DOI 10.1007/s00165-012-0253-4
BCS © 2012
Formal Aspects of Computing

Formal Aspects of Computing

The Safety-Critical Java memory model formalised

Ana Cavalcanti, Andy Wellings and Jim Woodcock
Department of Computer Science, University of York, Deramore Lane, York YO10 5GH, UK

To Carroll Morgan, on his 60th birthday

Abstract. Safety-Critical Java (SCJ) is a version of Java for real-time programming, restricted to facilitate certification of implementations of safety-critical systems. Its development is the result of an international effort involving experts from 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 Hoare and He's unifying theories of programming (UTP), enabling the integration of our theory with refinement models for object orientation and concurrency. In developing the SCJ theory, we also make a contribution to UTP by providing a general theory of invariants (an instance of which is used in the SCJ theory). The results presented here are a first essential ingredient to formalise the novel programming paradigm embedded in SCJ, and enable the justification and development of formal reasoning techniques based on refinement.

Keywords: Safety-Critical Java; Memory safety; Semantics; Unifying theories of programming; Integration; Refinement

1. Introduction

Modern society is almost totally reliant on software-based infrastructure. This demands modelling and programming languages and techniques that facilitate and ensure quality. Subsets of two languages have dominated high-integrity real-time engineering. The first is Ada [Bar05], which provides good support through its Spark [Bar03] and Ravenscar subsets [Bur99] and the Spark Examiner Toolset, but which has a limited community. The second is C/C++ (see, for example [Hat95, MIS07]), which jointly have a much larger community, but whose safer subsets 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 community effort has produced a high-integrity real-time version of Java: Safety-Critical Java (SCJ) [SCJDraft]. 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.

Correspondence and offprint requests to: J. Woodcock, E-mail: jim.woodcock@york.ac.uk

Published online: 29 June 2012

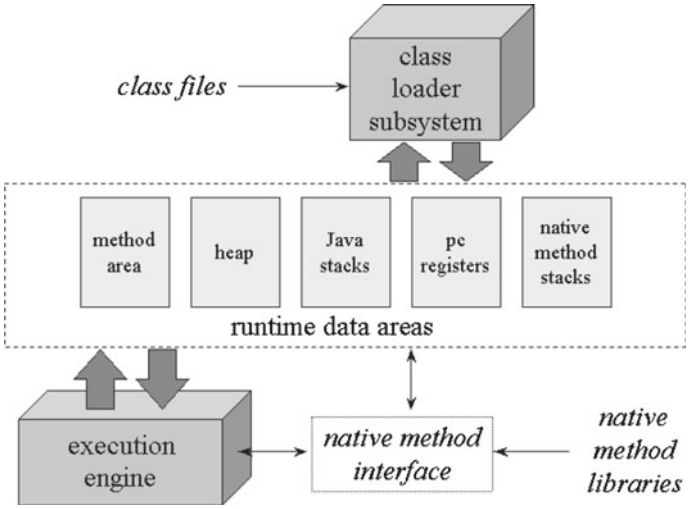

Fig. 1. The internal architecture of the Java virtual machine (taken from [Ven07, Chapter 5])

SCJ is based on a subset of Java augmented by the real-time specification for Java (RTSJ) [Wel04]. 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, and tools and techniques are needed to ensure efficient use of memory and absence of run-time errors. In order to understand the full implications of the SCJ memory model, it is necessary to appreciate the run-time data structures that are maintained by a Java virtual machine—Fig. 1 shows its internal architecture. In the context of this paper, our main concern is with the heap and the programming stacks.

All objects are placed on the heap, which is traditionally scanned by a garbage collector to remove any that are unused, and so unreachable. Each thread of control has an associated stack that is used to store variables that are local to methods. 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 [Ven07] called *memory areas*. As with the Java heap, these regions are used to store dynamically created objects; variables local to methods are still managed by the stacks associated with each thread of control.

SCJ restricts the RTSJ memory model to prohibit use of the heap. The RTSJ and SCJ both introduce two new kinds of memory area: a collection of scoped memories and a single 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. A program can have many scoped memory areas and their use can be nested. In contrast, each program has only a single instance of immortal memory, and objects created there have a lifespan equal to that of the program. Statically declared objects and fields are created in immortal memory, as are objects created by static initialisers.

Currently, SCJ includes annotations that can be used to document programs, and enable static verification of properties including memory safety (the avoidance of dangling references). The work in [TPV10] presents rules for use of the annotations, and a tool that checks statically that these rules are followed. It is not at all trivial to convince ourselves that the rules proposed and enforced by the tool achieve the level of memory safety claimed.

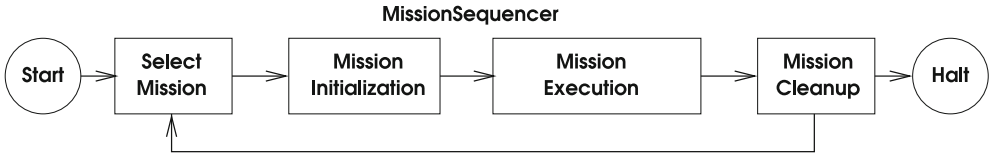

Fig. 2. Safety critical mission phases (taken from [SCJDraft])

While we do not necessarily expect to find any problems, the formalisation of the memory model (and of other aspects of the SCJ programming 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 [WK11].) 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) [HH98]. 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 [SCS06] and time [SCJS10]. By casting the SCJ memory model in UTP, we pave the way for its integration with these theories, that cater for other, also very important, aspects of an SCJ program.

In the next section, we present informally the SCJ memory model; an introduction to UTP is provided in Sect. 3. Next, Sect. 4 presents a general UTP theory for program invariants. In Sect. 5, we instantiate those results to formalise the SCJ memory model. We draw our conclusions in Sect. 8.

2. Safety-Critical Java memory model

SCJ recognises that safety-critical software varies considerably in complexity. 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 (0–2) 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 Fig. 2). 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 (ASEHs), which can be considered as being equivalent to real-time threads. Both periodic and aperiodic event handlers are supported: each handler 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 the disciplined use of 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) that can result in a violation of safe memory accesses due to dangling references, and (b) the complexity of automatic memory deallocation schemes based on garbage collection (and resulting difficulty regarding certification).

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 Fig. 3. Every thread of control in an SCJ program has a default memory allocation context: the memory area in which objects created by the allocator (the new operation) 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 application's mission sequencer and any objects that should exist throughout the lifetime of the program, and places them in immortal memory.

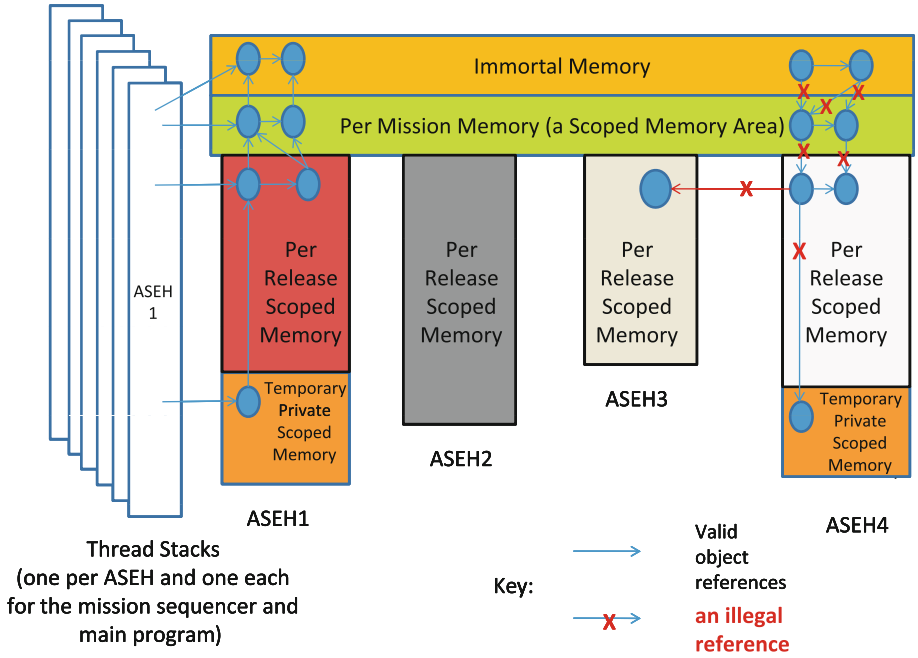

Fig. 3. SCJ memory model

The mission sequencer's thread of control is then started with immortal memory as its default allocation context. It is responsible for creating the mission memory, a scoped memory 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 of control performs the mission initialisation. The ASEHs are created in mission memory during the mission initialisation phase. Mission memory is cleared at the end of each mission, so that the memory can be used for the next mission. Any objects that must remain across missions must be stored in immortal memory.

Each ASEH has an associated private "per-release" memory area, and this is the default memory allocation context every time the ASEH is released. 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 Fig. 3 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.
- A stack of temporary private scoped memory area for each ASEH—accessible only by the associated ASEH.

Hence, each ASEH can access the immortal and mission memory areas, and its associated per-release and private memory areas. The mission sequencer can access only the mission and the immortal memory, and the main program can access only the immortal memory. The thread stacks associated with the ASEHs, the mission sequencer, and the main program can hold only references to objects in the memory areas that they are able to access.

