The Safety-Critical Java Memory Model: a formal account

Ana Cavalcanti, Andy Wellings, and Jim Woodcock

University of York, Department of Computer Science, York, UK

Abstract. Safety-Critical Java (SCJ) is a version of Java for real-time programming that facilitates certification of implementations of safety-critical systems. It is the result of an international effort involving industry and academia. What we provide here is, as far as we know, the first formalisation of the SCJ model of memory regions. We use the Unifying Theories of Programming (UTP) to enable the integration of our theory with refinement models for object-orientation and concurrency. In developing the SCJ theory, we also make a contribution to the UTP by providing a general theory of invariants (of which the SCJ theory is an instance). Our results are a first essential ingredient to formalise the novel programming paradigm embedded in SCJ, and enable the justification and development of reasoning techniques based on refinement.

Keywords. semantics, UTP, integration, refinement.

1 Introduction

Two language (subsets) have dominated high-integrity real-time engineering. Ada [2], which provides good support through its Spark [1] and Ravenscar subsets [4] and the Spark Examiner Toolset, has a limited community. Safe(r) subsets of C/C++ are often the choice, but lack support for formal development. In both cases, various modern programming features found useful in other sectors of the software industry are left out on the grounds of safety.

An international effort has produced a high-integrity real-time version of Java: Safety-Critical Java (SCJ) [13]. It achieves a compromise between the safety of Ada and the popularity of C/C++, and provides an ambitious novel take on the combined safe use of object orientation and real-time programming. SCJ lacks, however, a formal underpinning for its programming models. In this paper, we provide a formalisation for its memory management model.

SCJ is based on a subset of Java augmented by the Real-Time Specification for Java (RTSJ) [19]. To understand the full implications of the SCJ memory model, it is necessary to appreciate the run-time data structures maintained by a Java Virtual Machine. The main concern is the heap and the stacks. All objects are placed on the heap, which is scanned by a garbage collector to remove any that are unreachable. Variables that are local to methods are stored in a stack; each thread of control has an associated stack. Variables and object fields can be of a primitive type (int, short, and so on) or of a reference type. We ignore here all issues associated with native methods.

The RTSJ supplements Java's garbage-collected heap memory model with support for memory regions [18] called *memory areas*. As with the Java heap, these regions are used to store dynamically created objects.

SCJ restricts the RTSJ memory model to prohibit use of the heap. The RTSJ and SCJ introduce two new memory areas: scoped and immortal memory. Objects allocated in a scoped memory have a lifetime that is determined by the number of threads that are currently using that scoped memory area. When there are no such threads, all the objects are collected. In contrast, objects created in immortal memory have a lifetime equal to that of the program. A program can have many scoped memory areas, but only a single instance of immortal memory. To avoid dangling references, there are rules that must be obeyed by reference assignments. Violation of these rules results in runtime exceptions. SCJ defines a fixed structure for the use of scoped memories.

In Java, programmers need not be concerned with memory management. In contrast, in SCJ (and the RTSJ), a programmer must consider in which area to create objects according to their anticipated lifetime. Tools and techniques are needed to ensure efficient use of memory and absence of run-time errors.

SCJ includes annotations that can be used to document programs, and enable static verification of properties including memory safety. The work in [17] presents rules for use of the annotations, and a tool that checks statically that these rules are followed. It is not trivial to convince ourselves that the rules proposed achieve the level of memory safety claimed. While we do not necessarily expect to find any problems, the formalisation of the memory model is essential for the justification of the soundness of such techniques.

Our first contribution is an informal description of the SCJ memory model that explains the rationale for its design. (For a discussion of the design of the concurrency model, we refer to [20].) As a second contribution, we provide a relational semantics for this model; it is based on Hoare and He's Unifying Theories of Programming (UTP) [10]. Finally, we present a general UTP theory for operation and state invariants, which we instantiate to capture in an elegant and concise way the properties of the SCJ structure of memory areas.

The UTP is a relational framework that supports refinement-based reasoning about a variety of paradigms. It covers models for concurrent, functional and logic programming, for instance. It has also been used to define constructs related to object-orientation [15] and time [16]. By casting the SCJ memory model in the UTP, we pave the way for its integration with these theories, that cater for other, also very important, aspects of an SCJ program.

Next, we present informally the SCJ memory model; an introduction to the UTP is provided in Section 3. Section 4 presents a UTP theory for program invariants. In Section 5, we use those results to formalise the SCJ memory model. We draw our conclusions, and discuss related and future work in Section 6.

2 Safety-Critical Java memory model

SCJ recognises that safety-critical software varies considerably in complexity.

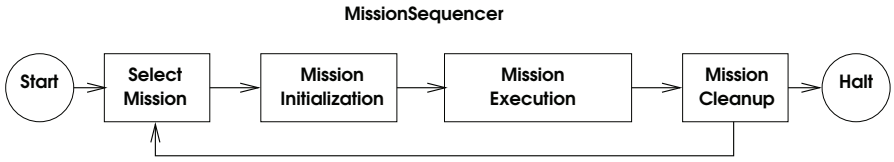

Fig. 1. Safety Critical Mission Phases (taken from [13])

At one end of the spectrum, the application consists of a single thread executing a single function on a single processor with a simple timing constraint. At the other end, it is multithreaded executing in multiple modes on multiple processors. Consequently, there are three compliance levels for SCJ programs and implementations. In this work, we are concerned with Level 1, which, roughly, corresponds in complexity to the Ravenscar profile for Ada.

The SCJ programming model is based on the notion of missions, which are managed by a mission sequencer (see Figure 1). At Level 1, missions may be composed into sequences, but nested missions are prohibited. A Level 1 mission consists of a bounded set of asynchronous event handlers (ASEH). Here, these can be considered as being equivalent to real-time threads. Both periodic and aperiodic threads are supported. Each thread executes a sequence of releases that are either time triggered (periodic) or event triggered (aperiodic). Consequently, an SCJ program is a concurrent program with threads of control for the main program, the mission sequencer, and one for each of the ASEHs.

