosen for execution. JPF records information about the current state it is in, and checks the previously visited states to ensure paths are not explored multiple times. The state graph is then used for backtracking to ensure every execution path from a particular choice point has been explored; considering all possible execution paths allows the model to be exhaustively checked. Programs without concurrency, and a predefined set of...
public class A {
  static String s1;
  static String s2;
  public static void x(String s) {
    assert(s != null);
    if ( s.equals("string") ) {}
  }
  public static void main(String[] args) {
    x(s1);
    x(s2);
  }
}

Figure 2.3: Java example for JPF to demonstrate a null-pointer exception

input values, have a single path of execution; in this case, JPF operates only in a similar way to standard testing. Symbolic model checking, however, allows JPF to check all possible input values as opposed to a fixed one; this removes the need to produce tests to establish code coverage of all valid inputs.

With concurrency, the state space of a given program increases rapidly; JPF therefore suffers from the state explosion problem. To tackle that, JPF uses an on-the-fly partial-order reduction technique, which combines operations that do not require communication between different threads into a single state. This is because it is the communications and interactions between threads in concurrent programs that represent choice points in the execution path.

Specific properties and behavioural constraints defined by the programmer are expressed as assertions in Promela or formulae in linear temporal logic. User-defined assertions in the Java program are translated into Promela assertions. The SPIN model checker automatically analyses these assertions and formulae to ensure the properties hold. If an error is found, that is, one of the assertions is violated, the execution trace is recorded and presented to the user. SPIN is able to find deadlocks, something particularly important when handling concurrent programs.

One of the challenges faced when translating between different languages is the problem of providing meaningful and useful error messages. In order to maintain links between the Java code and the corresponding Promela code, a series of print statements are used in the Java code and then translated into Promela. This gives a mapping between the two languages and allows errors found in Promela to be expressed in the corresponding Java code.

Evaluation

The tool is publicly available to download for all major platforms, and it is easy to use. JPF in its standard form is able to check for deadlocks and unhandled exceptions. The latter can be detected because JPF uses an execution-based approach, which halts when exceptions are raised, just like testing. Users have the option of extending JPF with listeners to check for more specific properties.

Potential null pointers in a program produce null-pointer exceptions at run-time, and in this case, JPF detects these. JPF does not detect exceptions that are caught in the program code. This is because there is no problem with the execution as such; even though a null-pointer exception has been thrown, the code has handled this and continued executing.

Although JPF is capable of detecting null-pointer exceptions, it is not complete; once an error has been detected, checking stops.

Consider the example shown in Figure 2.3, where two string variables s1 and s2 are declared, but not initialised. When a string is passed as a parameter to the method x, the assertion on line 7 ensures the string is not null in an attempt to prevent a null-pointer exception being raised at the comparison on line 8. When executed with the assertions enabled option (java -ea), this returns an assertion error as expected. Without assertions enabled, this returns a null-pointer exception.

When passed through JPF, the assertion error is detected and an error is passed to the user; similarly
public class B {
  
  public static void main(String[] args) {
    C a = new C();
    C b = new C();
    a.start();
    b.start();
  }
}

class C extends Thread {
  static String s;
  static String s2;
  
  public void run() {
    assert(s != null);
    if (s.equals("string")) {} 
  
    assert(s2 != null);
    if (s2.equals("string")) {} 
  }
}

Figure 2.4: Java example for JPF to demonstrate model checking of concurrent programs

when the assertion on line 7 is removed, JPF successfully detects the unhandled null-pointer exception. As JPF relies on the execution of Java programs, it does not proceed with the execution once an error has been detected. This is because the behaviour of a program is not known once an exception has been raised. This style of checking means that errors further along the execution path are never discovered. We consider again the example above; the second call to method x at line 13 encounters exactly the same problems as the first method call, however, the execution of the program never reaches this point, and so the error remains undetected.

To demonstrate JPF's ability to find errors in concurrent programs, the example in Figure 2.4 uses a thread class C to define a sequence of two assertions and string comparisons. The main method creates two threads to operate concurrently.

When executed with assertions enabled, two assertion errors (from line 16) are returned to the user as expected: one from each thread. Similarly, when checked using JPF, the two failed assertion errors are discovered. Using an option in JPF, it is possible to discover all possible ways of reaching a particular error, as opposed to the first execution path found. Using this option, JPF gives six possible execution traces that lead to the failed assertion error. These six traces show the different decisions made at the various choice points in the state graph; as the input values are fixed, these choice points represent the possible schedulings of the two threads in the program as they are swapped in and out of execution. A detailed account of the exact differences between the error traces can be produced by analysing the bytecode instructions performed in each trace. As demonstrated in the previous example, the assertion error that occurs at line 19 is never reached in either thread because execution is halted at the first error on line 16.

The limitation shown in the previous examples is common in model checking techniques that rely on a state graph to capture a program's behaviour. If the behaviour past a specific state is undefined due to an error, it is not possible to continue checking the model.

Model checking relies on explicit values in a program to check a specific execution path. Symbolic execution eliminates this problem, however, the inherent state explosion that is associated with JPF is a limiting factor for verification when compared to other theorem-proving techniques.

As JPF is an execution-based model checker for Java, it is not immediately applicable to real-time variants such as RTSJ and SCJ as they are different languages and not just subsets of standard Java.
Both the RTSJ and SCJ have custom JVMs that cater for the additional features found in real-time programming languages; one of the most obvious is time. It is true in fact that model checkers for standard Java do not need the concept of time as threads can be scheduled at any time.

2.3.3 R₅J

The R₅J tool is an extension to JPF to support real-time Java programs, specifically RTSJ and SCJ programs [18]. To overcome the limitations of JPF described above, and make JPF applicable to real-time programs, a scheduling algorithm that implements fixed-priority preemptive scheduling without time-slicing is at the heart of R₅J. The reason for using this algorithm is not clear, however, it is a common scheduling algorithm, widely used within the real-time systems sector.

The R₅J algorithm is a more recent version of the older R₃ algorithm [19]. The original R₃ algorithm was designed for RTSJ, whereas the newer R₅J is designed for both RTSJ and SCJ. One of the key differences between the two algorithms comes from the underlying platform used. R₃ is a platform-independent implementation that has no concept of how long code takes to execute; this leads to a series of unrealistic schedulings. This work was carried out without considering timing properties due to the unavailability of a precise timing model for execution [19]. It does, however, present a method to check time-independent RTSJ programs in JPF using a fixed-priority preemptive scheduling algorithm. The newer R₅J algorithm is based on the Java processor JOP, which is a hardware implementation of the JVM [20]. By using this platform optimised to give time-predictable results, R₅J is a more efficient algorithm that explores fewer impossible schedulings.

The R₅J algorithm is designed to operate with both RTSJ and SCJ programs; however, only a subset of SCJ programs can be checked. More specifically, Level 0 and Level 1 programs are supported, however, there is no support for multi-processors or aperiodic event handlers (APEHs). Multi-processor programs, which are valid at Level 1, are not included as the authors suggest that certification of multi-processor Java applications seems not to be possible in the near future [18]. APEHs are ignored due to the inevitable state explosion that occurs if included. Due to the unknown release times of aperiodic events, including these in the checking process would effectively require the algorithm to nondeterministically decide whether to release a currently non-released APEH after every instruction.

Level 0 SCJ programs contain no preemption or concurrency, and hence have only one possible scheduling. Therefore, as this tool does not make use of JPF’s symbolic execution mode, it cannot discover any additional errors to those found through testing at run-time. At Level 1, more errors can be found than standard testing as every possible scheduling is executed. The R₅J tool can be used to find memory access errors, race conditions, priority ceiling emulation protocol violations, dereferencing of null pointers, invalid arguments to library calls, array bound violations, divisions by zero, and failed assertions.

Two examples are used to evaluate the tool in the paper; these are the Collision Detector (CDₓ) and PapaBench benchmarks. The CDₓ benchmark is a real-time application that simulates potential aircraft collisions by analysing radar frames periodically. Two versions of the original example have been created to comply with the RTSJ and SCJ programming models. These examples are then injected with errors to ensure the tool is able to find them; the errors include an array out of bounds exception, two memory assignment errors (where a reference is created from mission memory to private memory), and a race condition.

The PapaBench example is an open source UAV program which has been successfully applied to several real-world UAVs [18]. The code is made up of the real-time components used to control the UAV; it consists of thirteen periodic tasks and six interrupt sources. The original code has been adapted to make it suitable for R₅J, that is, the interrupts have been modeled as periodic events, and additional code to make the code executable has been included. The injected errors to the PapaBench example include two memory assignment errors as in the CDₓ example. Errors produced are presented in a very similar way to JPF, with the addition of timing information for the release and finish times of individual handlers.

Evaluation The R₅J tool is publicly available to download complete with the two large examples mentioned above, and is executed from the command line on the Linux operating system. As R₅J is able to detect unhandled exceptions and report these as errors to the user, it is also possible to check memory
properties to try and ensure RTSJ or SCJ implementations are memory safe. Null-pointer exceptions are found and reported like in JPF. Violations of the memory structure produce exceptions, for example, when a reference in immortal memory points to an object in mission memory.

As \( R_{SJ} \) operates on top of JPF, which is designed for use on standard Java implementations, the memory model for RTSJ and SCJ is modelled on top of Java. Figure 2.5 shows how immortal and scoped memory areas are used in conjunction with the standard Java programming model. The RTSJ scope stack on the left is used to store reference and primitive values of variables in the individual scopes shown. Each new scope, however, contains a reference to the current memory area; for example, the variable \( ma \) in the first scope points to the immortal memory area \( ima \). The scoped memory areas are represented as objects placed on the heap; this is a way of implementing the scoped memory model on top of JPF. Similarly, objects that are allocated to a specific memory area are placed on the heap like any other object; a reference to the memory area in which it lies is stored as an additional field in the object.

Memory properties are managed in \( R_{SJ} \) using a JPF listener that monitors bytecode execution. Recording information about the program execution at this level allows specific commands that directly affect memory to be tracked; for example, the \texttt{putfield} and \texttt{new} commands. The listener is used to maintain the program stack and memory areas; garbage collection is disabled for memory area objects to maintain the properties of scoped and immortal memory. References to memory areas are not stored on the heap to ensure they are not removed by the garbage collector; they are instead stored in each scope on the stack as described above.

### 2.3.4 Perfect Developer

The Perfect Developer tool is primarily focused on the production of object-oriented programs from formal specifications using refinement [21]. Specifications are expressed in the Perfect language, which is a cross between a model-oriented formal language with sets and mappings such as Z, and an object-oriented programming language with classes, message passing, and inheritance such as Java [22]. Methods are expressed using a design-by-contract approach, where pre conditions and post conditions and invariants are included in the specification to allow program verification with the custom-built automatic theorem prover.

Figure 2.5: RTSJ and SCJ memory representation in \( R_{SJ} \)
It is not unreasonable to compare the Perfect language with JML; both languages facilitate the addition of constrains to programs for verification purposes. The main difference between the languages is found in their use. Perfect annotations are used to produce implementations through code generation, whereas JML annotations are used to add constraints to existing programs.

The semantics of the Perfect language has been designed to facilitate automatic verification [21]. The proof obligations automatically generated ensure that the specification satisfies the required properties, the refinement satisfies the specification, and the program produced terminates. A combination of run-time copy-on-write mechanisms and compile-time flow analysis are used to produce implementations in languages with object reference semantics such as Java and C++.