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. As already said, an assertion-based technique is presented in [TPV10].

Immortal Memory
Objects created are never deleted.
Objects can reference other objects only in immortal memory.
Mission Memory
A form of scoped memory.
Objects created are automatically destroyed at the end of the associated mission.
Objects created can access other objects only in the same mission memory or in immortal memory.
Per-release Scoped Memory
A private memory for each ASEH.
The objects allocated are destroyed at the end of each release (period for periodic ASEHs).
The memory is reused when the next release occurs.
An object created can access objects in the current mission memory, immortal memory, and its own per-release memory.
Private Temporary Scoped Memory
Every ASEH can create one or more temporary scoped memory areas during its release.
They must be explicitly entered and exited.
An object created can reference its mission memory, immortal memory, and per-release scoped memory.
It can reference all of its outer-level nested private areas.
The objects are destroyed when the memory area is exited.
References in objects at each level are permitted to target only objects at that level and higher levels, never lower levels.

Fig. 4. Summary of SCJ memory model

Figure 3 illustrates the SCJ memory model; Fig. 4 summarises the SCJ memory areas and their properties; and Sect. 5 formalises this model in UTP.

3. Unifying theories of programming

In UTP, relations are defined by predicates over an alphabet: a set of the names of observational variables that record information about the behaviour of a program. In the theory of general relations, these include the vector of 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 their alphabet and by healthiness conditions defined by monotonic idempotent functions on predicates. The theory contains all the predicates on the alphabet that are fixed points of the healthiness conditions. As an example, we consider designs.

The general theory of relations in UTP cannot distinguish between terminating and nonterminating programs. The distinction is achieved in the subtheory 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 are defined by the following fixed-point equations.

H1 $\quad P = ok \Rightarrow P$

H2 $\quad P = P ; J \qquad$ ***where*** $J \;\widehat{=}\; (ok \Rightarrow ok') \wedge v' = v$

Here the alphabet of P is $\{ok, ok', v, v'\}$. 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 $ok \wedge P \Rightarrow ok' \wedge Q$. The set of relations constructed in this way is exactly the set fixed points of $\mathbf{H}$. Moreover, 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.

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

Sequence is relational composition.

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

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 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 over a given alphabet forms 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 Π, which is defined as $(\mathbf{true} \vdash v' = v)$, where v is the list of programming variables in the alphabet. Interestingly, Π 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 under 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 UTP

In [HH98], 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; for instance, the history of previous interactions is not affected. In [CHW06, HCW08], we take a similar approach in a theory for objects and sharing as available in Java. Our theory captures essential physical properties of sharing; for instance, variables that share a memory 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. So we present our theory as a subset of the theory of designs. Other examples of design subtheories are presented in the line of work established in [He07], 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 UTP to define theories for particular programming features, and combine them to capture more complex paradigms. We could conceivably treat the two issues of the memory structure of SCJ programs and termination separately. We would then characterise a subtheory of relations using a healthiness condition $\mathbf{HSCJ}$, for instance, and then use the designs' healthiness condition $\mathbf{H}$ to embed it in the theory of designs.

This is, however, not adequate. The design characterised by $\mathbf{H}(P)$, if P is an $\mathbf{HSCJ}$-healthy predicate that does not have ok and ok' in its alphabet, is $\neg\ P \vdash \mathbf{false}$. Its precondition considers the possibility of $\mathbf{HSCJ}$ not holding (even if we are not in an 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, a situation from which a program cannot recover.

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 broken only by nontermination. We use these general results later to characterise our theory for the SCJ memory model.

4.1. Operation invariants

For an operation invariant defined by a predicate Ψ (a binary relation on before and after states), the subtheory of designs that satisfy this invariant is characterised by the healthiness condition $\mathbf{OIH}$(operation-invariant healthiness).

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

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

Theorem 4.1 $\mathbf{OIH}(\Psi)$ *is a monotonic idempotent function on designs.*

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

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

Since $ok \land \neg\, (\neg\, D^f \vdash D^t \land \Psi)^f = ok \land \neg\, D^f$, 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, for every v, $\Psi[v/v']$, we have that $\mathit{II}_{OI}(\Psi)$ is the sequence left unit.

Theorem 4.2 *If* Ψ *is reflexive,* $\mathit{II}_{OI}(\Psi)\ ;\ D = D$*, for every* $\mathbf{OIH}(\Psi)$*-healthy* D.

Proof.

$$\begin{array}{ll}
& \mathit{II}_{OI}(\Psi)\ ;\ D \\
= & \mathbf{OIH}(\Psi)(\mathit{II})\ ;\ \mathbf{OIH}(\Psi)(D) & \text{[definition of } \mathit{II}_{OI} \text{ and } D \text{ is } \mathbf{OIH}(\Psi)\text{-healthy]} \\
= & (\mathbf{true} \vdash v' = v \land \Psi)\ ;\ (\neg\, D^f \vdash D^t \land \Psi) & \text{[Theorem 4.1]} \\
= & \neg\, (v' = v \land \Psi\ ;\ D^f) \vdash v' = v \land \Psi\ ;\ D^t \land \Psi & \text{[design sequence closure]} \\
= & \neg\, (\Psi[v/v'] \land D^f) \vdash \Psi[v/v'] \land D^t \land \Psi & \text{[sequence one-point rule]} \\
= & \neg\, D^f \vdash D^t \land \Psi & \text{[}\Psi \text{ reflexive]} \\
= & D & \text{[definition of design, Theorem 4.1, and D is } \mathbf{OIH}(\Psi)\text{-healthy]}
\end{array}$$

□

$\mathit{II}_{OI}(\Psi)$ is not necessarily the right unit. As in the theory of general designs, this requires that the precondition refers to no dashed variables.

Lemma 4.1 *If* Ψ *is reflexive,* $D\ ;\ \mathit{II}_{OI}(\Psi) = (\neg\, \exists\, v' \bullet D^f \vdash D^t \land \Psi)$.

Proof.

$$\begin{array}{ll}
& D\ ;\ \mathit{II}_{OI}(\Psi) \\
= & \mathbf{OIH}(\Psi)(D)\ ;\ \mathbf{OIH}(\Psi)(\mathit{II}) & \text{[definition of } \mathit{II}_{OI} \text{ and } D \text{ is } \mathbf{OIH}(\Psi)\text{-healthy]} \\
= & (\neg\, D^f \vdash D^t \land \Psi)\ ;\ (\mathbf{true} \vdash v' = v \land \Psi) & \text{[Theorem 4.1]} \\
= & \neg\, (D^f\ ;\ \mathbf{true}) \vdash D^t \land \Psi\ ;\ v' = v \land \Psi & \text{[sequence design closure]} \\
= & \neg\, \exists\, v' \bullet D^f \vdash D^t \land \Psi\ ;\ v' = v \land \Psi & \text{[relational calculus]} \\
= & \neg\, \exists\, v' \bullet D^f \vdash D^t \land \Psi \land \Psi[v'/v] & \text{[sequence one-point rule]} \\
= & \neg\, \exists\, v' \bullet D^f \vdash D^t \land \Psi & \text{[}\Psi \text{ is reflexive]}
\end{array}$$

□

$\mathbf{OIH}(\Psi)$ is closed with respect to conjunction, disjunction (which models nondeterminism) and conditional. These are established by the following theorems.

Theorem 4.3 $\mathbf{OIH}(\Psi)$ *is closed with respect to conjunction.*

Proof.

$$
\begin{array}{ll}
& D_1 \wedge D_2 \\
= & \mathbf{OIH}(\Psi)(D_1) \wedge \mathbf{OIH}(\Psi)(D_2) & [\text{by } D_1 \text{ and } D_2 \text{ are } \mathbf{OIH}(\Psi)\text{-healthy}] \\
= & D_1 \wedge (ok \wedge \neg\, D_1^f \Rightarrow \Psi) \wedge D_2 \wedge (ok \wedge \neg\, D_2^f \Rightarrow \Psi) & [\text{definition of } \mathbf{OIH}(\Psi)] \\
= & D_1 \wedge D_2 \wedge ((ok \wedge \neg\, D_1^f) \vee (ok \wedge \neg\, D_2^f) \Rightarrow \Psi) & [\text{propositional calculus}] \\
= & D_1 \wedge D_2 \wedge (ok \wedge \neg\, (D_1 \wedge D_2)^f \Rightarrow \Psi) & [\text{propositional calculus}] \\
= & \mathbf{OIH}(\Psi)(D_1 \wedge D_2) & [\text{definition of } \mathbf{OIH}(\Psi)]
\end{array}
$$

□

Theorem 4.4 $\mathbf{OIH}(\Psi)$ *is closed with respect to disjunction.*

Proof.