The main goal of the SCJ memory model is to support dynamic memory management. Traditionally, safety-critical systems do not allocate memory during the execution of a mission due to (a) the error-prone nature of manual allocation and deallocation schemes (typified by `malloc` and `free` in C), and (b) the complexity of automatic deallocation schemes based on garbage collection.

The region-based approach of the RTSJ provides safer and more predictable support for dynamic memory management, but the overall model is still complex. SCJ, consequently, constrains the use of its features: garbage collection is not supported, and only a restricted version of the scoped memory model is provided.

Basically, the structure of the memory areas is fixed as shown in Figure 2. Every thread of control in an SCJ program has a default memory allocation context. This is the area in which created objects are placed. The main program's thread of control has immortal memory as its default allocation context. It is this thread that, for instance, creates the mission sequencer and any objects that should exist throughout the lifetime of the program.

The mission sequencer's thread of control is started with immortal memory as its default allocation context. It creates the mission memory, a scoped area that becomes the default allocation context for a mission. There is no thread of control associated with a mission. Instead, the mission sequencer's thread performs the mission initialisation, during which the ASEHs are created. The mission memory is cleared at the end of each mission. Any objects that must remain across missions must be stored in immortal memory.

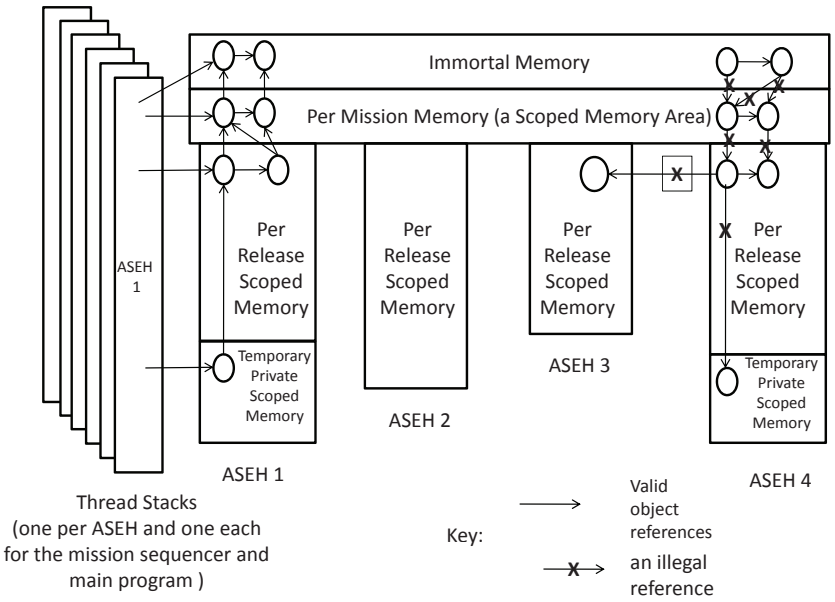

Fig. 2. SCJ memory model

Each ASEH has an associated per-release memory area: the default memory allocation context for its releases. It is cleared at the end of each release, for reuse in the next release. Any object that is required to live across releases must be placed in mission memory. An ASEH can create a temporary private scoped memory area and change its default allocation context to the newly created area. More than one of these can be created and they are used in a LIFO manner. The stack of private temporary memory areas arises from nested calls to a `create` method. As the inner calls are finished, memory areas are popped off.

In the example shown in Figure 2 there are, therefore, six thread-of-control stacks: one for the main program, one for the mission sequencer, and one for each ASEH; a single immortal memory – accessible by all threads of control; a single mission memory – accessible by the ASEHs and the mission sequencer; one private per-release memory area for each ASEH – accessible only by the associated ASEH; and a stack of temporary private scoped memory area for each ASEH – accessible only by the associated ASEH.

The aim of this restricted model is to ensure that dangling references cannot occur, and that programs are amenable to static analysis techniques that can determine the absence of run-time errors, such as illegal-assignment errors. A tool is provided in [17]. Section 5 formalises this model in the UTP.

3 Unifying theories of programming

In the UTP, relations are defined by predicates over an alphabet (set) of obser-

vational variables that record information about the behaviour of a program. In the theory of general relations, these include the programming variables v, and their dashed counterparts v', with v used to refer to an initial observation of the value of v, and v' to a later observation. The set of undecorated (unprimed) variables in the alphabet αP of a predicate P is called its input alphabet $in\alpha P$, and the set of dashed variables is its output alphabet $out\alpha P$. A condition is a predicate whose alphabet includes only input variables.

Theories are characterised by an alphabet and by healthiness conditions defined by monotonic idempotent functions from predicates to predicates. The predicates of a theory with an alphabet A are all the predicates on A which are fixed points of the healthiness conditions. As an example, we consider designs.

The general theory of relations does not distinguish between terminating and nonterminating programs. This is achieved in the theory of designs, which includes two extra boolean observational variables to record the start and the termination of a program: ok and ok'. The monotonic idempotents used to specify the healthiness conditions for designs can be defined as follows.

H1 $\quad P = ok \Rightarrow P$

H2 $\quad P = P\,;\, J,\ \textbf{where}\ J \mathrel{\widehat{=}} (ok \Rightarrow ok') \wedge v' = v$

If P is **H1**-healthy, then it makes no restrictions on the final value of variables before it starts. If P is **H2**-healthy, then termination must be a possible outcome from every initial state. The functional composition of **H1** and **H2** is named **H**.