Perfect Developer uses three refinement stages to produce an implementation from a specification; these are algorithm, data, and delta refinement. Each of these refinement stages are achieved in a single step from the specification to implementation. Algorithm refinement is used to convert methods within the specification into appropriate runnable code.

Data refinement is used to give a concrete implementation of a data representation in the specification. For example, sets are used within the Perfect language, however, these may be refined to arrays in an actual implementation.

Delta refinement allows the enhancement of an existing refinement step, which when applied to the original refinement, allows the expression of an enhanced implementation. For example, class inheritance is used to redefine or extend the behaviour of the parent class; the refinement step required to reach the specification of the child class from the parent class is referred to as a delta refinement.

The need to add assertions to the specification in order to perform automatic verification can make simple specifications seem unnecessarily strong. As a result, users often find themselves producing specifications that are suited to the verifier as opposed to the original problem [23]. Nevertheless, the Perfect Developer tool was produced using its own technology, that is 100,000 lines of Perfect source code. This produces 200,000 lines of C++ and 122,000 proof obligations, of which 90% were successfully discharged automatically. The remaining 10% that could not be discharged automatically arise from missing pre conditions, or a limitation in the theorem prover to dismiss them in a reasonable time. Some of the failed proof obligations resulted from actual bugs in the system [21].

**Evaluation** The tool is publicly available upon request, and provides the user with a simple GUI for writing specifications. It operates on the Windows and Linux platforms and is easy to use; the language is relatively easy to learn in comparison to alternative specification languages.

Perfect Developer is essentially an automatic code generator with the benefit of a theorem prover to ensure the program produced is a valid refinement of the original Perfect specification. The final implementation produced may be a Java program, however, there is no support for any real-time or safety-critical variants such as the RTSJ or SCJ, both of which have a different paradigm. Extensions or modifications to Perfect Developer may be possible to automatically produce SCJ programs, however, the lack of significant documentation means the feasibility of this is not obvious.

Another fundamental factor that restricts the application of Perfect Developer to SCJ programs is the lack of concurrency. To explore whether Perfect Developer can be used in its existing form for the automatic development of Java code applicable to SCJ, we consider the Perfect specification shown in Figure 2.6 of a simple queue. It describes a queue made up of elements of type X, with add and remove operations. The abstract variable queue is used to describe the composition of the queue, which in this case is a sequence of X elements. The function empty is used to check if the queue is empty; this is used later in one of the pre conditions. The add operator has no pre condition; its post condition requires it to ensure the element x is added onto the back of the queue. The remove operator has as pre condition the requirement that the queue is not empty; the post condition states that the head of the queue is removed, and the tail of the queue becomes the remaining queue. Finally, the build operator acts as the constructor; its post conditions states there must exist a queue.

The Java code shown in Figure 2.7 is produced automatically by the Perfect Developer tool from the Perfect specification. The schemas in the specification are translated into methods in the implementation. The functions are not translated directly; when used as pre conditions in the specification, they are implemented as if statements in the code.
As the code is generated automatically, additional methods and libraries have been included by Perfect Developer. The most obvious additions are the methods `_lEqual` and `equals`, which are not defined in the original specification, and also, the inheritance of the Queue class from the `_eAny` class, which is defined in the Perfect Developer library. These additions to the code would not be found in a regular implementation from a programmer, and certainly adds an additional complexity layer when trying to understand the output.

There is, however, a more fundamental reason why the automatic code produced is not suitable for use in the SCJ programming paradigm. The pre condition of the remove method checks that the queue is not empty before manipulating the `queue` variable, however, in checking this pre condition, there is a possibility that an `_xPre` exception could be thrown. In SCJ, it is not always viable to use the `new` operator to create objects in this way as it directly affects the memory.

The simple specification above does not generate any verification conditions in Perfect Developer as there is nothing to prove. The addition of properties to check in the specification would inevitably make the automatic code generation with verification more useful. Pre and post conditions that ensure references do not point to null should allow Perfect Developer to produce implementations that do not suffer from null dereferencing.

Analysis of the tool by Carter et al. found that algorithm refinement cannot be entirely completed automatically; it is often too difficult to automatically discharge the proof obligations with only one refinement step from the specification to the implementation [23]. Additional assertions in the specification are required to help the verifier automatically discharge the proofs; knowing what assertions to write can prove to be time consuming and costly. This is because it is not possible to work in a modular fashion, where smaller pieces of the specification are handled separately, within Perfect Developer.

In summary, Perfect Developer is a useful tool that allows developers to express systems in a formal language with pre and post conditions, and automatically produce an implementation in their chosen programming language, without the burden of learning a difficult formal specification language. Reviews of the tool have suggested that it is well suited for educational purposes, in particular, helping students to learn about formal specifications with pre and post conditions [23].

### 2.3.5 SCJ Checker

The SCJ Checker is a tool designed specifically to verify that a given SCJ program is valid according to the rules related to the SCJ annotations in the code [6]. The SCJ annotations, as discussed previously,
import Ertsys.*;

class _n1_Queue extends _eAny {
    protected _eSeq queue;
    public boolean empty () {
        return (0 == queue._oHash ());
    }
    public void add (_eAny x) {
        queue = queue.append (x);
    }
    public void remove (_eWrapper__eAny x, _eTemplate_0 _t0x) {
        if (_eSystem.enablePre && _eSystem.currentCheckNesting <=
            _eSystem.maxCheckNesting) {
            _eSystem.currentCheckNesting ++;
            try { if (!((!empty ()))) throw new _xPre ("Queue.pd:19,9"); } catch (_xCannotEvaluate _lException) {} _eSystem.currentCheckNesting --;
        }
        x.value = queue.head ();
        queue = queue.tail ();
    }
    public _n1_Queue () {
        super ();
        queue = new _eSeq ();
    }
    public boolean _lEqual (_n1_Queue _vArg_4_9) {
        if (this == _vArg_4_9) {
            return true;
        }
        return _vArg_4_9.queue._lEqual (queue);
    }
    public boolean equals (_eAny _lArg) {
        return _lArg == this || (_lArg != null && _lArg.getClass () ==
            _n1_Queue.class && _lEqual ((_n1_Queue) _lArg));
    }
}

Figure 2.7: Automatically generated Java implementation from Perfect specification

are part of the SCJ specification.

The checker is designed to check all three categories of SCJ annotation: compliance-level, behavioural, and memory safety. This static checking technique is achieved in two passes of the code. The first is used to produce an abstract syntax tree of the program; it also produces a scope stack based on the memory annotations and ensures no duplicates or cycles exist. Every scope stack must end with the immortal memory area. The second pass is used to actually check the program against the rules that accompany the annotations.

Compliance-level annotations are used to control the SCJ levels at which methods and classes are allowed to be used. The following compliance-level rules are built into the SCJ Checker.

- Classes, interfaces, and methods with a particular compliance-level, may only be used in programs of the same or greater level.
- Overridden methods in a subclass must remain at the same level as the method in the parent class.
- Subclasses must not have a lower compliance-level than their parents.

Behavioural annotations are used to define temporal and behavioural properties of methods; these include
allocation, blocking, and mission phase restrictions. The following rules are checked based on these annotations.

- Only methods with the annotation `mayAllocate = true` can allocate new objects.
- Methods not allowed to allocate must not call other methods that can allocate.
- Only methods with the annotation `maySelfSuspend = true` can use potential blocking statements such as `synchronized`.
- Methods not allowed to self suspend must not call other methods that can self suspend.
- Methods restricted to a particular execution phase must only call methods in the same phase, or those suitable for ALL phases.

Memory safety annotations are designed to impose restrictions on the program in order to prevent dangling references. The rules designed to ensure memory safety, and are integrated in the SCJ checker, are below.

- Objects must not be allocated outside the context defined.
- Arrays must not be allocated outside the context of their element type.
- Variables can only be declared in the same scope or those found further up the scope stack; that is, parent scopes cannot contain references to child scopes.
- Static variables must reside in the immortal scope, or have no annotations.
- Overridden methods inherit annotations from the super method.
- Method invocation is only allowed when the allocation context is the same, or the current context is a child of the method’s allocation context.
- The `executeInArea()` method can only be called on parent scopes; the runnable object passed as a parameter must be annotated to run in the corresponding scope.
- The `enterPrivateMemory()` method must be accompanied with an annotation on the runnable object that defines a new scope, where the new scope is a child of the current scope, and also has a `RunsIn` annotation defining its execution in the new scope.
- The `newInstance()` and `newArray()` methods can only be called if the element type is allowed (through annotations) to be allocated in the target scope.
- When casting a variable, the scope must be the same as the target type, or the type’s scope must be undefined at which point the current allocation context must be the same as the current variable.

Rules also need to be defined for unannotated classes; the SCJ checker has the following inbuilt rules.

- Unannotated classes may be instantiated anywhere.
- Unannotated classes may not be passed as parameters outside the context in which they were instantiated.
- Methods that return unannotated objects must allocate them in their own context.
- Unannotated classes may not reference annotated objects.

All of these rules above are checked to ensure that a given SCJ program is correct according to the constraints imposed by the annotations. Some of these annotations impose restrictive constraints on the program, and as such produce a number of limitations with the technique. Firstly, the possible requirement of a class residing in multiple scopes is restricted by assigning each one to a particular scope. To fulfil the requirement, it is necessary to replicate the class for each scope that requires it. Alternatively, the class could be defined without a specific scope, however, this becomes restricted by the rule that unannotated classes may not reference any other annotated objects. Secondly, variable assignments across different scopes raises an error in the checker to prevent the loss of scope knowledge. For example, we cannot assign an object defined in the mission memory area to a variable in an event handler, even though this reference points up the scope stack, which is valid.
Evaluation  

The tool is publicly available to download as part of the oSCJ project, and is easy to use through the Linux command line. It comes with approximately 100 test cases to demonstrate the potential errors the checker can find. It has also been applied to a larger example to assess the scalability of the tool. The example used is based on the CDx benchmark, however, it has been re-implemented as a Level 0 program specifically for SCJ (miniCDj). It consists of 24kloc and required a total of 92 annotations (61 compliance-level and 31 memory safety) to successfully pass through the checker. The authors evaluate the effort required to convert a vanilla RTSJ program into an annotated SCJ program as negligible.

To demonstrate the limitations of the annotations as discussed above, we consider the example shown in Figure 2.8. This simple example shows two classes that have different allocation contexts; class A is allocated in the Mission memory area, whilst class B is allocated in the Handler memory area. The method foo in class B takes an instance of class A as a parameter and assigns it to the object variable o in class B.

The behaviour shown in the example is perfectly acceptable in SCJ, as the variable in the Handler memory area points to an object stored in the Mission memory area. The variable is removed once the handler has finished executing; the mission, and subsequently any instances of class A, cannot be removed until all handlers have finished executing. Therefore, there is never a chance of a dangling reference from the handler to the mission memory area. When passed through the SCJ checker, the following error message is presented.

Warning: Cannot assign expression in scope MyMission to variable in scope MyHandler.

This error is raised, despite the behaviour being perfectly valid, because of the loss of scope knowledge that would occur. The SCJ checker relies on the annotations of each method and class in the program; assignments across scope areas does not maintain the scope rules applied by the annotations.

