SCJ: Memory-safety checking without annotations

Chris Marriott and Ana Cavalcanti
cam505@york.ac.uk, ana.cavalcanti@york.ac.uk

University of York, UK

Abstract. The development of Safety-Critical Java (SCJ) has introduced a novel programming paradigm designed specifically to make Java applicable to safety-critical systems. Unlike in a Java program, memory management is an important concern under the control of the programmer in SCJ. It is, therefore, not possible to apply tools and techniques for Java programs to SCJ. We describe a new technique that uses an abstract language and inference rules to guarantee memory safety. Our approach does not require user-added annotations and automatically checks programs at the source-code level, although it can give false negatives.

1 Introduction

Verification is costly; techniques to automate this task are an interesting research topic. A recent contribution is Safety-Critical Java (SCJ) [1] - a specification for Java that facilitates static verification and is suitable for safety-critical programs.

The Real-Time Specification for Java (RTSJ) [2] was designed to make Java more suitable for real-time systems: it provides timing predictability. The guarantees of reliability needed for safety-critical systems are, however hard to achieve without further restrictions. SCJ strikes a balance between languages that are popular and those already considered adequate for high-integrity systems.

Our work is focused on memory safety of SCJ programs: the memory model is one of their main distinguishing features. The RTSJ introduces scoped memory areas that are not garbage collected, although the heap is available. The SCJ model removes access to the heap and limits the use of scoped memory.

The strict memory model of SCJ, however, does not ensure memory safety by construction, and every program must be checked. It is not enough to check absence of null-pointers and array-out-of-bounds exceptions. The memory areas form a hierarchy; objects cannot reference others stored in child memory areas.

SCJ programs are defined at one of three possible compliance levels: Level 0 programs follow a cyclic executive design and are the simplest, whereas Level 2 programs can make complex use of concurrency and sharing. We are interested in Level 1 programs, which are similar in complexity to Ravenscar Ada [3, 4]. Level 1 programs introduce concurrency and aperiodic events over Level 0.

As SCJ is relatively new, verification tools and techniques are currently fairly sparse, however, techniques such as those in [5] and [6] have established ways to

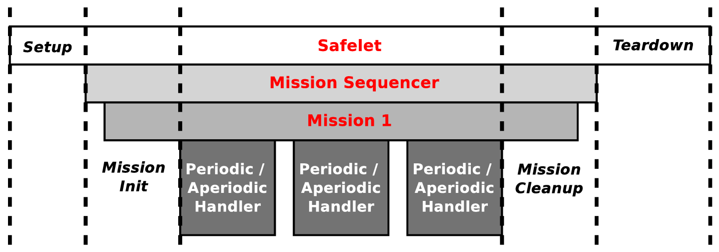
Setup
Safelet
Teardown
Mission Sequencer
Mission 1
Mission Init
Periodic / Aperiodic Handler
Periodic / Aperiodic Handler
Periodic / Aperiodic Handler
Mission Cleanup

Fig. 1. SCJ programming paradigm

check memory safety of SCJ programs through user-added annotations and bytecode analysis. We present an automated approach that operates at the source-code level without user-added annotations. It can, however, give false negatives.

Our technique uses an abstract language, *SCJ-mSafe*, to represent SCJ programs. Via abstraction, we focus on parts of SCJ programs required to verify memory safety, and present them in a consistent and structured format. Methods of a program are analysed individually to create a set of parametrised properties for each one that describes behaviour independently of the calling context. We define inference rules for *SCJ-mSafe* that describe memory safety for each component and apply them to the overall program. We assume the SCJ infrastructure is safe. For validation, besides constructing a tool and carrying out experiments, we have formalised our technique in Z [7].

The novelty of our approach is found in the abstraction technique, and the way in which we treat methods. In representing an SCJ program in *SCJ-mSafe*, we keep only the statements, methods, and classes that can influence memory safety. In checking the memory safety of an *SCJ-mSafe* program, we automatically calculate postconditions for each method. The postcondition of a method characterises its effect on the allocation of the fields and of the result, if any. Using this information, we can check the safety of method calls without restricting the calls to a specific scope. If there is a possibility that a method cannot be safe in any context, an error is raised during the calculation of its postcondition.

Section 2 of this paper introduces SCJ and its paradigm; our approach to verifying memory safety is discussed in Section 3. Our abstract language and translation strategy is described in Section 4, and the static-checking technique in Section 5. Section 6 describes our tool and some experiments we have conducted, before Section 7 draws some conclusions and describes our future work.