Every design D can be written in the form $P \vdash Q$, where P is its precondition, and Q its postcondition; $P \vdash Q$ is defined as $ok \wedge P \Rightarrow ok' \wedge Q$. Precisely, every design D can be written as $\neg\, D^f \vdash D^t$, where f is the boolean false, t is true, and D^b is the predicate $D[b/ok']$ obtained by substituting b for ok' in D.

Typically, a theory defines a number of programming operators of interest. Common operators like assignment, sequence, and conditional, are defined for general relations. A conditional is written as $P \lhd b \rhd Q$; its behaviour is (described by) P if the condition b holds, else it is defined by Q.

$$P \lhd b \rhd Q \mathrel{\widehat{=}} (b \wedge P) \vee (\neg b \wedge Q),\ \textbf{where}\ \alpha(b) \subseteq \alpha(P) = \alpha(Q).$$

Sequence is relational composition.

$$P;\ Q \mathrel{\widehat{=}} \exists\, w_0 \bullet P[w_0/w'] \wedge Q[w_0/w],\ \textbf{where}\ out\alpha(P) = in\alpha(Q)' = w'$$

The relation $P;\ Q$ is defined by a quantification that relates the intermediate values of the variables. It is required that $out\alpha(P)$ is equal to $in\alpha(Q)'$, which is named w'. The sets w, w', and w_0 are used as lists that enumerate the variables of w and the corresponding decorated variables in the same order.

A central concern of the UTP is refinement. A program P is refined by a program Q, written $P \sqsubseteq Q$, if, and only if, $P \Leftarrow Q$, for all possible values of the variables of the alphabet. The set of alphabetised predicates form a complete lattice with this ordering. Recursion is modelled by weakest fixed points.

The design that models skip, the program that terminates without changing any variable, is $\mathit{II} \mathrel{\widehat{=}} (\mathit{true} \vdash v' = v)$, where v is the list of programming variables in the alphabet. Interestingly, II is the left identity of sequential composition, but not necessarily the right identity. This requires that the precondition does not contain dashed variables, a property not adequate, for instance, in the theory of reactive designs used as a concurrency model (for CSP).

A theory needs to be closed with respect to the programming operators: they need to take healthy predicates to healthy predicates, so that they can be used to define models compositionally. In the next section, we provide some general results for the healthiness conditions of a theory of designs with invariants.

4 Invariants in the UTP

In [10], designs are used to construct more general relations to model, for example, reactive programs. For these, even in the presence of divergence, some properties hold. In [9], we take this approach in a theory for objects and sharing as available in Java. Our theory, in that case, captures physical properties of sharing; for instance, variables that share a location have the same value.

On the other hand, when an SCJ program aborts, there is no guarantee that its restrictions on memory areas are maintained. We, therefore, present our theory as a subset of the theory of designs. Other examples of subtheories of designs are presented in the line of work established in [12], which provides UTP theories for BPEL-like languages, with new forms of nontermination to handle exceptions. Here, we provide a general account of design subtheories characterised by invariants and with the standard notion of termination.

It is in the spirit of the UTP to define theories for particular programming features, and combine them to capture more complex paradigms. In this line, it could be conceivable to treat the memory structure of SCJ programs and termination separately. We would characterise a subtheory of relations using a healthiness condition **HSCJ**, for instance, and then use **H** to embed it in the theory of designs. For an **HSCJ**-healthy predicate P whose alphabet does not include ok and ok', however, the design $\mathbf{H}(P)$ is $\neg\ P \vdash \mathit{false}$. Its precondition considers the possibility of **HSCJ** not holding (even in an non-abortive state), and, in this case, it is miraculous. What we need instead is a theory that allows for the memory restrictions to be violated just in the case of nontermination.

In what follows, subtheories of designs are defined by healthiness conditions that either capture operation invariants or invariants of a single state observation. In both cases, invariants are only broken by nontermination.

4.1 Operation invariants

For an operation invariant defined by a predicate Ψ, the subtheory of designs that satisfy this invariant is characterised by the healthiness condition **OIH**.

$$\mathbf{OIH}(\Psi) \qquad D = D \wedge (ok \wedge \neg\ D^f \Rightarrow \Psi)$$

An $\mathbf{OIH}(\Psi)$-healthy design ensures that, when its precondition holds, so does Ψ.

Theorem 1. $\mathbf{OIH}(\Psi)$ *is a monotonic idempotent function on designs.*

Proof. First, we show that $\mathbf{OIH}(\Psi)(D)$ is a design.

$$
\begin{array}{ll}
& \mathbf{OIH}(\Psi)(D) \\
= & (\neg\, D^f \vdash D^t) \wedge (ok \wedge \neg\, D^f \Rightarrow \Psi) \\
& \qquad\qquad [\text{property of designs and definition of } \mathbf{OIH}(\Psi)] \\
= & (ok \wedge \neg\, D^f \Rightarrow ok' \wedge D^t) \wedge (ok \wedge \neg\, D^f \Rightarrow \Psi) \qquad [\text{definition}] \\
= & \neg\, D^f \vdash D^t \wedge \Psi \qquad [\text{propositional calculus and definition of designs}]
\end{array}
$$

Since $ok \wedge \neg\, (\neg\, D^f \vdash D^t \wedge \Psi)^f = ok \wedge \neg\, D^f$, then $\mathbf{OIH}(\Psi)$ is idempotent. Finally, to establish monotonicity, we consider designs D_1 and D_2 such that $D_1 \Rightarrow D_2$. That $\mathbf{OIH}(\Psi)(D_1) \Rightarrow \mathbf{OIH}(\Psi)(D_2)$, follows from $\neg\, D_2^f \Rightarrow \neg\, D_1^f$. □

We define the healthy identity $\mathit{II}_{OI}(\Psi) \mathrel{\widehat{=}} \mathbf{OIH}(\Psi)(\mathit{II})$. For reflexive Ψ, that is, for those such that $\Psi[v/v']$, we have that $\mathit{II}_{OI}(\Psi)$ is the sequence left unit.