The SCJ Checker provides guarantees that a given annotated SCJ program does not contain dangling references if accepted by the tool. The ability to guarantee memory safety does, however, come at a cost of the limitations described above. Therefore, it is not possible to verify the memory safety of every valid SCJ program using the SCJ checker, without workarounds such as the duplication of classes. Also, the tool is not built on top of any formal theories for SCJ. Although the guarantees are probably correct, there is currently no way to validate the soundness of the rules for determining memory safety of programs.

2.3.6 Other tools

Lots of work has been dedicated to try and find new and improved ways to verify Java programs. This section briefly describes a few other techniques for completeness.
JML  As briefly described in Section 2.3.1, JML is an annotation language for Java programs. Several tools and techniques to analyse JML annotations and verify Java programs exist; some of the less well known tools that are not commonly used in industry, are mentioned here.

The KeY tool facilitates the verification and specification of JavaCard programs using dynamic logic [24]. Dynamic logic can be viewed as an extension to Hoare logic. The main difference is found in the expression of pre and post conditions. Hoare logic uses pure first-order logic, whereas dynamic logic allows programs (a sequence of valid JavaCard statements) to be included. This allows greater expressiveness over first-order logic. The KeY tool is not limited to JavaCard, however, verification is made easier because of the language limitations. Implementations contain either JML or OCL annotations. UML specifications, JavaCard implementations, and JML/OCL constraints are translated into dynamic logic proof obligations for the inbuilt theorem prover. Reasoning in dynamic logic is based on symbolic execution of the implementation and simple program transformations.

The LOOP tool is used for complete verification of sequential Java implementations [25]. The tool uses a series of semantic-based Hoare logic rules to automatically generate logical theories. These theories can be discharged by the PVS [26] or Isabelle [27] theorem provers.

The Java Applet Correctness Kit (JACK) is designed to automatically verify that a particular sequential Java implementation satisfies its corresponding JML annotations [28]. Proof obligations are generated from JML annotations and discharged by automatic or interactive theorem provers including: Simplify, PVS, Coq [29], and HaRVeY [30]. These obligations are generated automatically from an implementation of the weakest pre condition calculus that works directly on the abstract syntax tree of the implementation, and does not rely on a translation to guarded commands, such as that of ESC/Java2. JACK is capable of automatically producing ‘obvious’ JML annotations; in particular, pre conditions that prevent run-time errors such as null-pointer or array-out-of-bounds exceptions.

The Krakatoa tool translates sequential JML annotated Java programs into the WHY language [31]. WHY is an ML-like language with limited imperative features, including references and exceptions. The WHY tool automatically produces proof obligations for programs written in its own language. The Coq theorem prover is used to manually discharge proofs; however, WHY can be used in conjunction with a wide range of theorem provers including PVS, Isabelle, Simplify, Z3 [32], etc. Proof obligations to guarantee that every dereference is non-null and that every index is within bounds are generated automatically. Integer overflows are not handled; work to map numeric values in Java into WHY integers with pre conditions to ensure integer overflow does not occur is ongoing.

The Jive tool is another sequential Java verification tool that uses Hoare logic and JML annotations to generate proof obligations [33]. Meyer and Poetzsch-Heffter argue against verification based on language semantics because specification and verification at the semantic level is tedious. They dismiss the use of automatic verification condition generation, as in realistic settings, the verification conditions become too large and complex. Instead, Jive uses interactive program verification which allows users to develop proofs around the program with annotations. A syntax-based analysis then follows to generate proof obligations automatically. Jive has no support for exceptions, and documentation is limited.

One of the main limitations of JML is lack of support for concurrency. Although the syntax for the necessary annotations already exists, work is ongoing to bring concurrency to JML [34].

SafeJML  Haddad et al. introduce an extension to JML for real-time systems [35], more specifically for SCJ. The original JML specification has little support for space and timing constraints, therefore they introduce SafeJML to handle these properties. As this is an extension to the original JML, SafeJML is capable of expressing both functional and timing behaviour. Haddad et al. use the oSCJ version of SCJ, as they believe this is currently the most mature implementation, and can use it to directly generate C code. This allows existing low-level worst-case execution time (WCET) analysis tools to be used.

To demonstrate the compatibility of SafeJML and WCET analysis tools, they use RapiTime [36] to check for timing violations. RapiTime uses a hybrid analysis technique, which combines run-time measurements with static analysis to try and compensate for a variety of processors. Firstly, a control-flow graph is constructed through a static model obtained whilst parsing the source code. Various test inputs are then run through the program, and execution times are recorded. The longest recorded time, in conjunction with results from static analysis and user annotations, is used to create a timing model.

The approximation of the WCET is only guaranteed if the user tests contain the necessary inputs to create the worst-case path. Haddad et al. indicate that the tests completed must provide complete
code coverage, however, it is not always the case that users know how to do this; and it could lead to a under-approximation of the WCET. They believe this disadvantage can be eliminated partially by using more sample inputs.

The extensions to JML include a range of annotations to add information about the possible WCET; they are designed to be translated into RapiTime annotations. Examples of these annotations include maximum loop iterations, local worst case for conditional statements, path annotations, and durations. Haddad et al. believe their approach is currently the only effort to produce a specification language for SCJ.

Separation logic   Separation logic is specifically used to reason about programs that use pointers, such as Java. It allows the specifier to define rules related not only to the functionality of the methods of a program, but also about the specific memory locations used by variables. This handles problems that may arise when two variables point to, and subsequently have access to, the same memory location, for example.

jStar uses separation logic and symbolic execution to automatically verify Java programs [37]. Programs are annotated with two pre and post conditions for each method. The first is a static specification that gives a precise definition of the behaviour of the method and its local variables. The second is a dynamic specification that gives a more abstract view; this must be obeyed by all subclasses (behavioural subtyping).

The theorem prover used is called during the symbolic execution of the program to decide implications. Symbolic execution constructs a control-flow graph of the program and calculates all the possible symbolic states after the execution of all nodes in the graph; this can be infinitely large. To reduce the size of the state space, and to guarantee termination during symbolic execution, jStar applies abstraction techniques based on abstraction rules provided by the user.

The main disadvantage with jStar is the expertise required to use the tool. Distefano and Parkinson admit that in its current state, jStar may be too complex for use by programmers. An understanding of how separation logic works, and the annotations required to specify properties in this style, is required.

Separation logic could be applicable to existing SCJ implementations to ensure memory violations are not present. However, when considering an entire refinement strategy from an abstract specification to a concrete implementation, like in [8], this type of error should not be allowed to exist in the first place.

Run-time monitoring   Java PathExplorer (JPaX) provides a run-time verification environment that monitors the execution of Java programs [38]. Given a Java program in bytecode format, JPaX evaluates the execution traces using logic-based monitoring and error-pattern analysis. Logic-based monitoring checks the execution traces of a program against a formal specification written in the Maude [39] language. Logical requirements in Maude are made up from linear temporal logics, future time, and past time logics. Error-pattern analysis compares the execution traces against a number of error-detection algorithms that identify common programming practices that create errors; specifically, JPaX checks for data races and deadlocks. This technique is performed on a single arbitrary trace; it is not sound or complete.

Run-time analysis of SCJ programs could potentially reveal errors as shown for standard Java above; however, using a single trace of execution (in a highly concurrent implementation) can never guarantee to find every possible error. Therefore, its use on programs that require high levels of assurance and comprehensive analysis is not suitable.

2.4 Final Considerations

This chapter has given an overview of the RTSJ and SCJ, and in particular, the different memory models that make them more applicable for real-time and safety-critical systems. The SCJ specification provides a programming paradigm for system development in order to make static verification and certification more accessible.

The number of existing Java verification tools and techniques shows that Java is a popular language with good levels of support for developers. The SCJ language is relatively new, yet it already has an active community in both academia and industry. The work presented in the next chapter aims at adding to the ongoing work into the verification of SCJ, more specifically, a sound static checking technique for SCJ memory safety.
Chapter 3

SCJ memory safety: a sound static checking technique

This chapter expands on the literature presented previously by describing a method to reason about memory safety in SCJ programs. The first part of this chapter describes what the term memory safety means in our approach and how other existing techniques go about checking memory safety of real-time and safety-critical Java programs. We discuss how memory safety is checked with our technique, which involves abstracting away from the raw SCJ programming language and creating a new language with a semantics defined in the UTP.

Section 3.2 introduces this new abstract language for SCJ, called SCJ-M. The new language is designed specifically to capture the structure of SCJ programs and aid analysis. A definition of the language syntax is presented in Z along with details of how the translation from SCJ to SCJ-M is achieved. Section 3.4 contains our first attempt at expressing formal rules to check memory safety. All of the rules we have defined so far are described informally; examples of the underlying formal definitions are included to demonstrate the technique used. Section 3.5 demonstrates how the rules described in Section 3.4 are applied to an example program. Finally, Section 3.6 draws some conclusions.

3.1 Memory safety

As seen in the previous chapter, program verification is concerned with various categories of properties including behavioural, timing, and memory properties. The focus here is on memory safety, which has a new meaning when applied to real-time Java variants such as RTSJ and SCJ. Memory safety in Java relates to the lack of null-pointer dereferences and array-out-of-bound indexes that lead to runtime exceptions. These problems are still a concern in RTSJ and SCJ, however, there is an additional complication: dangling references may exist. As described previously, the presence of a reference that points down the memory structure (to a child memory area) will result in a dangling reference when the child memory area is reclaimed. Dangling references are not a problem in Java as objects on the heap are not garbage collected when there is a reference pointing to them. It is, however, essential that this additional property is considered in the verification of an SCJ program.

There are two techniques that attempt to establish memory safety of SCJ programs [6, 18]. The first of these, RSJ, uses model checking and bytecode listening techniques to record commands that effect memory. As this approach uses model checking, its ability to verify properties is limited by the complexity of the program. As discussed in Section 2.3.3, the technique is not capable of handling aperiodic event handlers due to the state explosion that occurs for all possible release times of the event. Due to this well-known problem associated with model checking, it is worth investigating a static approach.

The second technique, the SCJ Checker, uses static analysis of SCJ annotations to ensure a given program conforms to the SCJ specification. With these annotations, the checker automatically verifies the program against the SCJ memory rules. This technique does not address null-pointer dereferences and array-out-of-bound indexes, but instead checks the memory structure of the program to ensure no illegal references occur (and subsequently no dangling references).
The addition of annotations in the code not only adds information explicitly about the scope and allocation context of a class or method, but simultaneously imposes restrictions on their uses. The rules that accompany the static checker described in Section 2.3.5 lead to limitations that impact on the programming facilities; these are discussed again here in more detail.

**Restricted scope** The main restriction that arises from annotating classes and methods with a specific scope (or memory area) is the limited domain in which the class or method can be instantiated or called. For example, consider a custom data structure. This class must be associated with an `@Scope("Memory Area")` annotation that defines the allocation context in which every instance of this class must exist. If no annotation is provided, the instances may be allocated in any context, however, unannotated classes cannot reference any annotated classes as they may try to allocate an object outside of its allocation context. A similar limitation is found in array definitions and method calls. Array objects must be allocated in the same context as the element type, and method calls can only take place when the allocation context of the method is equal to, or a parent of, the current allocation context. If a class is required in more than one scope, the class definition must be duplicated with a different `@Scope()` annotation for each different allocation context. Another way to avoid this problem would be to remove the scope annotation from the class, hence making it unannotated. However, as discussed previously, unannotated classes can have no interaction with annotated classes because of the scope restrictions.