2 Safety-Critical Java

The SCJ programming paradigm is focused on the concept of missions. In Level 1 programs, missions are executed in sequence, and each mission executes a number of event handlers in parallel. Figure 1 shows the components for execution.

The entry point of an SCJ program is the safelet, which performs the setup procedures for a sequencer that controls the missions to be executed. When exe-

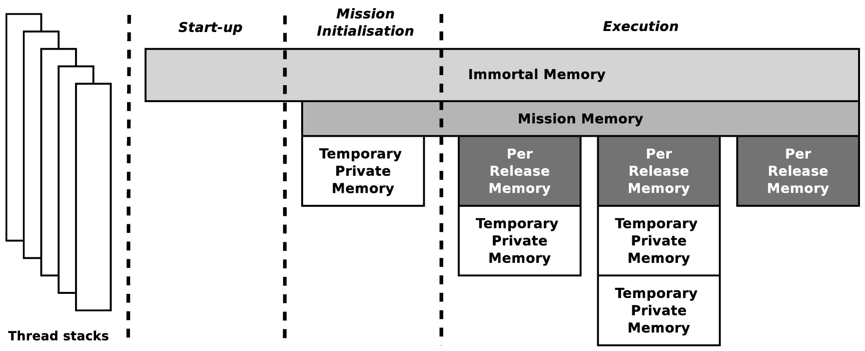

Fig. 2. SCJ memory structure

cuted, a mission goes through three phases: initialisation, execution, and cleanup. Objects used in missions are pre-allocated during the initialisation phase. In the execution phase, the event handlers are executed. When a mission has finished executing, the cleanup phase is entered for any final tasks. Level 1 programs can include periodic and aperiodic event handlers executed concurrently under the control of a fixed-priority pre-emptive scheduler.

Two types of memory area are used in the memory model: immortal and scoped. Each component of the paradigm has a default memory area; new objects created during execution are created in these associated areas unless specified otherwise. The safelet and mission sequencer are created in immortal memory, and allocate new objects in immortal memory. Individual missions are created inside the scoped mission memory area; new objects are created in the mission memory area, but can be created in the immortal memory. Event handlers are created in the mission memory, however once released, new objects are created inside a scoped per-release memory area associated with the handler. Handlers can create objects in the mission and immortal memory areas. Temporary private scoped memory areas can be used during the initialisation phase of a mission and by handlers; they are organised in a stack structure. Once a handler or mission finishes executing, the contents of its associated memory area(s) are reclaimed.

An example of this hierarchy of memory areas can be seen in Figure 2. It shows the immortal memory, mission memory, and three per-release memory areas associated with handlers in the mission. The mission and two of the handlers have their own private temporary memory areas. Finally, Figure 2 shows the thread stacks, which belong to the main program, mission sequencer, and event handlers; five stacks are used in this example.

To avoid dangling references, the SCJ memory model has rules to control their use. References can only point to objects in the same memory area, or in a memory area that is further up the hierarchy, that is, towards immortal memory.

Figure 3 shows an event handler in SCJ that repeatedly enters a temporary private memory area. It is part of a program taken from [8] that uses a single mission with a single periodic event handler. Its safelet, sequencer, and mission

```
public class Handler extends PeriodicEventHandler {
  int cnt;
  Object share = new Object();

  public Handler () {
    super (new PriorityParameters(11),
    new PeriodicParameters(new RelativeTime(0, 0),
                           new RelativeTime(500, 0)),
    new StorageParameters(10000, 1000, 1000), 500);
  }
  public void handleEvent() {
    System.out.println("Ping " + cnt);
    ++cnt;
    MyRunnable r = new MyRunnable();
    for (int i=0; i<10; ++i) {
      ManagedMemory m = (ManagedMemory) MemoryArea.getMemoryArea(this);
      m.enterPrivateMemory(500, r);
    }
    if (cnt > 5) {
      Mission.getCurrentMission().requestTermination();
    }
    share = new Object();
  }
}
```

Fig. 3. Nested Private Memory example in SCJ.

are omitted here for conciseness. The safelet creates the sequencer; the mission sequencer creates only a single mission with just one instance of `Handler`.