$$
\begin{array}{ll}
& D_1 \vee D_2 \\
= & \mathbf{OIH}(\Psi)(D_1) \vee \mathbf{OIH}(\Psi)(D_2) & [\text{Theorem 4.1}] \\
= & (\neg\, D_1^f \vdash D_1^t \wedge \Psi) \vee (\neg\, D_2^f \vdash D_2^t \wedge \Psi) & [\text{definition of } \mathbf{OIH}(\Psi)] \\
= & (\neg\, D_1^f \wedge \neg\, D_2^f \vdash (D_1^t \wedge \Psi) \vee (D_2^t \wedge \Psi)) & [\text{design disjunction closure}] \\
= & (\neg\, (D_1^f \vee \neg\, D_2^f) \vdash (D_1^t \vee D_2^t) \wedge \Psi) & [\text{propositional calculus}] \\
= & \mathbf{OIH}(\Psi)(D_1 \vee D_2) & [\text{Theorem 4.1}]
\end{array}
$$

□

Theorem 4.5 $\mathbf{OIH}(\Psi)$ *is closed with respect to conditional.*

Proof.

$$
\begin{array}{ll}
& D_1 \lhd b \rhd D_2 \\
= & \mathbf{OIH}(\Psi)(D_1) \lhd b \rhd \mathbf{OIH}(\Psi)(D_2) & [D_1 \text{ and } D_2 \text{ are } \mathbf{OIH}(\Psi)\text{-healthy}] \\
= & (\neg\, D_1^f \vdash D_1^t \wedge \Psi) \lhd b \rhd (\neg\, D_2^f \vdash D_2^t \wedge \Psi) & [\text{Theorem 4.1}] \\
= & (\neg\, D_1^f \lhd b \rhd \neg\, D_2^f) \vdash (D_1^t \wedge \Psi \lhd b \rhd D_2^t \wedge \Psi) & [\text{design conditional closure}] \\
= & \neg\, (D_1^f \lhd b \rhd D_2^f) \vdash (D_1^t \wedge \Psi \lhd b \rhd D_2^t \wedge \Psi) & [\text{conditional negation}] \\
= & \neg\, (D_1^f \lhd b \rhd D_2^f) \vdash (D_1^t \lhd b \rhd D_2^t) \wedge \Psi & [\text{conditional conjunction}] \\
= & \neg\, (D_1^f \lhd b^f \rhd D_2^f) \vdash (D_1^t \lhd b^t \rhd D_2^t) \wedge \Psi & [ok' \text{ not free in } b] \\
= & \neg\, (D_1 \lhd b \rhd D_2)^f \vdash (D_1 \lhd b \rhd D_2)^t \wedge \Psi & [\text{substitution}] \\
= & \mathbf{OIH}(\Psi)(D_1 \lhd b \rhd D_2) & [\text{Theorem 4.1}]
\end{array}
$$

□

For $\mathbf{OIH}(\Psi)$ to be closed with respect to sequence, we need Ψ to be transitive, that is, $(\Psi\ ;\ \Psi) \Rightarrow \Psi$. To prove that, we need the lemma below; it confirms that the postcondition of $P \vdash Q$ is indeed Q.

Lemma 4.2 $(P \vdash Q)^t = Q$, *provided* $ok \wedge P$ *and* ok' *is not free in* P.

Proof.

$$
\begin{array}{ll}
& (P \vdash Q)^t \\
= & (ok \wedge P \Rightarrow ok' \wedge Q)^t & [\text{definition of designs}] \\
= & ok \wedge P \Rightarrow Q & [ok' \text{ is not free in } P \text{ and propositional calculus}] \\
& Q & [ok \wedge P \text{ and propositional calculus}]
\end{array}
$$

□

Finally, the theorem below establishes closure with respect to sequence.

Theorem 4.6 *If D_1 and D_2 are **OIH**(Ψ)-healthy designs, and Ψ is transitive, then $D_1 \,;\, D_2$ is **OIH**(Ψ)-healthy.*

Proof.

$$
\begin{array}{ll}
& D_1 \,;\, D_2 \\
= & \neg\, (D_1 \,;\, D_2)^f \vdash (D_1 \,;\, D_2)^t & \text{[design property]} \\
= & \neg\, (D_1 \,;\, D_2)^f \vdash D_1^t \,;\, D_2^t & \text{[design sequence postcondition]} \\
= & \neg\, (D_1 \,;\, D_2)^f \vdash D_1^t \wedge \Psi \,;\, D_2^t \wedge \Psi & [D_1, D_2 \text{ are } \textbf{\textit{OIH}}\text{-healthy}] \\
= & \neg\, (D_1 \,;\, D_2)^f \vdash (D_1^t \wedge \Psi \,;\, D_2^t \wedge \Psi) \wedge \Psi & [\Psi \text{ transitive}] \\
= & \neg\, (D_1 \,;\, D_2)^f \vdash (D_1^t \,;\, D_2^t) \wedge \Psi & [D_1, D_2 \text{ are } \textbf{\textit{OIH}}\text{-healthy}] \\
= & \neg\, (D_1 \,;\, D_2)^f \vdash (D_1 \,;\, D_2)^t \wedge \Psi & \text{[design sequence postcondition]} \\
= & \textbf{\textit{OIH}}(D_1 \,;\, D_2) & \text{[Theorem 4.1]}
\end{array}
$$

□

The set of **OIH**(Ψ)-healthy designs is a complete lattice, since it is the image of a monotonic idempotent healthiness condition [HH98], 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 (**false** $\vdash$ **true**), which is equal to (**false** $\vdash \Psi$)) and magic (**true** $\vdash$ **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, **ISH** (input-state healthiness).

ISH(ψ) $\quad D = D \vee (ok \wedge \neg\, D^f \wedge \psi \Rightarrow ok' \wedge D^t)$

Theorem 4.7 **ISH**(ψ) *is an idempotent function on designs.*

Proof. First, we show that **ISH**(ψ)(D) is a design.

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

The arguments for idempotence and monotonicity are similar to those used in Theorem 4.1. □

We define the healthy identity $\mathit{II}_{IS}(\psi) \mathrel{\widehat{=}} \textbf{\textit{ISH}}(\psi)(\mathit{II})$. It is indeed the left-unit of sequence; the proof of this result is a simple consequence of the definitions of $\mathit{II}_{IS}(\psi)$ and sequence, and Theorem 4.7 above.

Theorem 4.8 $\mathit{II}_{IS}(\psi) \,;\, D = D$, *for every* **ISH**($\psi$)*-healthy* D.

Proof.

$$
\begin{array}{ll}
& \mathit{II}_{IS}(\psi) \,;\, D \\
= & \textbf{\textit{ISH}}(\psi)(\mathit{II}) \,;\, \textbf{\textit{ISH}}(\psi)(D) & [\text{definition of } \mathit{II}_{IS}(\psi) \text{ and } D \text{ is } \textbf{\textit{ISH}}(\psi)\text{-healthy}] \\
= & (\psi \vdash v' = v) \,;\, (\neg\, D^f \wedge \psi \vdash D^t) & \text{[Theorem 4.7]} \\
= & \neg\, (\neg\, \psi \,;\, \textbf{\textit{true}}) \wedge \neg\, ((v' = v) \,;\, \neg\, (\neg\, D^f \wedge \psi)) \vdash (v' = v) \,;\, D^t & \text{[sequence of designs]} \\
= & \psi \wedge \neg\, (D^f \vee \neg\, \psi) \vdash D^t & \text{[predicate calculus]} \\
= & \neg\, D^f \wedge \psi \vdash D^t & \text{[propositional calculus]} \\
= & \textbf{\textit{ISH}}(\psi)(D) & \text{[Theorem 4.7]} \\
& D & [D \text{ is } \textbf{\textit{ISH}}(\psi)\text{-healthy}]
\end{array}
$$

□

Again, right unit does not hold in all cases.

Theorem 4.9 *$D\ ;\ \Pi_{IS}(\psi) = \neg\,(\exists\, v' \bullet D^f) \wedge \psi \wedge \neg\,(D^t\ ;\ \neg\,\psi) \vdash D^t$, for every **ISH**($\psi$)-healthy* D.

Proof.

$$
\begin{array}{ll}
& D\ ;\ \Pi_{IS}(\psi) \\
= & \textbf{ISH}(\psi)(D)\ ;\ \textbf{ISH}(\psi)(\Pi) & \text{[definition of } \Pi_{IS}(\psi) \text{ and } D \text{ is } \textbf{ISH}(\psi)\text{-healthy]} \\
= & (\neg\, D^f \wedge \psi \vdash D^t)\ ;\ (\psi \vdash v' = v) & \text{[Theorem 4.7]} \\
= & \neg\,((D^f \vee \neg\,\psi)\ ;\ \textbf{true}) \wedge \neg\,(D^t\ ;\ \neg\,\psi) \vdash D^t\ ;\ (v' = v) & \text{[sequence of designs]} \\
= & \neg\,(\exists\, v' \bullet D^f \vee \neg\,\psi) \wedge \neg\,(D^t\ ;\ \neg\,\psi) \vdash \exists\, v_0 \bullet D^t[v_0/v'] \wedge v' = v_0 & \text{[sequence]} \\
= & \neg\,(\exists\, v' \bullet D^f) \wedge \psi \wedge \neg\,(D^t\ ;\ \neg\,\psi) \vdash D^t & \text{[predicate calculus]}
\end{array}
$$

□

So, not only does the precondition have to have no dashed variables, but the postcondition cannot break the invariant. The invariant ψ is assumed by all programs, including the identity. It aborts if the previous program does not establish ψ.