**Explicit annotations required** Each new scope must be defined explicitly with a new scope annotation (`@DefineScope("Name","Parent")`); this includes temporary private memory areas that may only be used very briefly. This annotation defines a symbolic reference to the memory area. It is useful because the programmer has no direct control over the creation and naming of scopes; the SCJ infrastructure automatically creates new memory areas at specific points of the program, including the mission, handlers, etc. Because the SCJ checker technique relies on everything having a scope, the checker does not rely on the presence of commands such as `EnterPrivateMemory()` to infer a new memory area automatically; it must be defined explicitly with annotations by the user. The runnable method passed as a parameter to the private memory area must also be annotated to run in the matching scope. This presents the same problem above where a particular method cannot be executed in several memory areas as and when necessary - it must be defined explicitly.

**Assignment rules** The rules of assignment in the SCJ memory structure restrict variables from pointing to objects in any child memory area to ensure no dangling references occur. It is, therefore, valid for a variable to point to an object in a parent memory area, however, this is not possible with scope annotations as objects must be stored in their specified memory areas.

In summary, the SCJ memory annotations provided in the specification, by their own definition, impose restrictions on SCJ programs in order to try and guarantee memory safety. In other words, the addition of annotations can result in an otherwise valid SCJ program being evaluated as invalid because of the tighter rules imposed.

Our overall approach is to provide a sound static verification method for memory safety. Most other techniques do not have a proof of soundness for their approach; the method in [40] does provide a sound checking technique for RTSJ programs, however, a large number of annotations are required and not all language features (such as classes and dynamic binding) are supported. The work with SCJ annotations in [6] has shown it is possible to check the validity of an SCJ program in accordance to the specification; however, it imposes a number of restrictions on the developer, and does not have any kind of formal representation or rules in order to substantiate the approach.

Our first goal is to formalise the rules of [6] to verify that the approach in [6] is sound. Figure 3.1 gives an overview of the stages proposed to make this possible. Firstly, a raw SCJ program is translated into a new abstract language called SCJ-M, which is described in the next section. By translating SCJ programs into a new, more abstract, language, we create a fixed structure for all programs; this makes it possible to define rules that are applicable to all programs. The abstraction from SCJ to SCJ-M also abstracts away from properties of the original program that are not associated with its memory safety.
properties; for example, priority, timing, and physical storage properties. All behavioural properties are retained to maintain soundness.

Given a definition of the SCJ-M syntax, a formal semantics in the UTP will be created; a mapping from the SCJ-M language to the semantics will give the underlying meaning of any SCJ-M program. As mentioned in the previous chapter, Cavalcanti et al. have already defined a memory model for SCJ in the UTP [7]. This can be combined with existing UTP theories about object-orientation, for example, to create a complete semantics for SCJ-M and its memory model; this will make proving the soundness of our technique possible.

The final part of the process uses a series of formal rules that define what it means for an SCJ-M program to be memory safe. These rules capture the memory properties described in the SCJ specification, and allow us to analyse any SCJ-M program. Initially, these rules have been formally defined in Z, along with the SCJ-M model; however, no proofs have been completed thus far.

### 3.2 SCJ-M

The SCJ-M language is an important asset in the verification of memory safety, as it provides a consistent description of the structure of an SCJ program; this structure can then be analysed using the formal rules in Section 3.4. The SCJ-M language remains close to the standard SCJ language, however, it includes several abstractions in order to capture the SCJ programming paradigm. These abstractions include the grouping of all static definitions and initialisers together, using a standard naming convention and structure for the safelet, mission sequencer, mission, and handler classes, and the removal of timing, priority, and storage space parameters. As a starting point, we consider SCJ Level 0 programs as these are the most simple; they do, however, still contain the scoped memory structure that makes memory safety in SCJ more complex. Level 1 programs are more complex because they include concurrency between handlers by introducing aperiodic event handlers.

To demonstrate the structure and syntax of SCJ-M, we present a representation of the language in Z. The first definition introduces given-types containing identifiers of variables ($VName$), methods ($MName$), missions ($Mid$), handlers ($Hid$), and classes ($Cid$).

$$[VName, MName, Mid, Hid, Cid]$$

The following three definitions are used to capture the type system in SCJ. The first free-type defines the primitive types; the second free-type defines the two different categories in which a type may be classified: either primitive (with one of the primitive types found in $PrimType$), or an object (linked to a specific class $Cid$). The notation below is used to define a free-type in $Z$; this is essentially a way of defining the set of possible values that belong to the type. For example, the $PrimType$ definition lists some of the possible primitive types found in Java, whilst the $Type$ definition describes the more abstract definition of a type: either primitive or an object.

$$PrimType ::= \text{short} \mid \text{long} \mid \text{int} \mid \text{bool} \mid ...$$

$$Type ::= Primitive(PrimType) \mid Object(Cid)$$

Declarations in SCJ introduce a variable name and a corresponding type; this is captured by a partial function that maps variable names to one of the types defined above.

$$Dec == VName \rightarrow Type$$
As well as having a specific type, each variable also has a specific value associated with it. Primitive types have the expected corresponding carrier sets (e.g. integer and \( \mathbb{N} \)), whereas the value associated with an object is a class (which is referenced here by its identifier).

\[
Value ::= b \langle \langle \text{Boolean} \rangle \rangle | i \langle \langle \mathbb{N} \rangle \rangle | l \langle \langle \mathbb{N} \rangle \rangle | \text{obj} \langle \langle \text{Cid} \rangle \rangle | \ldots
\]

Expressions in SCJ are operations that have values. The following free-type demonstrates a few of the expressions found in SCJ, and shows how they are represented in SCJ-M. Expressions can either be values \( (v) \), variable names \( (vn) \), or operator applications to other expressions such as SCJ’s ‘==’ operator which takes two expressions and compares them to see if they are equal \( (eq \langle \langle \text{Expr} \times \text{Expr} \rangle \rangle) \). Other expressions include the not equal, less than or equal, and greater than or equal comparisons; sum and product operators are also expressions. The definition below shows just some of the expressions found in SCJ; in particular, it does not include method calls at this stage.

\[
Expr ::= v \langle \langle \text{Value} \rangle \rangle | 
vn \langle \langle \text{VName} \rangle \rangle | 
\text{eq} \langle \langle \text{Expr} \times \text{Expr} \rangle \rangle | 
\text{neq} \langle \langle \text{Expr} \times \text{Expr} \rangle \rangle | 
\text{lesseq} \langle \langle \text{Expr} \times \text{Expr} \rangle \rangle | 
\text{greateq} \langle \langle \text{Expr} \times \text{Expr} \rangle \rangle | 
\text{sum} \langle \langle \text{Expr} \times \text{Expr} \rangle \rangle | 
\text{prod} \langle \langle \text{Expr} \times \text{Expr} \rangle \rangle | 
\ldots
\]

Commands in SCJ are represented in SCJ-M as described by the definition below. This definition includes only a selection of the commands to illustrate of the technique. The first command is \textit{skip}, which represents a command that does nothing; this is required in SCJ-M as the fixed structure may introduce blank pieces of code from the original SCJ program. A command may comprise of more than one individual command, therefore, the \textit{seq} \( \langle \langle \text{Com} \times \text{Com} \rangle \rangle \) command represents the sequence of more than one command. Commands can also be declarations that map variable names to types as shown above. Other common commands including the assignment, if, and for operators are also defined. The request termination command is used by the infrastructure as a signal for the current mission to stop executing. Finally, the enter private memory area command takes another command as its parameter and executes this command in a new memory area. Similarly to expressions, method calls are not handled yet as they add additional complexity to the program.

\[
Com ::= \text{skip} | 
\text{seq} \langle \langle \text{Com} \times \text{Com} \rangle \rangle | 
\text{var} \langle \langle \text{Dec} \rangle \rangle | 
\text{ass} \langle \langle \text{VName} \times \text{Expr} \rangle \rangle | 
\text{if} \langle \langle \text{Expr} \times \text{Com} \times \text{Com} \rangle \rangle | 
\text{for} \langle \langle \text{Com} \times \text{Expr} \times \text{Com} \times \text{Com} \rangle \rangle | 
\text{RequestTermination} | 
\text{EnterPrivateMemory} \langle \langle \text{Com} \rangle \rangle | 
\ldots
\]

The following Z schemas are used to define composite constructs used in SCJ-M programs; the first of these represents custom-defined classes. Each class has four fields: an identifier, class variables, the constructor method, and a function that maps method names to commands. The class identifier \( (id : \text{Cid}) \) is the unique identifier of the class; objects use this identifier to determine the underlying type. The \textit{fields} component is of type \textit{Dec}, which is a mapping of variable names to types as described above, and holds the fields declared in the class. The constructor method is defined as its own component in the schema because it is a common occurrence in SCJ classes; overloading is currently not handled, however, the inclusion of the component allows for this in future work. Finally, the \textit{methods} component links the names of the methods in the class to the commands they execute.
In a similar way to classes, handlers also have four components: a handler identifier, fields, constructor, and a handleAsyncEvent (hAe) method. The identifier is required for the mission classes, which list the handlers associated with the mission. By defining the handlers individually, and independently of the mission, they can be used repeatedly by different missions without redefinition. The fields component represents the variables used by the handler. The constr and hAe methods are commands common to all handlers; we assume at this stage that handlers, and other components presented here, do not contain any other custom methods. Additional components are required in the corresponding definitions to handle custom methods; these have not been included initially due to the possibility of parameter passing, which may introduce additional complexity within the scoped memory model.

Missions are defined with six components: a mission identifier, fields, constructor, initialize method, sequence of handler identifiers, and a clean-up method. At Level 0, missions contain only periodic event handlers that execute in parallel; however, the current definition of a mission defines a sequence of handlers. We assume that the order in which the handlers execute (based on their release times) is known, and subsequently transferred to the sequence in which the handlers are defined. Currently this representation is very limited and must be revised to fully capture the possible behaviours of event handlers, including multiple releases of the handler. The identifier is used by the mission sequencer, which contains a sequence of mission identifiers. The fields and constructor components are by now self-explanatory. The initialize method is common to all missions. It is defined as an abstract method that characterises the first of the three phases in the mission: initialisation, execution, and clean-up. The execution phase comes from the parallel execution of the handlers associated with the mission; the handlers component keeps track of all handlers associated with the mission. Finally, the cleanUp component is the final command in the mission, which performs any final commands before the mission is terminated.

The mission sequencer is one of the most basic elements in the SCJ-M language when considering Level 0 programs. The mission sequencer is defined as a sequence of mission identifiers, which represents the order in which missions should be executed. For now, we assume this ordering is known before execution; however, if calculated dynamically, additional components may be required in the mission sequencer in order to capture the behaviour correctly.

The safelet is defined as the top of all SCJ programs; it contains five components: fields, constructor, setUp method, the mission sequencer, and tearDown method. Once again, the fields and constructor
methods are common. The `setUp` method is the first piece of code to execute in the safelet. The `missionSeq` component holds the mission sequencer used to define the missions. Finally, the `tearDown` method executes once all missions have finished executing and the mission sequencer completes.