Theorem 2. *If* Ψ *is reflexive,* $\mathit{II}_{OI}(\Psi);\ D = D$*, for every* $\mathbf{OIH}(\Psi)$*-healthy* D*.*

Proof.

$$
\begin{array}{ll}
& \mathit{II}_{OI}(\Psi);\ D \\
= & \mathbf{OIH}(\Psi)(\mathit{II});\ \mathbf{OIH}(\Psi)(D) \qquad [\text{definition of } \mathit{II}_{OI} \text{ and } D \text{ is } \mathbf{OIH}(\Psi)\text{-healthy}] \\
= & (\mathit{true} \vdash v' = v \wedge \Psi);\ (\neg\, D^f \vdash D^t \wedge \Psi) \qquad [\text{Theorem 1}] \\
= & ok \wedge \neg\, (\Psi[v/v'] \wedge D^f) \Rightarrow ok' \wedge \Psi[v/v'] \wedge D^t \wedge \Psi \\
& \qquad\qquad [\text{definition of sequence and design, and predicate calculus}] \\
= & ok \wedge \neg\, D^f \Rightarrow ok' \wedge D^t \wedge \Psi \qquad [\Psi \text{ is reflexive}] \\
= & D \qquad [\text{definition of design, Theorem 1, and D is } \mathbf{OIH}(\Psi)\text{-healthy}]
\end{array}
$$

□

$\mathit{II}_{OI}(\Psi)$ is not necessarily the right unit. Like in the theory of general designs, this requires that the precondition refers to no dashed variables. Proofs of this and other results mentioned below can be found in [6].

$\mathbf{OIH}(\Psi)$ is closed with respect to conjunction, disjunction (which models nondeterminism) and conditional. For closedness with respect to sequence, we need Ψ to be transitive, that is, $(\Psi;\ \Psi) \Rightarrow \Psi$. The set of $\mathbf{OIH}(\Psi)$-healthy designs is a complete lattice, since it is the image of a monotonic idempotent healthiness condition [10]. So, recursion can still be defined using weakest fixed points. The bottom and top of the lattice are the same as that for the lattice of designs: abort, that is, the design $(\mathit{false} \vdash \mathit{true})$, and magic, $(\mathit{true} \vdash \mathit{false})$.

4.2 State invariants

For a state invariant defined by a condition ψ, the subtheory of designs whose input variables satisfy ψ is characterised by the following healthiness condition.

$\mathbf{ISH}(\psi) \qquad D = D \vee (ok \wedge \neg\, D^f \wedge \psi \Rightarrow ok' \wedge D^t)$

The invariant ψ is part of the precondition of $\mathbf{ISH}(\psi)$-healthy D.

Theorem 3. $\mathbf{ISH}(\psi)$ *is an idempotent function on designs.*

Proof. First, we show that $\mathbf{ISH}(\psi)(D)$ is a design.

$$
\begin{aligned}
&\mathbf{ISH}(\psi)(D) \\
&= (\neg\, D^f \vdash D^t) \lor (ok \land \neg\, D^f \land \psi \Rightarrow ok' \land D^t) \\
&\qquad\qquad [\text{property of designs and definition of } \mathbf{ISH}(\Psi)] \\
&= (ok \land \neg\, D^f \Rightarrow ok' \land D^t) \lor (ok \land \neg\, D^f \land \psi \Rightarrow ok' \land D^t) \qquad [\text{definition}] \\
&= \neg\, ok \lor D^f \lor \neg\, \psi \lor ok' \land D^t \qquad [\text{propositional calculus}] \\
&= \neg\, D^f \land \psi \vdash D^t \qquad [\text{propositional calculus and definition of designs}]
\end{aligned}
$$

The arguments for idempotence and monotonicity are similar to those used in Theorem 1. □

We define the healthy identity $\mathit{II}_{IS}(\psi) \mathrel{\widehat{=}} \mathbf{ISH}(\psi)(\mathit{II})$. It is indeed the left-unit of sequence; this is a simple consequence of the definitions of $\mathit{II}_{IS}(\psi)$ and sequence, and Theorem 3 above. Again, right unit does not hold in all cases.

$\mathbf{ISH}(\psi)$ is closed with respect to conjunction, disjunction, conditional, and sequence. The bottom of the lattice that it defines is abort, but the top is $(\psi \vdash \mathit{false})$. This is miraculous only when ψ holds.

The subtheory of designs whose output variables satisfy ψ' is characterised by the following healthiness condition. The predicate ψ' is that obtained by substituting all output alphabet variables for their input counterparts in ψ.

$$\mathbf{OSH}(\psi) \qquad D = D \land (ok \land \neg\, D^f \land \psi \Rightarrow \psi')$$

We observe that $\mathbf{OSH}(\psi)$ can be defined as $\mathbf{OIH}(\psi \Rightarrow \psi')$, and that $\psi \Rightarrow \psi'$ is reflexive and transitive. So, it satisfies all the properties discussed in the previous section. Most importantly, as shown below, $\mathbf{ISH}(\psi)$ and $\mathbf{OSH}(\psi)$ commute.