The handler's `handleEvent` method creates an instance of a runnable object and repeatedly executes it in a temporary private memory area using the `enterPrivateMemory` method. The example from [8] has also been expanded to include an additional class field `share`. This field is a reference and is instantiated with an object stored in mission memory, because fields of handlers are stored in the mission memory. When the `handleEvent` method executes, it executes in the per-release memory area associated with the handler; therefore, when `share` is re-allocated with a new object later, the SCJ memory safety rules are broken. We will continue to use this example throughout the paper, and will demonstrate later how our tool automatically detects this memory-safety violation.

3 Our approach to checking memory safety

Our technique has two main steps, as shown in Figure 4. The first step takes a valid SCJ program that is type correct and well formed according to the SCJ specification, and translates it into our new language called *SCJ-mSafe*, which is designed to ease verification. No information relevant to memory safety is lost,

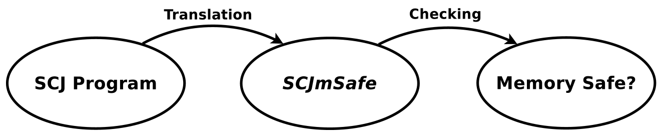

Fig. 4. Memory-safety checking technique

but all irrelevant information is discarded. Each SCJ program is described in the same style when translated to *SCJ-mSafe*; this makes programs easier to read and facilitates our analysis. A uniform structure also eases formalisation of *SCJ-mSafe* and of our checking technique, which is crucial in proving soundness.

In the second step, inference rules are applied to the *SCJ-mSafe* program using an environment that is automatically constructed to capture memory properties of expressions useful to determine memory safety. Each component of an *SCJ-mSafe* program has an associated rule that defines in its hypothesis the conditions that must be true for it to preserve memory safety. If all hypotheses of all rules applied to a program are true, then the program is memory safe. If any of the hypotheses are false, there is a possibility of a memory-safety violation.

Given an SCJ program, our technique automatically translates it into *SCJ-mSafe* and applies the memory-safety rules. In this way, we can verify safety without additional user-based input such as annotations, for example.

In general, the memory configuration at particular points of a program cannot be uniquely determined statically. It may depend, for example, from the values of inputs to the program. Since our aim is to perform a static analysis, we always assume the worst-case scenario for checking memory safety.

Our analysis is flow sensitive, path insensitive, context sensitive, and field sensitive. We consider the flow of the program by checking each command individually as opposed to summarising behaviour. We do not rely on precise knowledge of the control path. For example, we cannot determine statically which branch of a conditional statement is executed; we consider both branches. Although the behaviour may be different in each branch, the effect on memory may be the same; if not, the effects of both branches are considered separately in the remainder of the analysis. This creates a set of possible memory configurations during our analysis; each is updated throughout to give all possible scenarios of execution. Analysis of loops is also relatively straight forward as we consider every possible behaviour regardless of the iteration. This is achieved with a single pass where every execution path is analysed individually. Our analysis is context sensitive as we analyse methods based on their calling site, although each method is analysed once to establish a parametrised summary of behaviour. This summary is used in our analysis at each calling point of the method. Finally, we perform a field-sensitive analysis as we consider all fields of a referenced object when analysing assignments and new instantiations.

4 *SCJ-mSafe* and translation

SCJ-mSafe remains as close to SCJ as possible, and includes constructs to describe all behavioural components of the SCJ paradigm to reason about memory safety. This section introduces the language and describes the translation.

4.1 *SCJ-mSafe*

An *SCJ-mSafe* program is a sequence of definitions of components of an SCJ program: static fields and their initialisations, a safelet, a mission sequencer, missions, handlers, and classes. Every program follows this structure; it is not possible to combine the safelet and mission sequencer, for example.

The safelet component is comprised of class fields and their corresponding initialisation commands, any constructors, a `setUp` method, a mission sequencer, a `tearDown` method, and any additional class methods. The `setUp` and `tearDown` methods are declared separately from other methods as they are defined as part of the SCJ programming paradigm and identify the execution order of a program.

The order in which missions execute does not impact our analysis of memory safety. As only one mission executes at a time at Level 1, we treat each mission individually. Objects that are shared between different missions reside in immortal memory and are passed as references to missions. Even if the specific value of a shared variable cannot be determined because it is defined by missions that may execute earlier, our analysis still identifies possible memory safety violations for assignments or instantiations to subsequent fields. If a mission introduces a violation, it is caught; the order in which it is executed does not matter.