<table>
<thead>
<tr>
<th>Safelet</th>
</tr>
</thead>
<tbody>
<tr>
<td>fields : Dec</td>
</tr>
<tr>
<td>constr : Com</td>
</tr>
<tr>
<td>setUp : Com</td>
</tr>
<tr>
<td>missionSeq : MissionSeq</td>
</tr>
<tr>
<td>tearDown : Com</td>
</tr>
</tbody>
</table>

To complete the representation of the entire SCJ-M program, the actual program itself must be defined. This includes four components: static definitions, static initialisation, the safelet, and any classes defined. The `static` component holds all static definitions throughout the entire program, as the point of definition is irrelevant; these variables are stored in immortal memory and are initialised by the commands found in `sInit`. The `safelet` component holds the main safelet for the program. And finally, `classes` is a finite set of custom class definitions.

<table>
<thead>
<tr>
<th>SCJ-M</th>
</tr>
</thead>
<tbody>
<tr>
<td>static : Dec</td>
</tr>
<tr>
<td>sInit : Com</td>
</tr>
<tr>
<td>safelet : Safelet</td>
</tr>
<tr>
<td>classes : F Class</td>
</tr>
</tbody>
</table>

Using the language described above, it is possible to express SCJ programs in a standardised form that will aid verification. The current model only captures the basic structure and concepts of SCJ, however, it will evolve to specify more complex Level 1 programs. The structure of an SCJ-M program in plain-text is split into static definitions, the safelet, mission sequencer, missions, handlers, and custom classes; the basic BNF for the language is shown in Figure 3.2.

### 3.3 Translation from SCJ to SCJ-M

The translation from SCJ to SCJ-M presents a number of challenges, especially considering the input to the process is a raw SCJ program and not an idealised or simplified version. We assume that input programs are well typed and well defined. The translation attempts to make little or no change to the actual Java code; it simply rearranges the program based on the abstractions made.

The translation also needs to ensure that all names in the SCJ-M program are unique; this is because the definition of variable names in Z above does not associate names to scopes, therefore, all names throughout all scopes must be unique. It is not enough to rely of the scope rules of Java to distinguish between variables. To ensure this property holds, the translation needs to analyse the input program, and introduce a naming convention that ensures all variables, methods, classes, etc. are defined with unique names.

#### Example SCJ program

To demonstrate the translation from an SCJ program into SCJ-M, we consider the example SCJ program shown in Figure 3.3. This basic example includes the top level safelet, a mission sequencer, a mission, and a periodic event handler; it is designed to be simple to show the basic concepts of how the translation work, as opposed to containing complex functionality. The safelet does nothing other than create an instance of the mission sequencer in the `getSequencer` method with example values for its corresponding priority and storage parameters; there are no commands listed in the `setUp` and `tearDown` methods. The mission sequencer is also very simple, and simply returns a new instance of the `mission1` mission in the `getNextMission` method. The `mission1` class contains only one handler (`handler1`), and does not perform any additional commands in the `initialize` and `cleanUp` methods. The handler is once again created with example priority, periodic, and storage parameters; the specifics of these are not important here. The handler class contains some actual SCJ code to demonstrate how this is translated into SCJ-M. Firstly, the class contains two integer fields, which are set to
values of 50 and 100 respectively in the constructor. The `handleAsyncEvent` method contains the code executed when the handler is triggered; in this example, a simple comparison between the values of the two integers is made, and a corresponding increment for one of the values follows.

**Corresponding SCJ-M program** The result of translating the example described above into SCJ-M is shown in Figure 3.4. Static definitions come first, followed by the safelet and mission sequencer. There are no static definitions in the example provided, and therefore no initialisation code either; the `static`
class Safelet implements Safelet {
    public void setUp() {} 
    public MissionSequencer getSequencer() {
        priority = new PriorityParameter(PriorityScheduler.instance().getNormPriority()),
        storage = new StorageParameters(100000L, 1000, 1000));
        return new missionSequencer(priority, storage);
    }
    public void tearDown() {} 
}

class missionSequencer extends MissionSequencer {
    public missionSequencer (PriorityParameters p, StorageParameters s) {
        super (p, s);
    }
    public Mission getNextMission() {
        return new mission1();
    }
}

class mission1 extends Mission {
    public mission1() {} 
    public void initialize() {
        priority = new PriorityParameters(PriorityScheduler.instance().getNormPriority());
        periodic = new PeriodicParameters(new RelativeTime(0, 0), new RelativeTime(500, 0));
        storage = new StorageParameters(50000L, 1000L, 1000L);
        new handler1(p, r, s);
    }
    public void cleanUp() {} 
}

class handler1 extends PeriodicEventHandler {
    int v1;
    int v2;
    public handler1(PriorityParameters p, PeriodicParameters r, StorageParameters s) {
        super(p, r, s);
        v1 = 50;
        v2 = 100;
    }
    public void handleAsyncEvent() {
        if (v1 < v2) { v1++;
        } else { v2++; }
    }
}

Figure 3.3: Example SCJ program

and sInit components contain nothing and skip respectively. A blank set of declarations indicates that there are no variables, whilst the command skip is the command that does nothing.

The safelet class is also very basic, as it contains no class fields and all methods are empty. The safelet definition in the SCJ-M program is therefore made up of blank definitions and skip commands respectively. At Level 0, there is only one mission sequencer, therefore, the getSequencer method in the original SCJ program can be abstracted away; it is enough to simply have the mission sequencer defined in the SCJ-M representation. Level 2 programs, although not considered here, may contain more than one mission sequencer as they have the ability to nest sequences of missions with other missions. The priority and storage parameters associated with the mission sequencer can also be ignored, as these do not have any affect on the memory safety of the program; scheduling and physical memory space are not the focus of this technique.
```java
static {}

safelet {
    fields {}
    constr { skip; }
    setUp { skip; }
    tearDown { skip; }
}

missionSeq {
    missions { mission1; }
}

mission mission1 {
    fields {}
    constr { skip; }
    initialize { skip; }
    handlers { handler1; }
    cleanup { skip; }
}

handler handler1 {
    fields {
        int v1;
        int v2;
    }
    constr {
        v1 = 50;
        v2 = 100;
    }
    hAe {
        if (v1 < v2) { v1++; }
        else { v2++; }
    }
}
```

Figure 3.4: Example SCJ program expressed in SCJ-M

The `getNextMission` method in the mission sequencer is called to get the first mission of the program, and then subsequently after each mission has finished executing until there are no missions remaining. Once again, it is not necessary to capture the creation and initialisation of the mission class; listing the missions associated with the sequencer is enough for the SCJ-M representation.

During the initialisation phase of the mission, handlers associated with the mission are created. Once again we abstract away from their declaration and simply list the associated handlers as a component in the declaration of the mission. As the `initialize` method in the mission does not perform any other commands, it is represented by the `skip` command in SCJ-M.

The handler class is the only one that contains any example code. The two integer values defined as fields of the class are defined in exactly the same way, but in the `fields` definitions instead. Similarly to the mission sequencer, the priority, periodic, and storage parameters in the constructor are abstracted away, leaving only two assignment commands. The `handleAsyncEvent` method contains the same SCJ code as the actual SCJ program, this is because the abstractions made in SCJ-M are focused mainly on the structure of the programming paradigm, and not the functionality of the code.

One of the aims of our work is to create an automatic translation process from raw SCJ code into the SCJ-M language. Once complete, the new program (in SCJ-M) can be statically checked for memory safety based on a series of formal rules, which are described in the next section.
### 3.4 Memory-safety rules

The representation of the SCJ-M language in Z aids the formalisation of rules that can be used to establish memory-safety. In order to formally specify what memory safety means, we must also formally define the allocation contexts of a program, and the environments in which the program operates. To aid the reader, Table 3.1 shows the meta-variables used throughout this section.

The possible allocation contexts consist of all the SCJ memory areas, and another context in which primitive values are held.

\[
\text{AllocCon ::= Prim} | \\
\text{IMem} | \\
\text{MMem} | \\
\text{PRMem}⟨⟨\text{Hid}⟩⟩ | \\
\text{TPMem}⟨⟨\text{Hid} \times N⟩⟩ | \\
\text{TPMMem}⟨⟨\text{Mid} \times N⟩⟩
\]

The \( \text{Prim} \) context refers to the allocation of primitive values in stacks. \( \text{IMem} \) and \( \text{MMem} \) represent the immortal and mission memory areas respectively. \( \text{PRMem}⟨⟨\text{Hid}⟩⟩ \) is the per release memory area for handlers; the \( \text{Hid} \) is the handler identifier used to link the handler to the specific memory area. \( \text{TPMem}⟨⟨\text{Hid} \times N⟩⟩ \) and \( \text{TPMMem}⟨⟨\text{Mid} \times N⟩⟩ \) are the temporary private memory areas for handlers and missions respectively; once again the \( \text{Hid} \) and \( \text{Mid} \) identifiers are used to link the memory area to the handler or mission, whilst the natural numbers are indices used to identify which memory area, in the stack of temporary memory areas, is being referenced.

Table 3.2 shows the three environments that capture information about an SCJ-M program for use in the memory-safety rules. The first of these environments \( VEnv \) is used to map all of the variables currently in scope to a particular memory area; this allows us to check for violations of referencing rules between scopes. It is defined as a partial function from variable names to allocation contexts.

\[
VEnv ::= VName \rightarrow \text{AllocCon}
\]

In order to keep the environments simple, the requirement of no duplicate names exists. Two variables
Environments | Description
---|---
VEnv | Associates variable names with memory areas
MEnv | Associates mission ids with missions
HEnv | Associates handler ids with handlers

Table 3.2: SCJ-M environments table

with the same name in an actual program may refer to different things in different scopes; all names used in the SCJ-M environments are unique.

The second environment, MEnv, keeps a record of all missions and their identifiers in the program; this is necessary in order to associate the identifier of a particular mission with the actual mission itself. The environment is a partial function from mission identifiers to actual missions.

\[ MEnv = \text{Mid} \to \text{Mission} \]

Mission identifiers are used in the mission sequencer in order to make the actual missions independent of the sequencer; without this technique, missions have to be declared multiple times if they are executed more than once.

The third environment, HEnv, has a very similar concept, except that this environment associates handler identifiers with handlers.

\[ HEnv = \text{Hid} \to \text{Handler} \]

Once again, this allows handlers to be defined independently of the mission, and used in several different missions if necessary.

The mission and handler environments are calculated once, after the translation into SCJ-M is completed. The variable environment evolves and changes as the program executes; therefore, this environment must be maintained throughout the application of rules.