Theorem 4. $\mathbf{ISH}(\psi)$ *and* $\mathbf{OSH}(\psi)$ *commute.*

Proof.

$$
\begin{aligned}
&\mathbf{OSH}(\psi) \circ \mathbf{ISH}(\psi)(D) \\
&= \mathbf{OSH}(\psi)(\neg\, ok \lor D^f \lor \neg\, \psi \lor ok' \land D^t) \\
&\qquad [\text{function composition, Theorem 3, and propositional calculus}] \\
&= \neg\, (\neg\, ok \lor D^f \lor \neg\, \psi) \vdash (\neg\, ok \lor D^f \lor \neg\, \psi \lor D^t) \land (\psi \Rightarrow \psi') \\
&\qquad [\text{Theorem 1 and propositional calculus}] \\
&= \neg\, D^f \land \psi \vdash ok \land \neg\, D^f \Rightarrow (D^t \land (\psi \Rightarrow \psi')) \\
&\qquad [\text{propositional calculus and definition of designs}] \\
&= \neg\, (\neg\, ok \lor D^f) \land \psi \vdash \neg\, ok \lor D^f \lor D^t \land (\psi \Rightarrow \psi') \\
&\qquad [\text{propositional calculus and definition of designs}] \\
&= \mathbf{ISH}(\neg\, ok \lor D^f \lor ok' \land D^t \land (\psi \Rightarrow \psi')) \\
&\qquad [\text{propositional calculus, definition of designs, and Theorem 3}]
\end{aligned}
$$

$= \mathbf{ISH}(\neg\, D^f \vdash D^t \wedge (\psi \Rightarrow \psi'))$ [definition of designs]

$= \mathbf{ISH}(\psi) \circ \mathbf{OSH}(\psi)(D)$ [Theorem 1 and function composition]

□

As shown above, an $\mathbf{ISH}(\psi)$ and $\mathbf{OSH}(\psi)$-healthy design D can be written as $(\neg\, D^f \wedge \psi \vdash D^t \wedge \psi')$, so that ψ is assumed and established. Since $\mathbf{ISH}(\psi)$ and $\mathbf{OSH}(\psi)$ are idempotent, by Theorem 4, so is $\mathbf{SIH}(\psi) \mathrel{\widehat{=}} \mathbf{ISH}(\psi) \circ \mathbf{OSH}(\psi)$ [10]; this is our healthiness condition for a theory with state invariant ψ.

When healthiness functions **C1** and **C2** commute, then every predicate that is (**C1** ∘ **C2**)-healthy is also **C1** and **C2**-healthy. From this and the theorems above and in Section 4.1, we can conclude that $\mathbf{SIH}(\psi)$ distributes through conjunction, disjunction, conditional, and sequence.

Finally, for operation and state invariants Ψ_1 and ψ_2, $\mathbf{OIH}(\Psi_1)$ and $\mathbf{SIH}(\psi_2)$ commute. So, using an argument similar to that above, we can conclude that a theory characterised by $\mathbf{IH}(\Psi_1, \psi_2) \mathrel{\widehat{=}} \mathbf{OIH}(\Psi_1) \circ \mathbf{SIH}(\psi_2)$ is closed with respect to conjunction, disjunction, conditional, and sequence. The same applies to theories characterised by two operation invariants Ψ_1 and Ψ_2; $\mathbf{OIH}(\Psi_1)$ and $\mathbf{OIH}(\Psi_2)$ commute, and define a theory with invariant $\Psi_1 \wedge \Psi_2$. A similar result holds for state invariants ψ_1 and ψ_1. The UTP theory for the SCJ memory model presented in the next section combines several operation and state invariants.

5 A theory for the Safety-Critical Java memory model

In this section, we consider first a theory that captures the structure of memory areas in SCJ. Afterwards, we extend it to take into account the values of the variables stored in the memory areas.

Type definitions The elements of the stacks (for the program, mission sequencer, and handlers) are frames, which define a context of execution for a method. To provide a model for a frame, we introduce the notion of a variable name as an element of the unspecified set $VName$, and of a reference: from a set Ref. We also define the set of values as $Value = PValue \cup Ref$, where $PValue$ is the unspecified set of primitive values and the special value $null$. With these, we can define $Frame = VName \pfun Value$, so that a frame is a partial function associating the names of the variables in scope to their values.

A function $refsIn : Frame \to \mathbb{F}\, Ref$ defines the finite set of references (to objects in a memory area) in the stack. It is defined as $refsIn\, f = \mathrm{ran}(f \rhd Ref)$, using the range restriction operator $\rhd$.

We identify a memory area with its contents; we do not capture issues related to size. Concretely, we define the set $MAreaC = Ref \pfun OValue$ of memory contents, where $OValue$ is the set of record (object) values: functions that associate fields to their values, that is, $OValue = VName \pfun Value$.

We also define two functions $refsRes, refsIn : MAreaC \to \mathbb{F}\, Ref$. For a memory area ma, the set $refsRes\, ma$ contains the references that identify objects that reside in ma. The references used in these objects (to refer to other objects in the same or in other memory areas) are those in $refsIn\, ma$. Precisely,

refsRes ma = dom *ma*, and *refsIn ma* = $\bigcup$(ran(⦇ (_ ▷ *Ref*)⦇ ran *ma* ⦈ ⦈). For a memory area *ma* (or more precisely, for the contents *ma* of a memory area), ran *ma* gives its objects. By using relational image _(⦇ _ ⦈) to apply the operator (_ ▷ *Ref*) to all of them, we project out all their fields with a primitive or *null* value. The ranges of these objects are the references used in *ma*; distributed union provides a single set containing all of them.

In order to identify the handlers of a mission, we consider the set *HName*. It contains valid handler identifiers, or names.

The alphabet of our theory includes eight extra observational variables defined below, and their dashed counterparts, in addition to *ok*, *ok′*, and the programming variables (and their dashed counterparts). We have nine healthiness conditions, which are also specified and discussed in the sequel.

Alphabet First, we have the stacks *pStack*, *msStack* : stack *Frame* for the program and the mission sequencer. The set *handlers* : $\mathbb{F}$ *HName* records the handlers of the current mission, and the variable *hStack* : *handlers* → stack *Frame* groups their stacks as a total function associating each handler to its stack.

To record the memory areas, we have first *immortal*, *mission* : *MAreaC*. The per-release memory areas are grouped in *perR* : *handlers* → *MAreaC*. The temporary private memory areas are organised in a stack as recorded in the alphabet variable *tPriv* : *handlers* → stack *MAreaC*. A simple model for a stack is, of course, a sequence, whose last element is the top of the stack.