A mission is made up of its fields and the corresponding initialisation commands, any constructors, the `initialize` method, its handlers, the `cleanUp` method, and any additional user-defined class methods. The `initialize` and `cleanUp` methods execute before and after the handlers respectively.

Every handler component has its own unique identifier, and is made up of its fields and corresponding initialisation commands, any constructors, the `handleEvent` method, and any additional class methods. The handler of the nested private memory areas example is shown in Figure 5; it is very similar to the handler in SCJ shown in Figure 3, however, we do not distinguish between periodic and aperiodic handlers. We abstract away from the type of handler as our analysis does not rely on the scheduling of handlers.

User-defined classes are comprised of class fields and their corresponding initialisation commands, any constructors, and class methods.

Expressions and Commands As shown in Figure 5, expressions and commands in *SCJ-mSafe* are slightly different to those in SCJ. Some SCJ expressions are not required in *SCJ-mSafe* as they do not affect memory safety; the expression `++cnt` in Figure 3 is crucial to behaviour, but has no relevance to memory safety.

The important expressions in *SCJ-mSafe* are left expressions, which are expressions that can reference objects; identifiers and field accesses are left expressions. Values, identifiers, and field accesses denote objects manipulated in

```
handler Handler {
  fields {
    int cnt;
    Object share;
  }
  init {
    NewInstance(share, Current, Object, ());
  }
  constr () {
    PriorityParameters var1;
    NewInstance(var1, Current, PriorityParameters, (Val));
    ...
  }
  handleEvent {
    ...
    for ((int i; i = Val;), Val, (Skip;)) {
      ManagedMemory m;
      ManagedMemory var9;
      MemoryArea.getMemoryArea(this, var9);
      m = var9;
      m.enterPrivateMemory(Val, r);
    }
    if (Val) {
      Mission var12;
      Mission.getCurrentMission(var12);
      var12.requestTermination();
    } else {
      Skip;
    }
    NewInstance(share, Current, Object, ());
  }
}
```

Fig. 5. Nested Private Memory example in *SCJ-mSafe*.

a program whose allocations need to be checked. An identifier is a variable or an array access. Side effects are extracted as separate commands; all other SCJ expressions are represented as `OtherExprs`, which is a constant in *SCJ-mSafe*.

Commands in *SCJ-mSafe* include just a subset of those found in SCJ as not all commands in SCJ affect memory safety. For example, the assert statement is not part of *SCJ-mSafe* as it has no impact on memory safety. We do, however, include additional commands in *SCJ-mSafe*; SCJ expressions such as assignments, new instantiations, and method invocations are all represented as commands in *SCJ-mSafe*. They modify the value of program variables and are better characterised semantically as commands rather than expressions as in SCJ.

The *SCJ-mSafe* example in Figure 5 demonstrates several interesting differences between SCJ and *SCJ-mSafe*. In the constructor in Figure 3, the call to `super` includes several instantiations of objects that are passed as parameters. In *SCJ-mSafe*, a new variable is declared for each object and is instantiated individually; the new variables are then used as the parameters to the method call. For example, `var1` is a new variable of type `PriorityParameters`; it is then instantiated on the following line with the `NewInstance` command.

We note also the call to the `getMemoryArea` method in the `for` loop in the `handleEvent` method. In SCJ, the declaration and assignment to the variable `m` via a method call are defined on a single line; in *SCJ-mSafe*, the declaration and assignment are split. Also, because the right-hand side of the assignment is a method call, we introduce a new variable `var9`, which is used to store the result of the method call. The result of the `getMemoryArea` method is assigned to `var9` as it is passed as a result parameter to the method call. Finally, the reference stored in our variable `var9` is assigned to the original variable `m`.

The conditional statement below the `for` loop does not have an `else` statement in SCJ. In *SCJ-mSafe*, the `else` statement is always included, even if the behaviour of that branch is empty, or `Skip`. The command `Skip` describes a command that does nothing; it is also used to translate commands that we abstract away as they have no impact on the memory safety. Despite abstracting away the specific iteration, loops are maintained in our abstract language as the commands that form loop initialisations and loop updates must also be analysed.

Methods Methods in *SCJ-mSafe* are made up of the method name, return type, parameters, and method body. Further analysis, as discussed later, allows us to calculate the impact on memory safety of executing a specific method.