To establish the result for conjunction, we need the following lemma, which shows us that the precondition of a design $P \vdash Q$ is indeed P.

Lemma 4.3 *$ok \wedge \neg\,(P \vdash Q)^f = ok \wedge P$, provided ok' is not free in P.*

Proof.

$$
\begin{array}{ll}
& ok \wedge \neg\,(P \vdash Q)^f \\
= & ok \wedge \neg\,(ok \wedge P \Rightarrow ok' \wedge Q)^f & \text{[definition of designs]} \\
= & ok \wedge \neg\,\neg\,(ok \wedge P) & [ok' \text{ is not free in } P \text{ and propositional calculus}] \\
= & ok \wedge P & \text{[propositional calculus]}
\end{array}
$$

□

ISH(ψ) is closed with respect to conjunction, disjunction, conditional, and sequence. This is established by the following four theorems.

Theorem 4.10 ***ISH***(ψ) *is closed with respect to conjunction.*

Proof.

$$
\begin{array}{ll}
& D_1 \wedge D_2 \\
= & \textbf{ISH}(\psi)(D_1) \wedge \textbf{ISH}(\psi)(D_2) & \text{[by } D_1 \text{ and } D_2 \text{ are } \textbf{ISH}(\psi)\text{-healthy]} \\
= & (\neg\, D_1^f \wedge \psi \vdash D_1^t) \wedge (\neg\, D_2^f \wedge \psi \vdash D_2^t) & \text{[Theorem 4.7]} \\
= & ((\neg\, D_1^f \wedge \psi) \vee (\neg\, D_2^f \wedge \psi) \vdash (\neg\, D_1^f \wedge \psi \Rightarrow D_1^t) \wedge (\neg\, D_2^f \wedge \psi \Rightarrow D_2^t)) & \text{[conjunction of designs]} \\
= & ((\neg\, D_1^f \vee \neg\, D_2^f) \wedge \psi \vdash \psi \Rightarrow (\neg\, D_1^f \Rightarrow D_1^t) \wedge (\neg\, D_2^f \Rightarrow D_2^t)) & \text{[propositional calculus]} \\
= & ((\neg\, D_1^f \vee \neg\, D_2^f) \wedge \psi \vdash (\neg\, D_1^f \Rightarrow D_1^t) \wedge (\neg\, D_2^f \Rightarrow D_2^t)) & \text{[property of designs]} \\
= & \textbf{ISH}(\psi)((\neg\, D_1^f \vee \neg\, D_2^f) \vdash (\neg\, D_1^f \Rightarrow D_1^t) \wedge (\neg\, D_2^f \Rightarrow D_2^t)) & \text{[Theorem 4.7 and Lemma 4.3]} \\
= & \textbf{ISH}(\psi)(D_1 \wedge D_2) & \text{[conjunction of designs]}
\end{array}
$$

□

Theorem 4.11 ***ISH***(ψ) *is closed with respect to disjunction.*

Proof.

$$
\begin{array}{ll}
& D_1 \vee D_2 \\
= & \textbf{ISH}(\psi)(D_1) \vee \textbf{ISH}(\psi)(D_2) & \text{[by } D_1 \text{ and } D_2 \text{ are } \textbf{ISH}(\psi)\text{-healthy]} \\
= & (\neg\, D_1^f \wedge \psi \vdash D_1^t) \vee (\neg\, D_2^f \wedge \psi \vdash D_2^t) & \text{[Theorem 4.7]}
\end{array}
$$

$= (\neg\, D_1^f \wedge \neg\, D_2^f \wedge \psi \vdash D_1^t \vee D_2^t)$ [disjunction of designs]
$= \mathbf{ISH}(\psi)(\neg\, D_1^f \wedge \neg\, D_2^f \vdash D_1^t \vee D_2^t)$ [Theorem 4.7]
$= \mathbf{ISH}(\psi)(\neg\, (D_1 \vee \neg\, D_2)^f \vdash (D_1 \vee D_2)^t)$ [propositional calculus]
$= \mathbf{ISH}(\psi)(D_1 \vee D_2)$ [disjunction of designs]

□

Theorem 4.12 *$\mathbf{ISH}(\psi)$ is closed with respect to conditional.*

Proof.

$D_1 \lhd b \rhd D_2$
$= \mathbf{ISH}(\psi)(D_1) \lhd b \rhd \mathbf{ISH}(\psi)(D_2)$ [D_1 and D_2 are $\mathbf{ISH}(\Psi)$-healthy]
$= (\neg\, D_1^f \wedge \psi \vdash D_1^t) \lhd b \rhd (\neg\, D_2^f \wedge \psi \vdash D_2^t)$ [Theorem 4.7]
$= ((\neg\, D_1^f \wedge \psi) \lhd b \rhd (\neg\, D_2^f \wedge \psi) \vdash D_1^t \lhd b \rhd D_2^t)$ [conditional of designs]
$= ((\neg\, D_1^f \lhd b \rhd \neg\, D_2^f) \wedge \psi \vdash D_1^t \lhd b \rhd D_2^t)$ [propositional calculus]
$= \mathbf{ISH}(\psi)(\neg\, D_1^f \lhd b \rhd \neg\, D_2^f \vdash D_1^t \lhd b \rhd D_2^t)$ [Theorem 4.7]
$= \mathbf{ISH}(\psi)((\neg\, D_1 \lhd b \rhd \neg\, D_2)^f \vdash (D_1 \lhd b \rhd D_2)^t)$ [propositional calculus]
$= \mathbf{ISH}(\psi)(D_1 \lhd b \rhd D_2)$ [conditional of designs]

□

Theorem 4.13 *If D_1 is an $\mathbf{ISH}(\psi)$-healthy design, then $D_1 \,;\, D_2$ is $\mathbf{ISH}(\psi)$-healthy.*

Proof.

$D_1 \,;\, D_2$
$= \mathbf{ISH}(\psi)(D_1) \,;\, D_2$ [D_1 is $\mathbf{ISH}(\psi)$-healthy]
$= (\neg\, D_1^f \wedge \psi \vdash D_2^t) \,;\, (\neg\, D_2^f \vdash D_2^t)$ [Theorem 4.7]
$= \neg\, ((D_1^f \vee \neg\, \psi) \,;\, \mathbf{true}) \wedge \neg\, (D_1^t \,;\, D_2^f) \vdash D_1^t \,;\, D_2^t$ [sequence of designs]
$= \neg\, (\neg\, \psi \vee (D_1^f \,;\, \mathbf{true})) \wedge \neg\, (D_1^t \,;\, D_2^f) \vdash D_1^t \,;\, D_2^t$ [relational calculus, ψ is a condition]
$= \psi \wedge \neg\, (D_1^f \,;\, \mathbf{true}) \wedge \neg\, (D_1^t \,;\, D_2^f) \vdash D_1^t \,;\, D_2^t$ [propositional calculus]
$= \mathbf{ISH}(\psi)(\neg\, (D_1^f \,;\, \mathbf{true}) \wedge \neg\, (D_1^t \,;\, D_2^f) \vdash D_1^t \,;\, D_2^t)$ [Theorem 4.7]
$= \mathbf{ISH}(\psi)((\neg\, D_1^f \vdash D_2^t) \,;\, (\neg\, D_2^f \vdash D_2^t))$ [design sequence]
$= \mathbf{ISH}(\psi)(D_1 \,;\, D_2)$ [designs]

□

The bottom of the lattice that it defines is abort, but the top is $(\psi \vdash \mathbf{false})$. This is miraculous only when ψ holds.

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

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

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 4.14 ***ISH***(ψ) *and* ***OSH***(ψ) *commute.*

Proof.

$$
\begin{array}{ll}
& \textbf{\textit{ISH}}(\psi) \circ \textbf{\textit{OSH}}(\psi)(D) \\
= & \textbf{\textit{ISH}}(\psi) \circ \textbf{\textit{OIH}}(\psi \Rightarrow \psi')(D) & [\text{observation: } \textbf{\textit{OSH}}(\psi) = \textbf{\textit{OIH}}(\psi \Rightarrow \psi')] \\
= & \textbf{\textit{ISH}}(\psi)(\neg\, D^f \vdash D^t \wedge (\psi \Rightarrow \psi')) & [\text{Theorem 4.1}] \\
= & \neg\, D^f \wedge \psi \vdash D^t \wedge (\psi \Rightarrow \psi') & [\text{Theorem 4.7}] \\
= & \neg\, D^f \wedge \psi \vdash D^t \wedge \psi' & [\text{property of designs}] \\
= & \textbf{\textit{OSH}}(\psi)(\neg\, D^f \wedge \psi \vdash D^t) & [\text{Theorem 4.1}] \\
= & \textbf{\textit{OSH}}(\psi) \circ \textbf{\textit{ISH}}(\psi)(D) & [\text{Theorem 4.7}]
\end{array}
$$

□

As shown above, an ***ISH***(ψ) and ***OSH***(ψ)-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 ***ISH***(ψ) and ***OSH***(ψ) are idempotent, by Theorem 4.14, so is ***SIH***$(\psi) \;\widehat{=}\; \textbf{\textit{ISH}}(\psi) \circ \textbf{\textit{OSH}}(\psi)$ [HH98]; 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*** $\circ$ ***C2***)-healthy is also ***C1*** and ***C2***-healthy.