A stacked temporary private memory area is called a parent in relation to all those areas of the same handler that are stacked afterwards. More generally, the immortal memory area is the parent of the mission memory area, which is a parent of all per-release memory areas. Additionally, the per-release memory area of a handler is a parent of all its stacked temporary private memory areas.

Healthiness conditions We can only add object values to the immortal area. This is an operation invariant, and gives rise to our first healthiness condition **HSCJ1**. To define it, we introduce a function *profile* : *MAreaC* → (*Ref* ⇸ $\mathbb{F}$ *VName*). For a memory area *ma*, the function *profile ma* associates each reference residing in *ma* with the set of fields of the object that it identifies in *ma*. This is the domain of the function (in *OValue*) that defines that object. Formally, we have *profile ma* = {*r* : dom *ma* • *r* ↦ dom(*ma r*) }. Our healthiness condition **HSCJ1** requires that the immortal memory is changed only by adding new references to its profile. Existing references remain, and the structure of the objects to which they point (as captured by their sets of field names) is preserved.

HSCJ1 ≙ **OIH**(*profile immortal* ⊆ *profile immortal′*)

The operation invariant for **HSCJ1** is reflexive and transitive, because ⊆ is.

The references in the program stack can only target objects in the immortal memory. This is specified by the healthiness condition **HSCJ2**, which uses a lifted version of *refsIn* : stack *Frame* → $\mathbb{F}$ *Ref* that applies to stacks of frames *sf* (instead of frames or memory areas). We can define it in terms of the version

of *refsIn* for frames as $refsIn\ sf = \bigcup(refsIn(\!|\ \mathrm{ran}\ sf\ |\!))$. The range of sf is a set of frames; we use relational image to apply *refsIn* to all of them. The distributed union collects together all references occurring in all frames of sf.

$$\mathbf{HSCJ2} \ \widehat{=}\ \mathbf{SIH}(refsIn\ pStack \subseteq refsRes\ immortal)$$

Analogously, the references in the immortal memory can only target objects in the immortal memory itself. This is the state invariant specified below.

$$\mathbf{HSCJ3} \ \widehat{=}\ \mathbf{SIH}(refsIn\ immortal \subseteq refsRes\ immortal)$$

Similarly, the references in the mission-sequencer stack and in the mission memory area are for objects either in the immortal or in the mission memory areas. To capture this healthiness condition, we define $refsRes : \mathbb{F}\ MAreaC \to \mathbb{F}\ Ref$, for a set of memory areas *mas* as $refsRes\ mas = \bigcup(refsRes(\!|\ mas\ |\!))$. It collects the references in each of the memory areas in *mas*.

$$\mathbf{HSCJ4} \ \widehat{=}\ \mathbf{SIH}(refsIn\ msStack \subseteq refsRes\ \{immortal, mission\})$$
$$\mathbf{HSCJ5} \ \widehat{=}\ \mathbf{SIH}(refsIn\ mission \subseteq refsRes\ \{immortal, mission\})$$

For each handler, the references in its stack are for objects in its own temporary private areas, in its own per-release area, or in the mission or immortal memory.

$$\mathbf{HSCJ6} \ \widehat{=}\ \mathbf{SIH}\left(\begin{array}{l}\forall\, h : handlers \bullet \\ \quad refsIn\ (hStack\ h) \subseteq \\ \quad\quad refsRes\ (\{immortal, mission, perR\ h\} \cup \mathrm{ran}(tPriv\ h))\end{array}\right)$$

For each handler, the references in its per-release memory area are for objects in that same area, or in the mission or immortal memory areas.

$$\mathbf{HSCJ7} \ \widehat{=}\ \mathbf{SIH}\left(\begin{array}{l}\forall\, h : handlers \bullet \\ \quad refsIn\ (perR\ h) \subseteq refsRes\{immortal, mission, perR\ h\}\end{array}\right)$$

Finally, in a temporary private memory area of any handler, the references target objects that can be in the immortal memory, in the mission memory, in the associated per-release memory for the same handler, in a parent stacked area, or in that same temporary private memory area.

$$\mathbf{HSCJ8} \ \widehat{=}$$
$$\mathbf{SIH}\left(\begin{array}{l}\forall\, h : handlers;\ i : 1\,..\,\#(tPriv\ h) \bullet \\ \quad refsIn\ (tPriv\ h\ i) \subseteq \\ \quad\quad refsRes(\{immortal, mission, perR\ h\} \cup \{j : 1\,..\,i \bullet tPriv\ h\ j\})\end{array}\right)$$

We use $\#s$ to denote the size of the sequence (or stack) s.

Finally, the memory areas are disjoint in their use of the reference space.

$$\mathbf{HSCJ9} \ \widehat{=}$$
$$\mathbf{SIH}(\mathsf{disjoint}\ \langle refsRes\ immortal, refsRes\ mission\rangle \frown \mathsf{seqPR}\ perR \frown \mathsf{seqTP}\ tPriv)$$

We use $\mathsf{seqPR}\ perR$ and $\mathsf{seqTP}\ tPriv$ to denote the sequences of sets of references

residing in the per-release and temporary private memory areas in *perR* and *tPriv*. We omit the formal definition of these functions.

Our theory contains the fixed points of the healthiness conditions above. They are the fixed points of **HSCJ**, which we define as the composition of all the healthiness functions. With the results in Section 4, we conclude that **HSCJ** is closed with respect to conjunction, disjunction, conditional, and sequence.

The healthiness conditions **HSCJ2** to **HSCJ8** are enough to ensure that every SCJ program makes a safe use of memory, in the sense that, at no point, it has a variable in a stack whose value is a dangling reference or can be used to reach a dangling reference. **HSCJ10** justifies the treatment of the separate memory areas as a single global memory. We take advantage of that in the sequel, when we consider the value of the variables in the stacks.