4.2 Translation

The translation from SCJ programs to *SCJ-mSafe* is not trivial, and includes analysis of the input program to create an *SCJ-mSafe* program with the consistent structure required for analysis. Using the specification language Z, we have defined a model of SCJ and *SCJ-mSafe* in order to formalise a translation strategy. We define the rules to specify memory safety using the same model. We have a Z model that defines SCJ and *SCJ-mSafe*, the translation strategy from SCJ to *SCJ-mSafe*, and the memory-safety checking technique.

Overall approach The translation strategy is defined by a series of functions that map SCJ components to corresponding *SCJ-mSafe* components. There are functions that translate the overall program, and functions that translate individual expressions. The function to translate the overall program takes an SCJ program and returns an *SCJ-mSafe* program.

$$\begin{array}{|l} Translate : SCJProgram \rightarrow\!\!\!\!\!\!+\; SCJmSafeProgram \\ \hline \forall\, program : SCJProgram \bullet \exists\, scjmsafe : SCJmSafeProgram \mid \; \ldots \end{array}$$

For all input SCJ programs, there exists a corresponding *SCJ-mSafe* program whose components are defined by further translation functions. The functions used to translate commands and expressions are used at every stage of the translation as each SCJ component (such as the safelet, missions, and so on) has commands in its own individual elements (such as methods).

Translating expressions Expressions in SCJ are found individually and as part of larger statements; for example `++cnt;` is a valid expression, however, `cnt;` is also a valid expression, but only makes sense as part of another statement. Expressions that identify values or references are translated into expressions; the remaining expressions that impact memory safety are translated to commands.

Accordingly, we define two translation functions for expressions. The first defines the translation of expressions into commands (*TranslateExpression*). This function takes an SCJ expression and returns an *SCJ-mSafe* command.

$$
\begin{array}{|l}
\mathit{TranslateExpression} : \mathit{SCJExpression} \pfun \mathit{Com} \\
\hline
\mathrm{dom}\ \mathit{TranslateExpression} \subset \mathit{WellTypedExprs} \\
\quad \land \forall \mathit{scjExpr} : \mathrm{dom}\ \mathit{TranslateExpression} \bullet \\
\qquad ...\ \lor (\exists\, e1, e2 : \mathit{SCJExpression} \mid \mathit{scjExpr} = \mathit{assignment}(e1, e2) \bullet \\
\qquad\quad (\mathbf{let}\ \mathit{lexpr} == \mathit{ExtractExpression}\ e1 \bullet \\
\qquad\qquad (\mathbf{let}\ \mathit{rexpr} == \mathit{ExtractExpression}\ e2 \bullet \\
\qquad\qquad\quad ...\,(\mathit{TranslateExpression}\ \mathit{scjExpr} = \\
\qquad\qquad\qquad \mathit{Seq}((\mathit{TranslateExpression}\ e2), (\mathit{Asgn}(\mathit{lexpr}, \mathit{rexpr})))))))
\end{array}
$$

The domain of *TranslateExpression* is a subset of valid SCJ expressions that are well typed (*WellTypedExprs*); for all SCJ expressions in its domain, the resulting *SCJ-mSafe* command is defined based on the type of expression; part of the case for assignments is shown above. For example, the assignment `a = b` is translated into the *SCJ-mSafe* assignment command $Asgn(a, b)$. More complex assignments, such as `a = (b = c)`, which contain side effects, are translated as a sequence (*Seq*) of commands. The result of applying *TranslateExpression* to `a = (b = c)` is $Seq(Asgn(b, c), Asgn(a, b))$. This is done by translating any embedded side effects into separate commands that come first in a sequence, followed by the overall expression; `b = c` is an embedded side effect of `a = (b = c)`.

To deal with expressions with side effects, we define *ExtractExpression*. It is used by *TranslateExpression* to extract the meaning of expressions whilst ignoring side effects. It takes an SCJ expression and returns an *SCJ-mSafe* expression.