Theorem 4.15 *If* ***C1*** $\circ$ ***C2***$(P) = P$*, then* ***C1***$(P) = P$ *and* ***C2***$(P) = P$*, provided* ***C1*** *and* ***C2*** *commute.*

Proof.

C1

$$
\begin{array}{ll}
& \textbf{\textit{C1}}(P) \\
= & \textbf{\textit{C1}}(\textbf{\textit{C1}}(\textbf{\textit{C2}}(P))) & [\text{P is } \textbf{\textit{C1}} \circ \textbf{\textit{C2}}\text{-healthy}] \\
= & \textbf{\textit{C1}}(\textbf{\textit{C2}}(P)) & [\textbf{\textit{C1}} \text{ is idempotent}] \\
= & P & [\text{P is } \textbf{\textit{C1}} \circ \textbf{\textit{C2}}\text{-healthy}]
\end{array}
$$

C2

$$
\begin{array}{ll}
& \textbf{\textit{C2}}(P) \\
= & \textbf{\textit{C2}}(\textbf{\textit{C1}}(\textbf{\textit{C2}}(P))) & [\text{P is } \textbf{\textit{C1}} \circ \textbf{\textit{C2}}\text{-healthy}] \\
= & \textbf{\textit{C2}}(\textbf{\textit{C2}}(\textbf{\textit{C1}}(P))) & [\textbf{\textit{C1}} \circ \textbf{\textit{C2}} = \textbf{\textit{C2}} \circ \textbf{\textit{C1}}] \\
= & \textbf{\textit{C2}}(\textbf{\textit{C1}}(P)) & [\textbf{\textit{C2}} \text{ is idempotent}] \\
= & P & [\text{P is } \textbf{\textit{C1}} \circ \textbf{\textit{C2}}\text{-healthy and } \textbf{\textit{C1}} \circ \textbf{\textit{C2}} = \textbf{\textit{C2}} \circ \textbf{\textit{C1}}]
\end{array}
$$

□

From this and the theorems above and in Sect. 4.1, we can conclude that ***SIH***(ψ) distributes through conjunction, disjunction, and conditional.

Theorem 4.16 *For any operator* **op***, and healthiness conditions* ***C1*** *and* ***C2*** *closed with respect to* **op***, if they commute, then* ***C1*** $\circ$ ***C2*** *is closed as well.*

Proof.

$$
\begin{array}{ll}
& P \;\mathbf{op}\; Q \\
= & \textbf{\textit{C1}}(P \;\mathbf{op}\; Q) & [P \text{ and } Q \text{ are } \textbf{\textit{C1}}\text{-healthy (Theorem 4.15) and } \textbf{\textit{C1}} \text{ is closed with respect to } \mathbf{op}] \\
= & \textbf{\textit{C1}}(\textbf{\textit{C2}}(P \;\mathbf{op}\; Q)) & [P \text{ and } Q \text{ are } \textbf{\textit{C2}}\text{-healthy (Theorem 4.15) and } \textbf{\textit{C2}} \text{ is closed with respect to } \mathbf{op}] \\
= & \textbf{\textit{C1}} \circ \textbf{\textit{C2}}(P \;\mathbf{op}\; Q) & [\text{function composition}]
\end{array}
$$

□

For operation and state invariants Ψ_1 and ψ_2, ***OIH***(Ψ_1) and ***SIH***(ψ_2) commute.

Theorem 4.17 $\mathbf{OIH}(\Psi_1)$ *and* $\mathbf{SIH}(\psi_2)$ *commute.*

Proof.

$$
\begin{array}{ll}
& \mathbf{OIH}(\Psi_1) \circ \mathbf{SIH}(\psi_2)(D) \\
= & \mathbf{OIH}(\Psi_1)(\neg\, D^f \wedge \psi_2 \vdash D^t \wedge \psi_2') & \text{[Theorem 4.14]} \\
= & \neg\, D^f \wedge \psi_2 \vdash D^t \wedge \psi_2' \wedge \Psi_1 & \text{[Theorem 4.1 and Lemmas 4.3 and 4.2]} \\
= & \mathbf{SIH}(\psi_2)(\neg\, D^f \vdash D^t \wedge \Psi_1) & \text{[Theorem 4.14 and Lemmas 4.3 and 4.2]} \\
= & \mathbf{SIH}(\Psi_2) \circ \mathbf{OIH}(\psi_1)(D) & \text{[Theorem 4.1]}
\end{array}
$$

□

So, using an argument similar to that above, we can conclude that a theory characterised by

$$\mathbf{IH}(\Psi_1, \psi_2) \;\widehat{=}\; \mathbf{OIH}(\Psi_1) \circ \mathbf{SIH}(\psi_2)$$

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.

The identity of a $\mathbf{IH}(\Psi_1, \psi_2)$ theory is $\text{Ⅱ}_{IH}(\Psi_1, \psi_2) \;\widehat{=}\; \mathbf{IH}(\Psi_1, \psi_2)(\text{Ⅱ})$. It is the left unit of sequence, but not the right unit.

Theorem 4.18 *If* Ψ_1 *is reflexive,* $\text{Ⅱ}_{IH}(\Psi_1, \psi_2)\,;\, D = D$*, for every* $\mathbf{IH}(\Psi_1, \psi_2)$*-healthy* D.

Proof.

$$
\begin{array}{ll}
& \text{Ⅱ}_{IH}(\Psi_1, \psi_2)\,;\, D \\
= & \mathbf{IH}(\Psi_1, \psi_2)(\text{Ⅱ})\,;\, \mathbf{IH}(\Psi_1, \psi_2)(D) & \text{[definition of } \mathbf{IH}(\Psi_1, \psi_2)(\text{Ⅱ}), D \text{ is } \mathbf{IH}(\Psi_1, \psi_2)\text{-healthy]} \\
= & (\psi_2 \vdash v' = v \wedge \psi_2' \wedge \Psi_1)\,;\, (\neg\, D^f \wedge \psi_2 \vdash D^t \wedge \psi_2' \wedge \Psi_1) & \text{[Theorem 4.17]} \\
= & \begin{pmatrix} \psi_2 \wedge \neg\, ((v' = v \wedge \psi_2' \wedge \Psi_1)\,;\, (D^f \vee \neg\, \psi_2)) \\ \vdash \\ (v' = v \wedge \psi_2' \wedge \Psi_1)\,;\, (D^t \wedge \psi_2' \wedge \Psi_1) \end{pmatrix} & \text{[sequence of designs]} \\
= & \begin{pmatrix} \psi_2 \wedge \neg\, (\psi_2 \wedge \Psi_1[v/v'] \wedge (D^f \vee \neg\, \psi_2)) \\ \vdash \\ \psi_2 \wedge \Psi_1[v/v'] \wedge D^t \wedge \psi_2' \wedge \Psi_1 \end{pmatrix} & \text{[sequence one-point rule, twice]} \\
= & \psi_2 \wedge \neg\, (\psi_2 \wedge (D^f \vee \neg\, \psi_2)) \vdash \psi_2 \wedge D^t \wedge \psi_2' \wedge \Psi_1 & [\Psi_1 \text{ is reflexive}] \\
= & \psi_2 \wedge \neg\, D^f \vdash D^t \wedge \psi_2' \wedge \Psi_1 & \text{[propositional calculus and designs]} \\
= & \mathbf{IH}(\Psi_1, \psi_2)(D) & \text{[Theorem 4.17]} \\
= & D & [D \text{ is } \mathbf{IH}(\Psi_1, \psi_2)\text{-healthy}]
\end{array}
$$

□

Again, right unit does not hold in all cases.

Theorem 4.19 $D\,;\, \text{Ⅱ}_{IH}(\Psi_1, \psi_2) = \neg\, (\exists\, v' \bullet D^f) \wedge \psi \wedge \neg\, (D^t\,;\, \neg\, \psi) \vdash D^t$*, for every* $\mathbf{IH}(\Psi_1, \psi_2)$*-healthy* D.