What we have not captured is the fact that during the lifetime of a mission, we can only add objects to the mission memory. Similarly, objects can only be added to each of the per-release and temporary private memory areas until they are cleared. For the immortal memory, we have **HSCJ1**. It is not the case, however, that $\mathit{profile\ mission} \subseteq \mathit{profile\ mission}'$, for example, is an invariant of our theory. Since the mission area can be cleared, and later reused when a new mission is started, then there is no guarantee that $\mathit{mission}'$ is at all related to *mission* in every pair of observations of an SCJ program. The same comments apply to the per-release and private temporary memory areas in *perR* and *tPriv* in relation to the handler releases and the calls to the `create` method.

To establish the required properties, we need to keep a record of the sequence of missions that have been executed. Additionally, to restrict the use of the per-release and temporary private memory areas, during the execution of a mission, we need to keep the history of releases and calls to the `create` method for each ASEH. Details of how history can be added to our theory can be found in [6]. For instance, we keep a sequence of identifiers for the missions that have been executed, with a special identifier used to indicate that there is no mission currently executing. This approach is similar to that adopted in [16, 5] to cater for passage of time in the UTP theories for timed and synchronous processes.

Programming variables and their values Programming variables in the alphabet can be specification or allocated variables. Specification variables are used to write abstract definitions of the behaviour of programs; they model, for instance, inputs and outputs. Allocated variables are included in one of the stacks.

Our next three healthiness conditions require that the value of every allocated variable in the alphabet is in accordance with what is recorded in the stacks. To define them, we use a function $\mathit{vars} : \mathrm{stack}\, \mathit{Frame} \to \mathbb{F}\ \mathit{VName}$ that characterises the set of active variables in a given stack: those in the domains of the frames; formally, $\mathit{vars}\ \mathit{sf} = \bigcup \mathrm{dom}(\!|\ \mathrm{ran}\ \mathit{sf}\ |\!)$, provided there are no redeclarations, that is, $\mathsf{disjoint}\ \{i : 1 \mathinner{..} \#\mathit{sf} \bullet i \mapsto \mathrm{dom}(\mathit{sf}\ i)\}$. (As usual, we assume that variable names are not reused to avoid handling stacks of values for alphabet variables.)

The value of a variable *vn* (according to a stack *sf* and its associated memory areas *mas*) is characterised by a set A of sequences of variable names, and a function V that associates some of these sequences to primitive values. If the

value associated with vn in sf is primitive or $null$, then $\langle vn \rangle$ is the only sequence in A. If, on the other hand, the value of vn is a reference (to an object), then we also have all the (possibly infinite) extensions of $\langle vn \rangle$ that identify a field of that object, or a field of one of its fields, and so on. The function V associates the sequences of variable names that identify a variable or an object field with a primitive or null value to this value. This characterisation of values is the same used in [9], where we have defined a UTP theory for the Java memory model that captures the structure of objects and sharing.

Formally, we define the value $!(vn, sf, mas)$ using a dereferencing function $!\_ : VName \times \mathrm{stack}\, Frame \times \mathbb{F}\, MAreaC \to \mathbb{P}\, SName \times (SName \twoheadrightarrow PValue)$, specified as $!(vn, sf, mas) = (A(vn, sf, mas), V(vn, sf, mas))$. Here, $SName$ is the set of possibly infinite sequences of variable names (from $VName$). The set $SName \twoheadrightarrow PValue$ is that of the finite partial functions from $SName$ to $PValue$.

The set $A(vn, sf, mas)$ is defined as shown below.

$$A(vn, sf, mas) = \left\{ sn : SName \mid \left(\begin{array}{l} head\ sn = vn \land \\ \textbf{let}\ u == sval(vn, sf) \bullet \\ \quad u \in PValue \land tail\ sn = \langle\,\rangle \lor path(tail\ sn, \bigcup mas, u) \end{array} \right) \right\}$$

Here $sval(vn, sf) = (\bigcup(\mathrm{ran}\ sf))\ vn$ is the value of vn as recorded in sf. The fact that there are no variable redeclarations guarantees that $(\bigcup(\mathrm{ran}\ sf))$ is a function. The condition $path(sn, ma, r)$ requires that the sequence of variable names sn identifies a path in the memory area ma starting from the reference r. We use it above to make sure that the extensions of $\langle vn \rangle$ are in accordance with the information in the memory areas mas. With the assumption that they are disjoint, we consider $\bigcup mas$. The starting reference is the value u of vn in sf.

The formal definition of $path(sn, ma, r)$ is as follows. We require the existence of a (possibly infinite) sequence sr of references that can be traversed using the sequence of names sn. The last value of sn, if any, might be a primitive value, rather than a reference, so the type of sr is $SVal$, the set of sequences of values.

$$path(sn, ma, r) \Leftrightarrow \left(\exists\, sr : SVal \bullet \left(\begin{array}{l} head\ sr = r \land \\ \left(\forall\ i : \mathrm{dom}\ sn \bullet \left(\begin{array}{l} (sr\ i) \in \mathrm{dom}\ ma \land (sn\ i) \in \mathrm{dom}(ma\ (sr\ i)) \land \\ sr(i+1) = ma\ (sr\ i)\ (sn\ i) \end{array} \right) \right) \end{array} \right) \right)$$

For each name $sn\ i$ in sn, the corresponding value $sr\ i$ in sr must be a reference in ma to an object $ma\ (sr\ i)$ with a field named $sn\ i$. Additionally, the next value $sr\ (i+1)$ in sr must be the value $ma\ (sr\ i)\ (sn\ i)$ of that field.

The definition of $V(vn, sf, mas)$ is in many ways similar, and we omit it here.