$$
\begin{array}{|l}
\mathit{ExtractExpression} : \mathit{SCJExpression} \pfun \mathit{Expr} \\
\hline
\mathrm{dom}\ \mathit{ExtractExpression} \subset \mathit{WellTypedExprs} \\
\quad \land \forall \mathit{scjExpr} : \mathrm{dom}\ \mathit{ExtractExpression} \bullet \\
\qquad ...\ \lor (\exists\, e1, e2 : \mathit{SCJExpression} \mid \mathit{scjExpr} = \mathit{assignment}(e1, e2) \bullet \\
\qquad\quad \mathit{ExtractExpression}\ \mathit{scjExpr} = \mathit{ExtractExpression}\ e1) \\
\qquad ...\ \lor (\exists\, \mathit{name} : \mathit{Name};\ \mathit{id} : \mathit{Identifier} \mid \\
\qquad\quad \mathit{scjExpr} = \mathit{identifier}\ \mathit{name} \land \mathit{id} = \mathit{VariableName}\ \mathit{name} \bullet \\
\qquad\qquad \mathit{ExtractExpression}\ \mathit{scjExpr} = \mathit{ID}\ \mathit{id})
\end{array}
$$

The domain of *ExtractExpression* is also the subset of well-typed SCJ expressions. For all expressions in its domain, the *SCJ-mSafe* expression is extracted based on the type of the input expression. For example, when we apply the *ExtractExpression* function to `a[i = 10]`, the expression returned is `a[i]`, as it ignores the side effect `i = 10`. In the example `a = (b = c)`, the result of applying *ExtractExpression* to the left-hand side is the identifier `a`. When applied to the right-hand side, the assignment `b = c` is ignored and the identifier `b`, which is assigned to the left-hand side of the overall assignment (`a`), is returned.

If an expression has no embedded side effects, the result of *TranslateExpression* is the command `Skip`. For example, the SCJ assignment `a = b` has no side effects and is translated into the sequence `Skip` followed by `Asgn(a, b)`.

Translating commands If the SCJ command may impact memory safety, it is translated into the corresponding *SCJ-mSafe* command; otherwise, it is ignored. The exception is when a command has an embedded statement that may impact memory safety; the embedded statement is translated in this case.

$$
\begin{array}{|l}
\mathit{TranslateCommand} : \mathit{SCJCommand} \nrightarrow \mathit{Com} \\
\hline
\mathrm{dom}\ \mathit{TranslateCommand} \subset \mathit{WellTypedComs} \\
\quad \wedge\ (\forall\, \mathit{scjCom} : \mathit{SCJCommand} \bullet \\
\qquad \ldots\ \vee\ (\exists\, e1 : \mathit{SCJExpression};\ c1, c2 : \mathit{SCJCommand} \mid \\
\qquad\quad \mathit{scjCom} = \mathit{if}(e1, c1, c2) \bullet \\
\qquad\qquad \mathit{TranslateCommand}\ \mathit{scjCom} = \mathit{Seq}((\mathit{TranslateExpression}\ e1), \\
\qquad\qquad\quad (\mathit{If}((\mathit{ExtractExpression}\ e1), (\mathit{TranslateCommand}\ c1), \\
\qquad\qquad\qquad (\mathit{TranslateCommand}\ c2)))))
\end{array}
$$

The extract from the *TranslateCommand* function above shows we translate a conditional command in SCJ using *TranslateExpression* and *ExtractExpression*.

Translating methods The signature of an SCJ method is almost identical to an *SCJ-mSafe* method. Method calls in *SCJ-mSafe* are commands, and so the value or object returned from a method cannot be directly assigned to an expression. Instead, methods with a return type (that is not `void`) have an additional result parameter introduced during translation. For example, the method call `var = getMyVar(param);` is translated to `getMyVar(param, var);`.

A more in-depth description of the formalisation of *SCJ-mSafe*, the translation strategy, and the checking technique can be found in [9].

5 Static checking

Our technique for checking memory safety of *SCJ-mSafe* programs uses inference rules. These rely on a environment, which maintains a model of reference variables allocation in the program. In this section, we describe an environment used to check memory safety at a given point, our analysis of methods to define properties for each, and the inference rules to check memory safety.

```
public class MyMission extends Mission {
  CustomClass c;
  MemoryArea immortalRef;
  ...
  public void initialize() {
    int x,y;
    Object obj1 = new Object();
    Object obj2;
    if (x != y) { obj2 = new Object();
    } else { obj2 = immortalRef.newInstance(Object.class); }
    if (x > y) {
      c = new CustomClass();
      c.setField(obj1);
    } else {
      c = (CustomClass) immortalRef.newInstance(CustomClass.class);
      c.setField(obj2);
    } ...
```

Fig. 6. Environment explanation example in SCJ.