Proof.

$$
\begin{array}{ll}
& D\,;\, \text{Ⅱ}_{IH}(\Psi_1, \psi_2) \\
= & \mathbf{IH}(\Psi_1, \psi_2)(D)\,;\, \mathbf{IH}(\Psi_1, \psi_2)(\text{Ⅱ}) \\
& & \text{[definition of } \text{Ⅱ}_{IH}(\Psi_1, \psi_2) \text{ and } D \text{ is } \mathbf{IH}(\Psi_1, \psi_2)\text{-healthy]} \\
= & (\neg\, D^f \wedge \psi_2 \vdash D^t \wedge \psi_2' \wedge \Psi_1)\,;\, (\psi_2 \vdash v' = v \wedge \psi_2' \wedge \Psi_1) & \text{[Theorem 4.17]} \\
= & \begin{pmatrix} \neg\, ((D^f \vee \neg\, \psi_2)\,;\, \mathbf{true}) \wedge \neg\, ((D^t \wedge \psi_2' \wedge \Psi_1)\,;\, \neg\, \psi_2) \\ \vdash \\ (D^t \wedge \psi_2' \wedge \Psi_1)\,;\, (v' = v \wedge \psi_2' \wedge \Psi_1) \end{pmatrix} \\
& & \text{[sequence of designs]}
\end{array}
$$

$$= \begin{pmatrix} \neg\,(\exists\, v' \bullet D^f \lor \neg\, \psi_2) \land \neg\,(\exists\, v' \bullet D^t \land \psi_2' \land \Psi_1 \land \neg\, \psi_2') \\ \vdash \\ \exists\, v_0 \bullet D^t[v_0/v'] \land \psi_2'[v_0/v'] \land \Psi_1[v_0/v'] \land v' = v_0 \land \psi_2' \land \Psi_1[v_0/v] \end{pmatrix}$$

[sequence]

$$= \neg\,(\exists\, v' \bullet D^f \lor \neg\, \psi_2) \vdash D^t \land \psi_2' \land \Psi_1 \land \Psi_1[v'/v] \qquad \text{[predicate calculus]}$$

$$= \neg\,(\exists\, v' \bullet D^f) \land \psi_2 \vdash D^t \land \psi_2' \land \Psi_1 \qquad [\Psi_1 \text{ is reflexive}]$$

□

The bottom of the lattice is abort, and the top is ($\psi_2 \vdash \textbf{\textit{false}}$).

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 execution model of the missions, releases of the handlers, and the creation of private temporary memory areas. This separation of concerns simplifies the presentation of our theory, and allows the treatment of various aspects of the model independently.

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 just an element of the unspecified set $VName$, and of a reference, from the 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; for simplicity, we do not capture issues related to size, although this could be done by extending our model. 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, $refsRes\ ma = \mathrm{dom}\, ma$, and $refsIn\ ma = \bigcup(\mathrm{ran}(\!|\ (\_ \rhd Ref)(\!|\ \mathrm{ran}\, ma\ |\!)\ |\!))$. For a memory area ma (or more precisely, for the contents ma of a memory area), $\mathrm{ran}\, ma$ gives its objects. By using relational image $\_(\!|\ \_\ |\!)$ to apply the operator $(\_ \rhd Ref)$ 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 also ten 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

$$hStack : handlers \to \mathrm{stack}\ Frame$$

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 $perR : handlers \to MAreaC$. The temporary private memory areas are organised in a stack as recorded in the alphabet variable $tPriv : handlers \to \mathrm{stack}\ MAreaC$. 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 $profile : MAreaC \to (Ref \pfun \mathbb{F}\, VName)$. 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 $profile\ ma = \{r : \mathrm{dom}\, ma \bullet r \mapsto \mathrm{dom}(ma\ r)\}$. 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.

$$\textbf{\textit{HSCJ1}} \mathrel{\widehat=} \textbf{\textit{OIH}}(profile\ immortal \subseteq profile\ immortal')$$

The operation invariant for ***HSCJ1*** is reflexive and transitive, because $\subseteq$ is.

The references in the program stack can target only objects in the immortal memory. This is specified by the healthiness condition ***HSCJ2***, which uses a lifted version of $refsIn : \mathrm{stack}\ Frame \to \mathbb{F}\, Ref$ 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.

$$\textbf{\textit{HSCJ2}} \mathrel{\widehat=} \textbf{\textit{SIH}}(refsIn\ pStack \subseteq refsRes\ immortal)$$

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

$$\textbf{\textit{HSCJ3}} \mathrel{\widehat=} \textbf{\textit{SIH}}(refsIn\ immortal \subseteq refsRes\ immortal)$$

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.

$$\textbf{\textit{HSCJ4}} \mathrel{\widehat=} \textbf{\textit{SIH}}(refsIn\ msStack \subseteq refsRes\ \{immortal, mission\})$$

$$\textbf{\textit{HSCJ5}} \mathrel{\widehat=} \textbf{\textit{SIH}}(refsIn\ mission \subseteq refsRes\ \{immortal, mission\})$$

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.

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

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.

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

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.

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

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

As already said, private temporary memory areas are created by calls to a method `create`. For every handler, therefore, the size of its frame stack is greater than the size of the stack of its temporary private memory areas.

$$\textbf{\textit{HSCJ9}} \mathrel{\widehat=} \textbf{\textit{SIH}}(\forall h : handlers \bullet \#(hStack\ h) > \#(tPriv\ h))$$

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

$$\textbf{\textit{HSCJ10}} \mathrel{\widehat=} \textbf{\textit{SIH}}(\mathsf{disjoint}\ \langle \mathit{refsRes\ immortal}, \mathit{refsRes\ mission}\rangle \frown \mathsf{seqPR}\ \mathit{perR} \frown \mathsf{seqTP}\ \mathit{tPriv})$$

We use $\mathsf{seqPR}\ \mathit{perR}$ and $\mathsf{seqTP}\ \mathit{tPriv}$ to denote the sequences of sets of references residing in the per-release and temporary private memory areas in perR and tPriv. Precisely, we have that $\#\mathsf{seqPR}\ \mathit{perR} = \#\mathit{handlers}$ and

$$\exists\, b : (1 \mathinner{..} \#\mathit{handlers}) \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{handlers} \bullet \\ \forall\, i : 1 \mathinner{..} \#\mathsf{seqPR}\ \mathit{perR} \bullet \mathsf{seqPR}\ \mathit{perR}\ i = \mathit{refsRes}\ (\mathit{perR}\ (b\ i))$$

We use $A \rightarrowtail\!\!\!\!\!\!\rightarrow B$ to refer to the set of bijective functions between A and B. Additionally, we define $\mathsf{seqTP}\ \mathit{tPriv} = \frown/(\mathsf{seqS}\ \mathit{tPriv})$, to get the sequence of all sets of references residing in all memory areas in all stacks in tPriv. In $\mathsf{seqS}\ \mathit{tPriv}$, we have a sequence of sequences of sets of references. We define that $\#\mathsf{seqS}\ \mathit{tPriv} = \#\mathit{handlers}$ and additionally

$$\exists\, b : (1 \mathinner{..} \#\mathit{handlers}) \rightarrowtail\!\!\!\!\!\!\rightarrow \mathit{handlers} \bullet \\ \forall\, i : 1 \mathinner{..} \#\mathsf{seqS}\ \mathit{tPriv} \bullet \mathsf{seqS}\ \mathit{tPriv}\ i = (\mathit{tPriv}\ (b\ i) \fatsemi \mathit{refsRes})$$

We use $(\mathit{tPriv}\ (b\ i) \fatsemi \mathit{refsRes})$ to denote the composition of $\mathit{tPriv}\ (b\ i)$ with the function $\mathit{refsRes}$. We observe that $\mathit{tPriv}\ (b\ i)$ is a stack of memory areas, and so a function from natural numbers to memory areas. By composing this function with $\mathit{refsRes}$, we get a function from natural numbers to the sets of references residing in the memory areas. Because $\mathit{tPriv}\ (b\ i)$ is a sequence, so is $\mathit{tPriv}\ (b\ i) \fatsemi \mathit{refsRes}$. This is a sequence of sets of references. The distributed concatenation of all such sequences (operator $\frown/$) defines $\mathsf{seqTP}\ \mathit{tPriv}$.

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 Sect. 4, we conclude that ***HSCJ*** is closed with respect to conjunction, disjunction, conditional, and sequence.

6. Programming variables and their values

Programming variables in the alphabet can be either specification or allocated variables. As the name suggests, specification variables are used only to write abstract definitions of the behaviour of programs (or their components). They model, for instance, inputs and outputs. Allocated variables, on the other hand, are included in one of the frames of 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. This is what links our mathematical account of a program, in terms of the names of the variables in its alphabet, with the way those variables are represented as values in memory. We have one healthiness condition for the program stack (***HV1***), one for the mission-sequencer stack (***HV2***), and one for the handlers stacks (***HV3***).

To define these conditions, we use a function $\mathit{vars} : \mathsf{stack}\ \mathit{Frame} \rightarrow \mathbb{F}\ \mathit{VName}$ that characterises the set of active variables according to a given stack. These are the variables in the domains of the frames:

$$\mathit{vars}\ \mathit{sf} = \bigcup \mathrm{dom}(\!|\ \mathrm{ran}\ \mathit{sf}\ |\!)$$

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 in declarations 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 $\langle \mathit{vn}\rangle$ 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 $\langle \mathit{vn}\rangle$ that identify a field of that object, or a field of one of its fields, and so on. For example, for the variable z in the stack s of Fig. 5, A contains only $\langle z\rangle$. In the case of x, we have $\langle x\rangle$, $\langle x, m\rangle$, $\langle x, n\rangle$, $\langle x, m, u\rangle$, and so on. For clarity, in examples, we describe these sequences as z, x, $x.m$, $x.n$, $x.m.u$, and so on.

The function V maps sequences of variable names to values. The sequences describe addressing paths that identify a variable or an object field, each of which contain a primitive or null value. V maps these paths, which are names, to the values they contain. For instance, it maps the name z to 2, and it maps the path $x.m.u$ to null.