Table 3.3 shows a series of auxiliary functions used by the rules; all those defined with the prefix \( \text{CalcE} \) are used to calculate the environment after a particular part of the program. For example, \( \text{CalcEM} \) is used to calculate the state of the environment after a particular mission has finished executing. It is necessary to calculate the resulting environment because it is this that must be used as the starting point for the next part of the program (in this case, the next mission). The formal definition of the \( \text{CalcEM} \) function in Z is shown below.

\[
\text{CalcEM} : \text{VEnv} \times \text{HEnv} \times \text{Mission} \to \text{VEnv} \\
\forall \text{venv} : \text{VEnv}; \text{henv} : \text{HEnv}; \text{m} : \text{Mission} \bullet \\
\text{CalcEM}(\text{venv}, \text{henv}, \text{m}) = \\
(\text{venv} \oplus \text{CalcEC}(
(\text{CalcEHS}(
(\text{CalcEC}(
(\text{CalcEC}(v\text{env}, \text{m}.\text{fields}, \text{MMem}),
\text{m}.\text{constr}, \text{MMem})),
\text{m}.\text{initialize}, \text{MMem})),
\text{henv}, \text{m}.\text{handlers})),
\text{m}.\text{cleanUp}, \text{MMem})) \triangleright \{ \text{IMem} \}
\]

This function takes three parameters: the old variable environment, the handler environment, and the mission just executed; it then defines an updated variable environment. The function definition states that for all possible input parameters, the environment that exists after the mission has executed is the accumulation of the environments after the fields of the mission are declared, the constructor and
Auxiliary Functions | Action
--- | ---
CalcE | Calculate environment function
CalcEC | Calculate environment function (for commands)
CalcEH | Calculate environment function (for a handler)
CalcEH5 | Calculate environment function (for seq handlers)
CalcEM | Calculate environment function (for a mission)
CalcEMS | Calculate environment function (for seq missions)
ExpAc | Calculate the memory area of a given expression
CalcMEnv | Calculates the mission environment
CalcHEnv | Calculates the handler environment

Table 3.3: SCJ-M auxiliary functions table

initialize methods have executed, all the handlers are complete, and the clean-up method finishes. In order to calculate the environments at each one of these states, the other CalcE functions in Table 3.3 are used for the individual components.

The resulting environment after the mission fields have been declared is obtained by using the function CalcE(venv, m.fields, MMem). This takes the old environment, the declarations to be added, and the allocation context in which they are allocated (mission memory in this case). The environment returned from this function then becomes the first parameter to the CalcEC function that calculates the resulting environment after the constructor method is executed. This process continues until all components of the mission class have been executed, and one final environment exists.

The final environment is then restricted to contain only those variable names that reside in the immortal memory area. This restriction removes a lot of the variables added by the preceding functions, however, this function is designed to calculate the environment after the mission has completed. The variables assigned in the mission memory, or per release memory for any handlers in the mission, are no longer in scope once the mission has completed. It is, however, essential to complete all of the stages in this calculation, because it may be the case that one of the components has allocated or changed something in immortal memory, which is not removed. All of the CalcE functions have a formal definition and are included here when used for completeness.

The definition of memory-safety in Section 3.1 is taken into account in the definition of inference. The top-level rule can be used to determine the safety of an SCJ-M program and its hypothesis can be discharged using the rules for individual commands and expressions. The Z model of SCJ-M described previously shows how the individual components of the language are constructed, including some of the possible commands and expressions found in SCJ.

The individual rules defined to guarantee memory-safety are discussed individually below. Each rule includes a conclusion on the bottom, which defines what the rule is actually trying to establish, and the hypotheses on the top that must all be true in order to guarantee the conclusion.

**MSafe(Program)** The first, and top-level rule, gives hypotheses in order to verify the overall SCJ-M program $p$ is memory safe. In this case, the program is only safe if both the static initialisation commands ($p.sInit$) and the safelet ($p.safelet$) are also safe. Rules that verify the memory-safety of commands also take the allocation context of the command into account (immortal memory in this case); this will be discussed further in the rule for commands below.

$$
\frac {\text{msafe}_1(p.sInit, IMem) \quad \text{msafe}_2(p.safelet)} {\text{msafe}(p)} \quad (1)
$$

where

$e1 = \text{CalcE}(\emptyset, P.static, IMem)$
$e2 = \text{CalcEC}(e1, p.sInit, IMem)$

Note that both hypothesis have different environments: $e1$ and $e2$; this is because the environment can change after a command. To capture the environment updates, the CalcE functions described previously
are used; these take three parameters: the old environment, the specific component to be checked (e.g. declarations, commands, expressions, etc.), and the memory area in which the component is allocated.

The first environment \( e_1 \) is the very first in the program, therefore, the initial environment is empty. The static declarations to be added to the environment are contained in \( P.s.init \), and the memory area in which they are allocated is immortal memory (IMem). The second environment, \( e_2 \), is calculated based on the first environment \( (e_1) \), the commands to initialise the static definitions \( (P.s.init) \), and the immortal memory area. The two \( \text{CalcE} \) functions used in this rule are presented below.

\[
\text{CalcE} : \text{VEnv} \times \text{Dec} \times \text{AllocCon} \rightarrow \text{VEnv}
\]

\[
\forall \text{venv} : \text{VEnv}; \ d : \text{Dec}; \ ac : \text{AllocCon} \bullet \\
\exists \ cid : \text{Cid}; \ ptype : \text{PrimType} \bullet \\
\text{CalcE}((\text{venv}, d, ac) = \text{venv} \oplus \{ \text{vm : dom} d | \text{dvm = Object cid \bullet (vm \mapsto ac)} \} \\
\oplus \{ \text{vm : dom} d | \text{dvm = Primitive \bullet ptype \bullet (vm \mapsto \text{Prim})} \})
\]

This \( \text{CalcE} \) function is used to update the environment after a series of declarations. It takes the old environment and overrides the contents with all of the new variables that represent objects and primitive types in the \( \text{Dec} \) parameter.

\[
\text{CalcEC} : \text{VEnv} \times \text{Com} \times \text{AllocCon} \rightarrow \text{VEnv}
\]

\[
\forall \text{venv} : \text{VEnv}; \ c : \text{Com}; \ ac : \text{AllocCon} \bullet \\
(\exists \ d : \text{Dec} \bullet \\
\quad c = \text{var}d \land \text{CalcEC}(\text{venv}, c, ac) = \text{CalcE}(\text{venv}, d, ac)) \lor \\
(\exists \ \text{vm : VName}; \ \text{exp : Expr}; \ \text{val : Value} \bullet \\
\quad \text{exp = vval} \land c = \text{ass}(\text{vm}, \text{exp}) \land \text{CalcEC}(\text{venv}, c, ac) = \text{venv} \oplus \{(\text{vm} \mapsto \text{ac})\}) \lor \\
(\exists \ c_1, c_2 : \text{Com} \bullet \\
\quad c = \text{seq}(c_1, c_2) \land \text{CalcEC}(\text{venv}, c, ac) = \text{CalcEC}((\text{CalcEC}(\text{venv}, c_1, ac)), c_2, ac)) \lor \\
(\exists \ c_1 : \text{Com} \bullet \\
\quad (\text{let ac1} = \text{EnterPrivMemAC} ac \bullet \\
\quad \quad c = \text{EnterPrivateMemory} c_1 \land \text{CalcEC}(\text{venv}, c, ac) = \text{CalcEC}(\text{venv}, c_1, ac_1) \mapsto \{ac_1\}))
\]

The \( \text{CalcEC} \) function is used to calculate the environment after a particular command. The parameter is first evaluated to establish which command is being analysed; the necessary \( \text{CalcE} \) functions are then applied to the command in order to establish the new environment. For example, in the case of the \( \text{EnterPrivateMemory} c_1 \) command, the result of the function is established by applying the \( \text{CalcEC} \) function on the command to be executed in the new private memory area and subsequently removing the variables that belong to the new memory area. This restriction is applied because the \( \text{CalcE} \) functions are used to calculate the resulting environment after a particular component has executed; therefore, after a private memory area has finished, the variables local to that area are no longer in the environment.

The \( \text{EnterPrivMemAC} \) function is used to determine which memory area should be enter when a new private memory area is created; it is shown below.

\[
\text{EnterPrivMemAC} : \text{AllocCon} \rightarrow \text{AllocCon}
\]

\[
\forall \ ac : \text{AllocCon} \bullet \\
(\exists \ h : \text{Handler}; \ n : \text{N} \bullet \\
\quad ac = \text{PRMem}.id \land \text{EnterPrivMemAC} ac = \text{TPMem}(h.\text{id}, 1) \lor \\
\quad ac = \text{TPMem}(h.\text{id}, n) \land \text{EnterPrivMemAC} ac = \text{TPMem}(h.\text{id}, (n + 1)) ) \lor \\
(\exists \ m : \text{Mission}; \ n : \text{N} \bullet\bullet (ac = \text{MMem} \land \text{EnterPrivMemAC} ac = \text{TPMMem}(m.\text{id}, 1) \lor \\
\quad ac = \text{TPMMem}(m.\text{id}, n) \land \text{EnterPrivMemAC} ac = \text{TPMMem}(m.\text{id}, (n + 1)))
\]

This function takes an allocation context as a parameter, and based on the current context, determines which allocation context should be used next. The new memory area is determined based on the previous memory area. For example, if this is the first call from a handler (where the current memory area is \( \text{PRMem}(h.\text{id}) \)), the new memory area will be the first temporary private memory area associated with the same handler (\( \text{TPMem}(h.\text{id}, 1) \)). All subsequent calls from existing temporary private memory areas simply increase the number associated with the new memory area. When called from the initialisation
phase of a mission, the same principle applies: the first call (where the current memory area is \(MMem\)) will result in the creation of the first temporary private mission memory (\(TPMMem(1)\)), and all subsequent calls simply increase the number used to identify the memory area.

All of the rules presented here have a formal definition in \(Z\); the top-level rule is defined below.

\[
\text{theorem MemSafe} \\
\forall p : SCJ_M \quad \bullet \\
\quad \text{let } \text{env}_1 == \text{CalcE}(\emptyset, p.\text{static}, IMem); \text{menv} == \text{CalcMEM}(p, \text{menv}) \quad \bullet \\
\quad \quad \text{let } \text{env}_2 == \text{CalcC}((\text{env}_1, p.\text{sInit}, IMem)); \text{henv} == \text{CalcHE}(p, \text{menv}) \quad \bullet \\
\quad \quad \quad \text{msafeC}(p.\text{sInit}, \text{env}_1, IMem) = \text{True} \land \\
\quad \quad \quad \quad \text{msafeS}(p.\text{safelet}, \text{env}_2, \text{henv}, \text{menv}) = \text{True} \\
\quad \Rightarrow \text{msafe } p = \text{True}
\]

This theorem states that if both the static initialisation (\(p.\text{sInit}\)) and safelet (\(p.\text{safelet}\)) of a given SCJ program \(p\) are memory safe, then the function \(\text{msafe } p\), which is described below, must also be true. Local variables \(\text{env}_1\) and \(\text{env}_2\) are used to record the environment after the static declarations (for the static initialisation), and after the static initialisation (for the safelet) respectively. The variables \(\text{menv}\) and \(\text{henv}\) represent the mission and handler environment respectively, and are calculated using the functions \(\text{CalcMEM}\) and \(\text{CalcHE}\).

The proof of this theorem, and all subsequent rules is part of our long-term goal to ensure the soundness of this technique.

The function \(\text{msafe } p\) is used to describe the overall meaning of memory-safety in our technique. It is not enough to apply all of these rules and say a program is memory safe, as the underlying semantics of the program are not taken into consideration. The following definition states that an SCJ-M program is memory safe if and only if all of the hypothesis described above are discharged and the semantic definition of the program is a part of \(MSTheory\), which is the set of all programs that satisfy the healthiness conditions outlined in [7].

\[
\text{msafe} : SCJ_M \rightarrow \text{Boolean} \\
\forall P : SCJ_M \quad \bullet \quad \text{msafe } P = \text{True} \Leftrightarrow \text{semantics } P \in MSTheory
\]

The two hypotheses created by the \(\text{msafe } p\) rule must now be checked using the rules below to form a tree; only once every hypothesis leads to a true (or safe) statement, we know the entire program is safe.