5.1 Environment

The environment records information about left expressions that reference objects. It is defined as a function, which has as its domain the set of possible expression-share relations of a program at a particular point of execution. An expression-share relation associates the left expressions in a program that share the same reference. The set of all expression-share relations is defined as follows.

$$\mathit{ExprShareRelation} == \mathit{LExpr} \leftrightarrow \mathit{LExpr}$$

Expression-share relations are mapped to expression reference sets.

$$\mathit{ExprRefSet} == \mathit{LExpr} \mathrel{+\!\!\!\to} \mathbb{P}\,\mathit{RefCon}$$

An expression reference set describes the set of possible reference contexts in which the objects referenced by left expressions may reside. The reference context of an object is an abstraction of the location to which its reference value points. This includes all memory areas in SCJ plus a new context *Prim*, which is for expressions of a primitive type. The definition of the environment is shown below.

$$\begin{aligned}
&\mathit{Env} == \{\mathit{env} : \mathit{ExprShareRelation} \mathrel{+\!\!\!\to} \mathit{ExprRefSet} \\
&\quad \mid \forall\, \mathit{rel} : \mathit{ExprShareRelation};\ \mathit{ref} : \mathit{ExprRefSet} \mid (\mathit{rel}, \mathit{ref}) \in \mathit{env} \\
&\qquad\quad \bullet \operatorname{dom}(\mathit{rel}^{\,*} \cup (\mathit{rel}^{\,*})^{\sim}) = \operatorname{dom} \mathit{ref} \\
&\qquad\quad \wedge (\forall\, e_1, e_2 : \mathit{LExpr} \mid e_1 \mapsto e_2 \in (\mathit{rel}^{\,*} \cup (\mathit{rel}^{\,*})^{\sim}) \bullet \mathit{ref}\ e_1 = \mathit{ref}\ e_2)\}
\end{aligned}$$

For every possible share of left expressions, there is a related function that describes the set of reference contexts in which the objects may reside. We take the reflexive, symmetric, and transitive closure of expression-share relations. This model allows us to capture information about all execution paths; for example, a share relation may have an associated reference set that includes a set of possible reference contexts for an object allocated in different memory areas on different execution paths. The environment may have multiple share relations mapped to a single reference set when assignments differ based on the execution path.

Consider the excerpt from a mission class shown in Figure 6. The `initialize` method includes conditional statements that affect the memory configurations. The reference `obj2` is instantiated in mission memory if the first condition is true, and in immortal memory if it is false. The allocation of the reference `c` and the argument of the method call `setField` depend on the second condition: if true, `c` is instantiated in mission memory and its field points to `obj1`; if false, `c` resides in immortal memory and its field points to `obj2`. The environment after the conditionals is below; it is simplified to illustrate the example and does not include the reflexive, symmetric, transitive closure of the expression shares.

$$\begin{aligned}
env = (&\{c.\mathit{field} \mapsto \mathit{obj}1\} \mapsto \{c \mapsto \{\mathit{MMem}\}, \mathit{immortalRef} \mapsto \{\mathit{IMem}\},\\
&\quad x \mapsto \{\mathit{Prim}\}, y \mapsto \{\mathit{Prim}\}, \mathit{obj}1 \mapsto \{\mathit{MMem}\},\\
&\quad \mathit{obj}2 \mapsto \{\mathit{IMem}, \mathit{MMem}\}, c.\mathit{field} \mapsto \{\mathit{MMem}\}\}),\\
(&\{c.\mathit{field} \mapsto \mathit{obj}2\} \mapsto \{c \mapsto \{\mathit{IMem}\}, \mathit{immortalRef} \mapsto \{\mathit{IMem}\},\\
&\quad x \mapsto \{\mathit{Prim}\}, y \mapsto \{\mathit{Prim}\}, \mathit{obj}1 \mapsto \{\mathit{MMem}\},\\
&\quad \mathit{obj}2 \mapsto \{\mathit{IMem}, \mathit{MMem}\}, c.\mathit{field} \mapsto \{\mathit{IMem}, \mathit{MMem}\}\})
\end{aligned}$$

The environment has two shares, as the assignments differ on each execution path. The first, where $c.\mathit{field} \mapsto \mathit{obj}1$, has $c \mapsto \{\mathit{MMem}\}$ and $c.\mathit{field} \mapsto \{\mathit{MMem}\}$ in its reference set, and is memory safe. The second, where $c.\mathit{field} \mapsto \mathit{obj}2$, has $c \mapsto \{\mathit{IMem}\}$ and $c.\mathit{field} \mapsto \{\mathit{IMem}, \mathit{MMem}\}$ in its reference set, and is not memory safe: the field of c points to an object that may reside in a lower memory area.