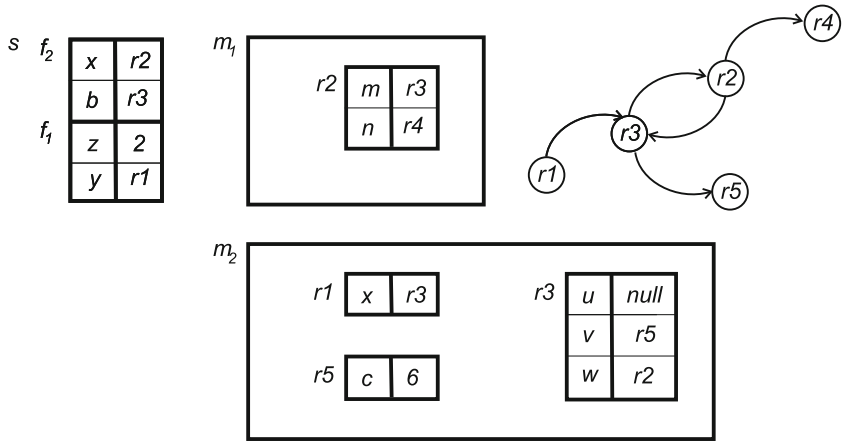

Fig. 5. Example memory configuration

This characterisation of values is the same used in [CHW06, HCW08], where we have defined a UTP theory for the Java memory model that captures the structure of objects and sharing.

We introduce a dereferencing function to yield the values associated with a variable name, relative to a stack and some memory areas. This function returns a pair: the first element is the set of all valid paths that start from variable name; the second element is a mapping from those paths to particular values.

Formally, we define the value $!(vn, sf, mas)$ of a variable name vn according to a stack sf and associated memory areas mas using the dereferencing function

$$!\_ : VName \times \text{stack } Frame \times \mathbb{F}\, MAreaC \to \mathbb{P}\, SName \times (SName \nrightarrow PValue)$$

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 $SName \nrightarrow PValue$ 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 $sval(vn, sf) = (\bigcup(\text{ran } sf))\ vn$ is the value of vn as recorded in sf. 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.

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

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. In its domain, we have only finite sequences in $A(vn, sf, mas)$: those that lead to a primitive value.

$$V(vn, sf, mas) = \left\{ \begin{array}{l} sn : \text{seq } VName;\ pv : PValue \mid \\ \quad \left(\begin{array}{l} sn \in A(vn, sf, mas) \land \\ \textbf{let } u == sval(vn, sf) \bullet \\ \quad u \in PValue \land pv = u \lor tpath(tail\ sn, \bigcup mas, u, pv) \end{array} \right) \end{array} \right\}$$

The condition $tpath(sn, ma, r, pv)$ requires that sn identifies a path in the memory area ma starting from r and terminating at the primitive value pv. Its formalisation is similar to that of $path(sn, ma, r)$ as shown below.

$$tpath(sn, ma, r) \Leftrightarrow \left(\exists sr : \text{seq } Val \bullet \left(\begin{array}{l} head\ sr = r \land \#sr = \#sn + 1 \land last\ sr = pv \\ \left(\begin{array}{l} \forall\ i : 1 \,..\, \#sn \bullet \\ \quad \left(\begin{array}{l} (sr\ i) \in \text{dom}\, ma \land \\ (sn\ i) \in \text{dom}(ma\ (sr\ i)) \land \\ sr(i+1) = ma\ (sr\ i)\ (sn\ i) \end{array} \right) \end{array} \right) \end{array} \right) \right)$$

In this case, we require the existence of a finite sequence sr of values, with r as its first element, pv as its last element, and that can be traversed using sn.

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.

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

The definition takes the conjunction over all the variable names in the program stack, and makes sure that the value for this name is the same as the one we get by dereferencing the corresponding name. We make the distinction between v as a mathematical variable, which denotes a value (the occurrence on the left-hand side in the equation above), and v as the name of a program variable (the occurrence on the right-hand side in the equation). The healthiness conditions ***HV2*** and ***HV3*** are similar.

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

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

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

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

7. History of memory usage

What we have not captured so far is, for instance, the fact that during the life time of a mission, we can only add objects to the mission memory, until it is cleared at the end of the mission. For the immortal memory, we have ***HSCJ1***, which defines $profile\ immortal \subseteq profile\ immortal'$ as an invariant. It is not the case, however, that $profile\ mission \subseteq profile\ mission'$, for example, is an invariant of our theory. Since *mission* is cleared when the mission is finished, and later used when a new mission is started, there is no guarantee that $mission'$ is 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. Once created, we can only add objects to these areas, until they are cleared.

To establish the required properties, we keep information about the program flow, in terms of the start and end of missions, of handler releases, and of `create` methods. For that, we consider a set *Ident* of identifiers for missions, releases, and `create` calls. The special identifier *none* is used when there is no mission or release in execution. For `create` calls, we use $Ident_p = Ident \setminus \{none\}$.

History is recorded in our theory by three additional alphabet variables; with them, we can state five extra healthiness conditions. We add to the alphabet first an injective non-empty sequence *missions* : $\mathsf{iseq}_1\ Ident$ of mission identifiers. Similarly, to record the history of handler releases during the current mission, we use a sequence $releases : handlers \to \mathsf{iseq}_1 Ident$.

Finally, to record the sequence of nested `create` calls during the current release of a handler, we use $creates : handlers \to \mathrm{seq}_1(\mathsf{iseq}\ Ident_p)$. An injective sequence of identifiers (different from *none*), that is, an element of $\mathsf{iseq}\ Ident_p$, is used to represent a particular configuration of the stack of nested calls: each call has a different identifier. As the stack grows and shrinks, the history of individual changes is recorded in a sequence, an element of $\mathrm{seq}_1(\mathsf{iseq}\ Ident_p)$. We have in *creates* such a sequence for the current release of each of the handlers. If there is no current release, the stack of calls is, of course, empty.

Example 7.1 If we use natural numbers as identifiers, when in a program three missions have started and finished, the value of *missions* might be $\langle 1, 2, 3, none\rangle$. If a fourth mission is then started, its value might become $\langle 1, 2, 3, 4\rangle$.

If for a particular mission, for example, 4 above, we have handlers a, b, and c, the function *releases* takes the value $\{a \mapsto \langle none\rangle, b \mapsto \langle none\rangle, c \mapsto \langle none\rangle\}$ when none of the handlers have been released yet. Similarly, in this situation, *creates* takes the value $\{a \mapsto \langle\ \langle\rangle\ \rangle, b \mapsto \langle\ \langle\rangle\ \rangle, c \mapsto \langle\ \langle\rangle\ \rangle\}$, because if no handler has been released, then no calls to `create` can have been made.

If the handler a, for instance, is released a few times, for instance, twice, *handlers a* takes a value like $\langle 1, 2\rangle$, if the second release is still active, or the value $\langle 1, 2, none\rangle$, if it is now finished. In this latter situation, *creates a* is again $\langle\ \langle\rangle\ \rangle$, because there can be no current calls to `create`.

If a is then released for a third time, so that *handlers a* takes the value like $\langle 1, 2, 3\rangle$, for example, and during this release, it makes two nested calls to `create`, then *creates a* takes the value $\langle\ \langle\rangle, \langle 1\rangle, \langle 1, 2\rangle\ \rangle$. If the second nested call is finished, and then a new call is made, the history record *creates a* is expanded to $\langle\ \langle\rangle, \langle 1\rangle, \langle 1, 2\rangle, \langle 1, 2, 3\rangle, \langle 1, 2, 4\rangle\ \rangle$. If all calls are finished, and a new fresh call is made, it evolves to $\langle\ \langle\rangle, \langle 1\rangle, \langle 1, 2\rangle, \langle 1, 2, 3\rangle, \langle 1, 2, 4\rangle, \langle 1, 2\rangle, \langle 1\rangle, \langle\rangle, \langle 5\rangle\ \rangle$.

As said above, if there is no current mission, the last element of *missions* is the identifier *none*. In this case, the mission memory is empty, and there are no handlers. This is captured by our next healthiness condition ***HH1***.

$$\textbf{\textit{HH1}} \ \widehat{=}\ \textbf{\textit{SIH}}(\mathrm{last}\ missions = none \Rightarrow refsRes\ mission = \emptyset \land handlers = \emptyset)$$

History records can only ever grow: history information should not be lost or altered. We do not keep in the record, however, the *none* identifiers. The following relation $s_1 \leq_H s_2$ between sequences of identifiers establishes when s_2 is a proper extension of the history record in s_1.

$$s_1 \leq_H s_2 \ \widehat{=}\ \mathrm{front}\, s_1 \leq s_2 \land \mathrm{last}\, s_1 \neq none \Rightarrow \mathrm{last}\, s_1 = s_2(\#s_1)$$

The next healthiness condition uses this relation to restrict changes to *missions*.

$$\textbf{\textit{HH2}} \ \widehat{=}\ \textbf{\textit{OIH}}(missions \leq_H missions')$$

Reflexivity and transitivity of this invariant follows from that of $\leq_H$.