\textbf{M Safe(Safelet)} For the safelet to be safe, so must its constructor, setup, mission sequencer, and tear-down components. Each one has its own hypothesis in the following rule.

\[
\frac{
\text{msafe}_{e2}(S.\text{constr}, IMem) \quad \text{msafe}_{e3}(S.\text{setUp}, IMem)}{
\text{msafe}_{e4}(S.\text{missionSeq}) \quad \text{msafe}_{e5}(S.\text{tearDown}, IMem)}
\]  (2)

\[
\text{where} \\
\quad e2 = \text{CalcE}(e1, S.\text{fields}, IMem) \\
\quad e3 = \text{CalcEC}(e2, S.\text{constr}, IMem) \\
\quad e4 = \text{CalcEC}(e3, S.\text{setUp}, IMem) \\
\quad e5 = \text{CalcEMS}(e4, S.\text{missionSeq})
\]

Components \(S.\text{constr}\), \(S.\text{setUp}\), and \(S.\text{tearDown}\) are commands that execute in the immortal memory area; \(S.\text{missionSeq}\) is the mission sequencer for the program; all of these are checked individually with their own variable environments. The initial safelet environment \((e_1)\) is updated for the constructor method to include the declarations of the safelet found in \(S.\text{fields}\). Subsequent hypothesis also have updated environments based on the previous hypothesis.

The \(\text{CalcEMS}\) function is used to calculate the resulting environment after all of the missions have finished executing; it is shown below.
\[
\text{CalcEMS} : VEnv \times HEnv \times MEnv \times \text{MissionSeq} \rightarrow VEnv
\]
\[
\forall \text{venv} : VEnv; \ \text{menv} : MEnv; \ \text{henv} : HEnv; \ \text{ms} : \text{MissionSeq} \bullet
\]
\[
\text{CalcEMS}(\text{venv}, \text{henv}, \text{menv}, \text{ms}) = \bigcup \{ m : \text{ran ms} \bullet (\text{CalcEM}(\text{venv}, \text{henv}, (\text{menv m}))) \} \triangleright \{ \text{IMem} \}
\]

This function applies the \text{CalcEM} function to all missions in a particular mission sequencer and combines all of the resulting environments. The result is then restricted to only those variables found in the immortal memory area, as the mission memory area is not in scope after all of the missions have finished executing.

**MSafe(Missions)**

The mission sequencer in the safelet consists of a sequence of mission identifiers. This rule is designed specifically to extract missions from the sequence one at a time and establish whether they are safe.

\[
\frac{\text{msafe}_1(MEnv(Mid)) \ \text{msafe}_2(MS)}{\text{msafe}_1(< Mid > ^{MS})} \quad (3)
\]

where
\[
e_2 = \text{CalcEM}(e_1, MEnv(Mid))
\]

As already said, the environment \text{MEnv} is used to map mission identifiers (\text{Mid}) to actual missions; it is the actual missions that are used in the hypothesis to determine safety. The second hypothesis can be resolved by using the same rule again to consider the remaining mission identifiers in the sequence; eventually this will be the empty sequence, which is always safe by definition below. The environment is updated for the remaining mission identifiers as the previous mission may have changed the environment. This particular version of the environment function (\text{CalcEM}) updates the environment and then removes all variables in the environment from the previous mission and all its handler’s per release and temporary private memory areas; this is because all memory areas associated with the previous mission and its handlers are reclaimed once the mission finishes. The definition of \text{CalcEM} was presented as an example previously.

**MSafe(Empty Sequence)**

The definition above contains a sequence of mission identifiers; these sequences are broken down to check individual missions, however, the sequence will eventually become empty. To ensure that the empty sequence does not present a problem when applying rules, it is always safe.

\[
\frac{\text{True}}{\text{msafe}_1(<>)} \quad (7)
\]

**MSafe(Mission)**

This rule takes a single mission as its parameter. The mission components outlined in the SCJ-M Z model must be checked individually and are included in individual hypotheses that must all be true.

\[
\frac{\text{msafe}_2(M.\text{constr}, \text{MMem}) \ \text{msafe}_3(M.\text{initialize}, \text{MMem})}{\text{msafe}_4(M.\text{handlers}) \ \text{msafe}_5(M.\text{cleanUp}, \text{MMem})}
\]

where
\[
e_2 = \text{CalcE}(e_1, M.\text{fields}, \text{MMem})
\]
\[
e_3 = \text{CalcEC}(e_2, M.\text{constr}, \text{MMem})
\]
\[
e_4 = \text{CalcEC}(e_3, M.\text{initialize}, \text{MMem})
\]
\[
e_5 = \text{CalcEHS}(e_4, M.\text{handlers})
\]
The $M\text{.constr}$, $M\text{.initialize}$, and $M\text{.cleanUp}$ components are commands that execute in the mission memory area, the $M\text{.handlers}$ component is a sequence of handler identifiers associated with the mission. The initial environment is updated to include the fields of the mission, which are stored in the mission memory area, before the constructor executes. The fields of missions are stored in the immortal memory area because this is where they are allocated; only once the mission starts executing in its own memory area can the methods use the mission memory area.

The $CalcEHS$ function is similar to the $CalcEMS$ function in a sense that it calculates the resulting environment after all of the handlers have finished executing; the definition is shown below.

$CalcEHS : VEnv \times HEnv \times \text{seq Hid} \rightarrow VEnv$

$\forall venv : VEnv; henv : HEnv; hs : \text{seq Hid} \bullet$

$CalcEHS(venv, henv, hs) =$

$\bigcup \{ h : \text{ran hs} \bullet (CalcEH(venv, (henv h))) \} \triangleright \{IMem, MMem\}$

MSafe(Handlers) Similarly to the mission sequencer, this rule is used to check a sequence of handlers given the identifier. Each handler is identified through the $HEnv$ environment and checked individually.

$\text{msafe}_1(\text{HEnv}(\text{Hid})) \text{ msafe}_2(\text{HS})$

$\text{msafe}_3(< \text{Hid} \triangleright \text{H}>)$  

where $e2 = CalcEH(e1, \text{HEnv}(\text{Hid}))$

The environment is updated for each subsequent handler; the $CalcEH$ function removes any variables assigned in the per release or temporary private memory areas belonging to the handler, as these are not in scope for any other handlers. The definition of $CalcEH$ is shown below.

$CalcEH : VEnv \times \text{Handler} \rightarrow VEnv$

$\forall venv : VEnv; h : \text{Handler} \bullet$

$CalcEH(venv, h) = CalcEC(
\text{(CalcEC}(\n\text{(CalcE(\nvenv, h.fields, MMem)), \n.h.constr, MMem)), \n.h.hAe, (PRMem h.id))) \triangleright \{IMem, MMem\}$

MSafe(Handler) The rule to determine memory-safety for a handler is comprised of two hypotheses: one for the constructor, and one for the handleAsyncevent (or $hAe$) method.

$\text{msafe}_2(H\text{.constr}, PRMem(H.id)) \text{ msafe}_3(H.hAe, PRMem(H.id))$

$\text{msafe}_1(H)$  

where $e2 = CalcE(e1, H\text{.fields, MMem})$

$e3 = CalcEC(e2, H\text{.constr}, PRMem(H.id))$

As handlers are created during the initialisation phase of a mission, the fields are allocated in the mission memory. Once created, the per release memory area for the handler exists, and all further methods and declarations are stored in there.

MSafe(Declaration) This rule is used to check memory-safety for declarations. As the very nature of a declaration is to define a new variable with a specific type, it only adds to the existing state of the environment, and does not change what already exists. Therefore, it is safe to assume that all declarations
are safe, irrespective of what memory area (MA) the declaration is found in.

\[
msafe_e(var(d), MA) \\
\text{True} \tag{8}
\]

**MSafe(Expression)** An example set of expressions for SCJ-M were presented earlier in the SCJ-M \(Z\) model. Whilst we currently do not model method calls in expressions, we can safely assume that expressions are always safe in all memory areas.

\[
msafe_e(exp, MA) \tag{9}
\]

When an expression contains a method call it must be evaluated further; we do not know what side-effects may occur from the method call, and what impact it may have on the environment. Method calls may also contain assignments that create problems with the environment.

**MSafe(Command)** Methods in an SCJ-M program are defined as commands; however, a command may be a sequence of multiple commands. This simple rule ensures that a sequence of commands is safe by analysing them individually.

\[
msafe_{e1}(c_1) \quad msafe_{e2}(c_2) \\
msafe_{e1}(c_1; c_2, MA) \tag{10}
\]

where

\[
e_2 = CalcEC(e1, c1, MA)
\]

The environment is updated after the first command as commands can change or add to the previous environment. This rule is applicable to all commands in any memory area; the memory area defined (MA) is substituted for the actual memory area when the rule is applied.

**MSafe(Assignment)** In our current version of SCJ-M, assignments consist of a variable name and an expression; we do not yet consider the use of field accesses. As shown previously, expressions can be of a variety of forms including variable names, values, boolean comparisons, etc. In the following rule, the function \(ExpAc\) is used to analyse the expression and determine the memory area associated with it. For example, if the expression is another variable name, the memory area is retrieved by looking at the respective mapping in the environment. If the expression is a value, the memory area is either \(Prim\) for primitive types or the current memory area for objects.

\[
ExpAc(exp, e) \mapsto e(vn) \in msafeRefs \\
msafe_e(ass(vn, exp), MA) \tag{11}
\]

Part of the function \(ExpAC\) is defined below, this is not yet complete as all possible expressions in SCJ are currently not handled.

\[
\begin{align*}
\forall exp : \text{Expr}; \ venv : \text{VEnv} & \quad \bullet \\
(\exists \ vname : \text{VName} \quad \bullet \ exp = \text{vnamevarn} \land \ ExpAc(exp, venv) = \text{vnamevarn}) & \lor \\
(\exists \ exp1 : \text{Expr} \quad \bullet \ exp = eq(exp, exp1) \land \ ExpAc(exp, venv) = \text{Prim}) & \lor \\
(\exists \ exp1 : \text{Expr} \quad \bullet \ exp = neq(exp, exp1) \land \ ExpAc(exp, venv) = \text{Prim}) & \lor \\
& \ldots
\end{align*}
\]

**MSafe(Request Termination)** The request termination command is a method in the mission class that initiates the finalisation of a mission. The necessary steps to make this happen are conducted by
the infrastructure; we assume that any method performed by the infrastructure is safe.

\[
\text{True} \quad \text{msafe}_e(\text{RequestTermination}, \text{MA}) \quad (12)
\]

**MSafe(Enter Private Memory)** Temporary private memory areas can be entered during the initialisation phase of a mission and in handlers. The enter private memory area command takes a separate command as its parameter; the rule to establish memory-safety checks the new command is safe in the new memory area.

\[
\text{msafe}_e(c, \text{EnterPrivMemAC}(\text{MA})) \quad (13)
\]

The new memory area is determined with the \text{EnterPrivMemAC} function that was defined previously; it analyses the previous memory area and returns the new context based on the previous one.