The condition **HV1** requires that the value of every variable v in the program stack is given by $pStack$ itself and its associated $immortal$ area.

$$\mathbf{HV1} \ \widehat{=}\ \mathbf{SIH}(\textstyle\bigwedge v : vars(pStack) \bullet v =!(v, pStack, \{immortal\}))$$

The healthiness conditions **HV2** and **HV3** are similar. The former considers the

mission-sequencer stack, and the latter the handlers stacks.

$$\mathbf{HV2} \mathrel{\widehat=} \mathbf{SIH}(\bigwedge v : vars(msStack) \bullet v =!(v, msStack, \{immortal, mission\}))$$

$$\mathbf{HV3} \mathrel{\widehat=} \mathbf{SIH}\left(\begin{array}{l}\forall h : handlers \bullet (\bigwedge v : vars(hStack\, h) \bullet \\ \quad v =!(v, hStack\, h, \{immortal, mission, perR\, h\} \cup \mathrm{ran}\,(tPriv\, h)))\end{array}\right)$$

Implicitly, these conditions require that all variables v in the stacks are in the alphabet, since they are in the alphabet of the conjunctions.

We define **HV** as the composition of the functions **HV1**-**HV3**.

6 Conclusions

To the best of our knowledge, we have presented here the only formal characterisation of the SCJ memory model available so far. This is an essential ingredient to justify the soundness of assertion-based static checking techniques (like that in [17]). As a UTP theory, our model is also adequate for unification with existing models of concurrency, object orientation, and timing.

We reuse the ideas of an existing UTP model for objects and sharing [9] to address the relationship between the structure established by the references in the memory areas and the values of the programming variables and attribute accesses. What we do not cover are features of models like [11, 7]; these do not consider the issue of variable values, but provide support for reasoning about the memory graph structure. For SCJ, we will need to build on such techniques to take advantage of the separation enforced by the memory areas.

Another assertion-based technique proposed for SCJ is SafeJML [8]. It extends the well-established JML [3] to cover functionality and timing properties. The focus is on annotations that allow the use of existing technology for worst-case execution-time analysis to reason about SCJ programs.

Another contribution of this paper is a general characterisation of subset theories of designs. With this, we have given an elegant definition for the SCJ theory. Our general results are useful for all theories for programs that do not exhibit special forms of termination, and do not provide guarantees on abortion.

Our model does not capture the flow of control of an SCJ program, as partially depicted in Figure 1. This is the subject of ongoing work, which formalises the SCJ programming model in *Circus* [14], a refinement language based on Z and CSP. The semantic model of *Circus* is based on the UTP, and it is our plan to use the theory presented here as basis for the design of an extension of *Circus* that is appropriate to reason about SCJ programs. The intended model of a complete SCJ program will a predicate of the stateless CSP theory, just like that of a complete *Circus* program. So, it will have the form shown below, where the alphabet variables representing the memory structure are local.

$$\mathbf{var}\ immortal, mission \ldots;\ P;\ \mathbf{end}\ immortal, mission \ldots$$

In this case, P will be a predicate in the theory resulting from the embedding of the SCJ model presented here in the *Circus* theory of reactive designs. In the

long run, we plan to provide a reasoning framework for SCJ programs that can cater for concurrency, object-orientation, time, and sharing.

Acknowledgements This work is funded by EPSRC (grant EP/H017461/1) and UKIERI (grant SAO8-047).

References

1. J. Barnes. *High Integrity Software: The SPARK Approach to Safety and Security*. Addison-Wesley, 2003.
2. J. Barnes. *Programming in Ada 95*. Addison-Wesley, 2005.
3. L. Burdy et al.. An overview of JML tools and applications. *Software Tools for Technology Transfer*, 7(3):212 – 232, 2005.
4. A. Burns. The Ravenscar Profile. *Ada Letters*, XIX:49 – 52, 1999.
5. A. Butterfield, A. Sherif, and J. C. P. Woodcock. Slotted Circus: A UTP-family of reactive theories. In *ICFEM*, volume 4591 of *LNCS*, pages 75 – 97. Springer, 2007.
6. A. L. C. Cavalcanti, A. Wellings, and J. C. P. Woodcock. The Safety-Critical Java Mission Model: a formal account – Extended Version. Technical report, 2011. Available at `www-users.cs.york.ac.uk/ alcc/CWW11b.pdf`.
7. Y. Chen and J. Sanders. Compositional Reasoning for Pointer Structures. In *MPC*, volume 4014 of *LNCS*, pages 115 – 139. Springer, 2006.
8. G. Haddad, F. Hussain, and G. T. Leavens. The Design of SafeJML, A Specification Language for SCJ with Support for WCET Specification. In *JTRES*. ACM, 2010.
9. W. Harwood, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Theory of Pointers for the UTP. In *ICTAC*, volume 5160 of *LNCS*, pages 141 – 155. Springer, 2008.
10. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
11. C. A. R. Hoare and He Jifeng. A trace model for pointers and objects. *Programming methodology*, pages 223 – 245, 2003.
12. He Jifeng. UTP semantics for web services. In *IFM*, volume 4591 of *LNCS*, pages 353 – 372. Springer-Verlag, 2007.
13. D. Locke and et al. *Safety Critical Java Specification*. The Open Group, UK, 2010. Available at `jcp.org/aboutJava/communityprocess/edr/jsr302/index.html`.
14. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3 – 32, 2009.
15. T. L. V. L. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. Object Orientation in the UTP. In *UTP*, volume 4010 of *LNCS*, pages 18 – 37. Springer-Verlag, 2006.
16. A. Sherif, et al A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153 – 191, 2010.
17. D. Tang, A. Plsek, and J. Vitek. Static Checking of Safety Critical Java Annotations. In *JTRES*. ACM, 2010.
18. M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.
19. A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.
20. A. Wellings and M. Kim. Asynchronous event handling and safety critical Java. In *JTRES*. ACM.