Finally, if the mission history does not change, that is, during the execution of a particular mission, we can only ever add objects to *mission*, the set of *handlers* does not change, and the history of releases is preserved.

$$\textbf{\textit{HH3}} \ \widehat{=}\ \textbf{\textit{OIH}}\left(\begin{array}{l} missions = missions' \Rightarrow \\ \quad \left(\begin{array}{l} profile\ mission \subseteq profile\ mission' \land \\ handlers' = handlers \land \\ \forall\, h : handlers \bullet releases\ h \leq_H releases'\, h \end{array}\right) \end{array}\right)$$

Transitivity follows from transitivity of $\subseteq$, equality, and $\leq_H$.

If a handler is currently not released, then its per-release memory area is empty, as is its stack of temporary private memory areas.

$$\textbf{\textit{HH4}} \ \widehat{=}\ \textbf{\textit{SIH}}\left(\begin{array}{l} \forall\, h : handlers \mid \mathrm{last}(releases\ h) = none \bullet \\ \quad refsRes\ (perR\ h) = \emptyset \land creates\ h = \langle\ \langle\rangle\ \rangle \end{array}\right)$$

To identify the valid history records of all changes to the stack of calls to the `create` method, we first define a relation between sequences from $\mathsf{iseq}\ Ident_p$. It holds when only one change is necessary to obtain one from the other.

$$s_1 \sim_S s_2 \ \widehat{=}\ \ | \#s_1 - \#s_2 | \ = 1 \land (s_1 < s_2 \lor s_2 < s_1)$$

Additionally, in a proper history record for a sequence of stacks, if a new element is added to the stack at any point, its identifier should be fresh. This is the property *histSS ss* defined below for sequences *ss* of stacks.

$$histSS\ ss \ \widehat{=}\ \left(\begin{array}{l} \forall\, i : i\,..\,\#ss - 1 \bullet \\ \quad \left(\begin{array}{l} ss\ i \sim_S ss\ (i+1) \land \\ \#(ss(i+1)) > \#(s\ i) \Rightarrow \\ \quad \mathrm{last}(s(i+1)) \notin \bigcup \mathrm{ran}(\!|\ \mathrm{ran}\ ((1\,..\,i) \lhd ss)\ |\!) \end{array}\right) \end{array}\right)$$

The next healthiness condition establishes that all sequences of stacks in the range of *creates* satisfy the *histSS* property.

$$\textbf{\textit{HH5}} \ \widehat{=}\ \textbf{\textit{SIH}}(\forall\, h : handlers \bullet histSS\ (creates\ h))$$

The current number of nested calls to `create` determines the size of the stack of temporary private memory areas of the associated handler.

$$\textbf{\textit{HH6}} \ \widehat{=}\ \textbf{\textit{SIH}}(\forall\, h : handlers \bullet \#(\mathrm{last}(creates\ h)) = \#(tPriv\ h))$$

Finally, during the same release of a handler *h*, the contents of its per-release memory area can only be increased (with the addition of new objects), and the history of `create` calls can only be extended. Additionally, the contents of every stacked temporary private memory area that remains across observations, namely, those that correspond to the `create` calls in the range of the last stack in both *creates h* and *creates′ h*, can only be increased.

$$\textbf{\textit{HH7}} \ \widehat{=}$$

$$\textbf{\textit{OIH}}\left(\begin{array}{l} missions = missions' \Rightarrow \forall\, h : handlers \mid releases\ h = releases'\ h \bullet \\ \quad \left(\begin{array}{l} profile\ (perR\ h) \subseteq profile\ (perR'\ h) \land creates\ h \leq creates' h \land \\ \left(\begin{array}{l} \forall\, id : \mathrm{ran}(\mathrm{last}(creates\ h)) \cap \mathrm{ran}(\mathrm{last}(creates'\ h)); \\ \quad i : \mathbb{Z} \mid i = pos(id, \mathrm{last}(creates\ h)) \bullet \\ \quad\quad profile(tPriv\ h\ i) \subseteq profile(tPriv'\ h\ i) \end{array}\right) \end{array}\right) \end{array}\right)$$

The function $pos(e, s)$ determines the unique position in the injective sequence s in which the element e occurs. We observe that, when the previous healthiness conditions hold, $pos(id, \mathrm{last}(creates\ h))$ is equal to $pos(id, \mathrm{last}(creates'\ h))$.

We call the composition of the above healthiness functions ***HH***.

8. 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 [TPV10]). 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 [HCW08] 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 [HH03, CS06]; 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 [HHL10]. It extends the well-established JML [Bur+05] 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 Fig. 2. This is the subject of ongoing work, which formalises the SCJ programming model in *Circus* [OCW09], a refinement language based on Z and CSP. The semantic model of *Circus* is based on 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 be 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.

var *immortal*, *mission* . . . ; *P* ; **end** *immortal*, *mission* . . .

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.

Acknowledgments

The work reported here is funded by EPSRC (EP/H017461/1, "hiJaC: High-integrity Java Applications using Circus") and UKIERI (SA08-047, "Verified Software Initiative: Embedded Systems"). This paper is a revised and extended version of that presented at FM 2011 in Limerick [CWW11]; the results were also presented at IFIP WG 2.3 in Winchester in September 2011.

References

[Bar03] Barnes J (2003) High integrity software: the SPARK approach to safety and security. Addison-Wesley, Boston
[Bar05] Barnes J (2005) Programming in Ada 95. Addison-Wesley, Boston
[Bur+05] Burdy L et al. (2005) An overview of JML tools and applications. Softw Tools Technol Transf 7(3):212–232
[Bur99] Burns A (1999) The Ravenscar profile. Ada Lett XIX(4):49–52. ACM, New York
[CHW06] Cavalcanti ALC, Harwood W, Woodcock JCP (2006) Pointers and records in the unifying theories of programming. In: Dunne S, Stoddart B (eds) UTP symposium. Lecture notes in computer science, vol 4010. Springer, Berlin, pp 200–216
[CS06] Chen Y, Sanders J (2006) Compositional reasoning for pointer structures. In: Mathematics of program construction. Lecture notes in computer science, vol 4014. Springer, Berlin, pp 115–139
[CWW11] Cavalcanti A, Wellings AJ, Woodcock J (2011) The Safety-Critical Java memory model: a formal account. In: Butler M, Schulte W (eds) Proceedings FM 2011: 17th international symposium on formal methods. Lecture notes in computer science, vol 6664. Springer, Berlin, pp 246–261
[HCW08] Harwood W, Cavalcanti ALC, Woodcock JCP (2008) A theory of pointers for the UTP. In: Fitzgerald JS, Haxthausen AE, Yenigun H (eds) ICTAC08: theoretical aspects of computing. Lecture notes in computer science, vol 5160. Springer, Berlin, pp 141–155
[HH03] Hoare CAR, He J (2003) A trace model for pointers and objects. In: Programming methodology. Springer, Berlin, pp 223–245
[HH98] Hoare CAR, He J (1998) Unifying theories of programming. Prentice-Hall, Eaglewoods Cliffs
[HHL10] Haddad G, Hussain F, Leavens GT (2010) The design of SafeJML, a specification language for SCJ with support for WCET specification. 8th international workshop on Java technologies for real-time and embedded systems. ACM, New York
[Hat95] Hatton L, Safer C (1995) Developing software in high integrity and safety-critical systems. McGraw-Hill, New York
[He07] He J (2007) UTP semantics for web services. In: Davies J, Gibbons J (eds) Integrated formal methods. Lecture notes in computer science, vol 4591. Springer, Berlin, pp 353–372
[MIS07] MISRA (2007) Motor Industry Software Reliability Association. In: MISRA AC INT: introduction to the MISRA guidelines for the use of automatic code generation in automotive systems. ISBN: 978-906400-00-2
[OCW09] Oliveira MVM, Cavalcanti ALC, Woodcock JCP (2009) A UTP semantics for *Circus*. Formal Aspects Comput 21(1–2):3–32
[SCJDraft] Locke D, Andersen BS, Brosgol B, Fulton M (2010) Safety Critical Java specification, first release 0.76, The Open Group, 2010, UK. http://jcp.org/aboutJava/communityprocess/edr/jsr302/index.html.
[SCJS10] Sherif A, Cavalcanti ALC, He J, Sampaio ACA (2010) A process algebraic framework for specification and validation of real-time systems. Formal Aspects Comput 22(2):153–191
[SCS06] Santos TLVL, Cavalcanti ALC, Sampaio ACA (2006) Object orientation in the UTP. In: Dunne S, Stoddart B (eds) Unifying theories of programming. Lecture notes in computer science, vol 4010. Springer, Berlin, pp 18–37
[TPV10] Tang D, Plsek A, Vitek J (2010) Static checking of Safety Critical Java annotations. In: 8th international workshop on Java technologies for real-time and embedded systems. ACM, New York
[Ven07] Venners B (2007) Inside the Java virtual machine. http://www.artima.com/insidejvm/ed2
[Wel04] Wellings A (2004) Concurrent and real-time programming in Java. Wiley, New York
[WK11] Wellings A, Kim M (2011) Asynchronous event handling and Safety Critical Java. In: Concurrency and computation: practice and experience, vol 23. doi:10.1002/cpe.1756

Received 18 December 2011
Revised 9 May 2012
Accepted 27 May 2012 by Peter Höfner, Robert van Glabbeek, Ian Hayes and Cliff Jones