**MSafe(If)** The if command is essentially made up of two separate commands, of which only one is executed based on the condition of the expression.

\[
\text{msafe}_e(c_1, \text{MA}) \quad \text{msafe}_e(c_2, \text{MA}) \quad\frac{\text{msafe}_e(\text{if}(\text{exp}, c_1, c_2), \text{MA})}{(14)}
\]

There is no need to update the environment inbetween the two commands here, because there is no sequence of execution. Either \(c_1\) or \(c_2\) will execute in the environment \(e\).

**MSafe(For)** The for loop is one that often presents problems in verification techniques. The rule presented here is a very simple one that does not consider all possible behaviours of the for loop; however, it does present the basic structure and concept of how the SCJ-M for loop is checked for memory-safety.

\[
\text{msafe}_e(c_1, \text{MA}) \quad \text{msafe}_e(c_3, \text{MA}) \quad \text{msafe}_e(c_2, \text{MA}) \quad\frac{\text{msafe}_e(\text{for}(c_1, \text{exp}, c_2, c_3), \text{MA})}{(15)}
\]

where

\[e_2 = \text{CalcEC}(e_1, c_1, \text{MA})\]
\[e_3 = \text{CalcEC}(e_2, c_3, \text{MA})\]

The command \(c_1\) refers to the initialisation of the for loop, \(\text{exp}\) is the expression that determines whether the body \((c_4)\) is executed another time, and \(c_3\) is the command executed after each iteration of the body. Therefore the standard execution pattern of a for loop that does not include extra conditionals, or break early is \(c_1, c_3,\) and \(c_2\). The environment used by each command is updated after the previous command as the sequence of commands rely on the resulting environment of the previous command.

**MSafe(Skip)** The final rule presented here is for the command \text{skip}, which represents commands that do nothing. If the command does not do anything, it is always safe.

\[
\text{True} \quad \text{msafe}_e(\text{skip}, \text{MA}) \quad (16)
\]

All of the rules described above give a starting point to define whether an SCJ-M program is memory safe. To better illustrate how these rules are applied to a program, the next section demonstrates how they would be used on the example in Figure 3.4.


3.5 Applied example

Figure 3.5 shows the proof tree that is constructed by applying the memory-safety rules above to the SCJ-M program presented earlier in Figure 3.4. The tree grows upwards in a left-to-right, depth-first manner, starting from the msafe\((P)\) rule; the rules applied at each level of the tree are identified using the number in parenthesis on the right hand side.

As the example presented is very simple, most of the methods contain only the skip command, which is always memory safe; however, the structure of the proof tree is clearly demonstrated. The sequence of missions and handlers in rules (3) and (5) respectively are singular in this example; however, in larger examples, the rule would be repeatedly applied to the remaining sequence to ensure all missions and handlers are safe.

No variables are declared in the program until the fields of the handler are defined; therefore, the environment remains empty \(\emptyset\) until the application of rule (6), which is the first point at which the handler fields are defined. Variables \(v1\) and \(v2\) are defined as integers and are stored in the Prim allocation context; the environment is updated to include the mappings from these variable names to the associated allocation contexts.

Rule (11), or the assignment rule, is applied four times at the top of this proof tree; this is because the constr and hAe methods contain two assignment commands respectively. The constr method is used to assign the initial values to the variables, and the hAe method increments one of the two variables based on the result of an expression. This rule is versatile as it can analyse different assignments; for example, the first two assignments assign primitive values to variable names, which is always safe. The second set of assignments are used to store the incremented value associated with a variable name, to the same variable name; this is an assignment of an expression to a variable name. Rule (11) states that this is valid if the allocation context of the expression is equal to, or a parent of, the variable name. In this case, the expression is an additive operator on two primitive values, therefore, it is also safe.

3.6 Final considerations

This chapter has presented the basic concepts behind our approach: we aim to provide a sound static checking technique to verify memory-safety of SCJ programs that is backed up by formal foundations and proof. Our definition of memory-safety is justified in the context of SCJ to include additional checks based on the scoped memory model. Several other techniques that complement our work are discussed and evaluated; however, the use of program annotations is one that we will try to minimise.

The SCJ language establishes a novel programming paradigm that has allowed us to define a new abstract language, SCJ-M, which gives a consistent and well-defined structure to SCJ programs. The syntax of the SCJ-M language has been defined and, by means of example, demonstrated its ability to express simple SCJ programs. The development of the language will be an ongoing part of the work in order to express any SCJ program.

We have also presented a series of formal rules that every SCJ-M program must adhere to in order to be classed as memory safe. Currently these rules only cater for the basic concepts of the SCJ programming language; however, further evaluation of examples will present more challenges and quickly enrich the domain of programs that can be checked. In particular, our technique does not currently handle parametrised method calls, multi-dimensional data structures, and dynamic binding, to mention a few.

The next chapter concludes this document by summarising the existing tools and techniques presented in Chapter 2, and describing further the long term goals of this technique and how they might be achieved.
Figure 3.5: Example application of memory-safety rules
Chapter 4

Conclusion

In this document, we have tried to establish some of the verification techniques used to reason about the Java programming language. Furthermore, we have analysed the applicability of these techniques to the SCJ language. The range of tools available for Java checking is vast; properties checked or verified by each tool differ greatly and the choice of tool to use is focused heavily on the level of guarantees required. Most of the techniques discussed rely heavily on user annotations to express pre conditions and post conditions, and invariants.

In order to establish a better focus for the tool analysis, we looked more specifically at memory properties and how these are checked. It is true to say that none of the Java specific tools discussed in Section 2.3 are able to verify the extended definition of SCJ memory safety described in Section 3.1. Those designed specifically for SCJ make an attempt to verify memory properties of SCJ programs; however, there is no evidence of the underlying formal approach used to ensure the techniques are correct or sound.

The lack of a formal approach has justified the work described in Chapter 3, which presents the beginning of a technique to prove the rules of memory safety in SCJ are sound. The first part of this approach involves the definition of SCJ-M, an abstract language for SCJ, which captures the SCJ programming paradigm in a consistent and structured format. The chapter also introduces the initial rules that can be applied to the SCJ-M language in order to verify memory safety. These rules are specific to the structure of SCJ-M, and express what it means for each component of the programming paradigm to be memory safe. For example, the safelet, mission sequencer, missions, etc. all have a set of hypotheses in their corresponding rules that must be true in order for the component itself to be memory safe. The application of these rules creates an entire program creates a proof tree from the top level safelet down to the individual commands found in class methods.

4.1 Further Work

The work presented in Chapter 3 is only the first part of what is required in order to achieve a sound static checking technique for SCJ programs. The following two sections describe what must be done to achieve this goal, and what the overall technique could be used for in the longer term.

4.1.1 Thesis goal

The definition of SCJ-M is currently very basic and is only capable of capturing the simplest of programs; it is not, for example, able to handle method calls, multi-dimensional data types, parameter passing, or dynamic binding. The programming paradigm of Level 0 SCJ programs has been captured, however, Level 1 programs and a comprehensive domain of programming constructs is yet to be developed. Similarly, the definitions of rules provided only cater for the domain of programs that SCJ-M is able to express. The goal of the SCJ-M language is to be able to express all programming constructs found in Level 1 SCJ programs, including all of the necessary rules required to express memory safety of these constructs.

The basic concepts of manual translation from SCJ to SCJ-M have been presented in the previous chapter, however, the difficulties of translating more complex SCJ program are yet to be seen. One of the
main problems likely to arise will come from the lack of an idealised input program. The ability to pickup any given SCJ program and translate it into SCJ-M is one of the key properties that makes this technique distinct from those that impose programming restrictions on the input language. Once a comprehensive translation strategy has been developed, a tool to automate the process will be developed. The design of the tool should allow for additional programming features to be translated at a later stage: this allows a spiral development cycle to be adopted to gradually add features to the translation, including all of the definitions and proofs required to ensure memory safety is achieved.

In order to understand the meaning of programs written in SCJ-M, a formal semantics of the language must be defined; we will use the UTP to do this. By using the UTP, we can combine several existing theories about object orientation and the SCJ memory model, for example. The flexibility of reusing UTP theories gives a modular approach to verification as the required theories can be used or created as and when required. It is essential to define a formal semantics of the language in order to maintain the soundness of our approach.

The rules defined to guarantee memory safety must also be proved to be correct; currently these are expressed in Z, however, it is not our intention to use Z to define every part of the technique. These rules make up only half of the process required to determine whether a program is memory safe: a mapping from SCJ-M programs to their underlying semantics will be created in order to check the program adheres to the memory safety healthiness conditions defined in [7].

Given the memory safety rules and the underlying healthiness conditions for SCJ-M programs, the main goal of the thesis is to prove the main memory safe theorem described in the previous chapter.

\[
\text{theorem MemSafe} \\
\forall p : \text{SCJ-M} \bullet \text{msafe } p \Rightarrow \text{msafe } p = \text{True}
\]

This theorem is used to check the SCJ-M program using all of the rules defined in the previous chapter; in order to prove that this correct, we must also prove that all of the individual rules are also correct. The theorem concludes that if all of the components of the SCJ-M program are memory safe, then the function \text{msafe } p must also evaluate to true. The \text{msafe } p function, as discussed previously, states that an SCJ-M program can only be memory safe if and only if the underlying semantics of the same program also satisfy the healthiness conditions; the definition of \text{msafe } p is included again below.

\[
\text{msafe } : \text{SCJ-M} \rightarrow \text{Boolean} \\
\forall P : \text{SCJ-M} \bullet \text{msafe } P = \text{True} \Leftrightarrow \text{semantics } P \in \text{MSTheory}
\]

The proof of the theorem above, and the definition of a semantics for SCJ-M may present many challenges when trying to add language constructs and programming features. We hope to minimise the number of annotations that may be required in our technique to ensure the set of programs applicable to our technique is not heavily restricted and to reduce the annotation burden on the programmer.

The current state of the work has not yet managed to capture the full programming paradigm of Level 0 SCJ programs, however, the goal of the thesis is to fully capture Level 1 programs.

4.1.2 Above and beyond

The goal of the thesis is to develop a sound static checking technique to verify SCJ programs are memory safe; however, there are some parts of the overall approach that will be not be addressed in the given time frame. For example, we do not consider the larger challenge of certification for safety-critical systems. Our technique will provide assurances that a particular SCJ program is memory safe according to the SCJ specification, however, there is no indication on whether this technique will be sufficient for certification authorities.
The goal of the thesis is to capture the paradigm of Level 1 SCJ programs; however, SCJ has three compliance levels. It is easy to assume that our technique will be applicable to Level 0 if we are able to handle Level 1 programs, however, Level 2 programs present another significant layer of complexity. Level 2 programs contain a high level of concurrency with the possibility of creating nested missions and further mission sequencers, all of which have the potential to execute in parallel. Level 2 programs present an even larger challenge for techniques based on model checking as the state explosion that comes with the complexity of concurrency at Level 2 is too large to handle.

The work presented in [8] describes a technique to express SCJ programs in the formal language Circus. This work is complementary to that proposed here and may present opportunities to combine techniques. SCJ programs translated into SCJ-M may be further translated into Circus, for example, and act as a way of verifying that a given implementation is a refinement of some abstract specification.
Bibliography