5.2 Methods

Methods can be executed in different memory areas. Typically, we cannot determine whether a method is always safe; whilst it may be safe to execute a method in a particular default allocation context, it may not be safe in another.

We do not restrict methods to specific allocation contexts; as part of the checking phase, methods in *SCJ-mSafe* are analysed to record properties that describe their behaviour from the point of view of memory allocation. In checking a call, we identify which method is called by extracting information from the left expression and the types of arguments passed. Due to dynamic binding, if more than one method matches the criteria of the method call, all are analysed.

Methods are recorded in our rules as elements of the following set.

$$\mathit{Method} == \mathit{Name} \times \mathit{Type} \times \operatorname{seq} \mathit{Dec} \times \mathit{MethodProperties} \times \mathit{Com}$$

The method name, return type, sequence of declarations (parameters), and command are as defined in the method description; the additional method properties describe the changes to the environment when the method is executed.

$$\mathit{MethodProperties} == \mathit{ExprShareRelation} \mathrel{+\!\!\!\!\!\rightarrow} \mathit{MethodRefChange}$$

The reference set in the environment is replaced by the *MethodRefChange* function, which uses meta-reference contexts (*MetaRefCon*) that contain all of the reference contexts defined previously, plus two additional ones to describe the current reference context of the callee (*Current*), and the set of reference contexts associated with a specific left expression in the environment (*Erc LExpr*).

$$MethodRefChange == LExpr \rightarrow\!\!\!\!\!\!+\;\; \mathbb{P}\, MetaRefCon$$

Meta-reference contexts allow us to describe the behaviour of methods independently of actual parameters of a method call; we can reason about method calls without checking each separate call. For example, consider the following method.

```
public void myMethod(A a, A b) {
   a.x = new CustomClass();
   b.x = a.x;
}
```

The result of calling this method with parameters `a` and `b` is as follows: the field `x` of the object referenced by variable `a` references a new instance of `CustomClass` located in the callee's current allocation context. Also, the field `x` of the object referenced by the variable `b` points to the same newly instantiated object referenced by `a.x`. Without knowing where a method is called, we capture this behaviour using meta-reference contexts. More specifically, we identify that `a.x` references an object in the *Current* reference context ($a.x \mapsto \{Current\}$), and `b.x` references the object associated with `a.x` ($b.x \mapsto Erc\ a.x$).

In conjunction with the environment, method properties allow us to establish at any point whether a method call can lead to a memory violation. The properties correspond to the changes to the environment.

5.3 Rules

We present the rule for the assignment command, as it can have a significant impact on memory safety. It is one of the commands that can change the environment most significantly, whilst also being able to cause memory violations.

$$\frac{\begin{array}{l} DominatesTop(\ LExprRc(\ lexpr, rc, e_1)\ ) \mapsto \\ DominatesLeast(\ e_1\ \ rexpr\ ) \in Dominates^{\,*} \end{array}}{mSafeCom_{\ e_1}\ (\texttt{Asgn(}lexpr, rexpr\texttt{)}, rc)}$$

The rule states that for an assignment `Asgn(`*lexpr*, *rexpr*`)` to be memory safe, a mapping between two reference contexts must be in the reflexive transitive closure of the *Dominates* relation. The *Dominates* relation describes the relationship between all reference contexts in *SCJ-mSafe*; for example, *IMem* dominates *MMem*, which means the immortal memory area is higher in the structure than mission memory. We can establish from this relation whether a mapping between reference contexts is safe; or more specifically, whether an assignment violates the rules of SCJ, which could potentially be a violation of memory safety.

In the rule above, the left-hand side of the mapping is the reference context in which the left expression is defined, if it is a variable, or the set of reference contexts in which the object may reside, if it is a reference. The *LExprRc* function determines the reference context(s) of a left expression. The highest reference context of the left expression (according to *Dominates*) must map to the lowest reference context of the object associated with the right expression to be safe.

The *DominatesLeast* function returns the lowest reference context in a set of reference contexts, according to the *Dominates* relation. We take the lowest reference context from the right expression as we must assume the worst case when checking mappings. Similarly, we use *DominatesTop* to establish the highest reference context in the set associated with the left expression.

For example, the assignment to `share` in the `handleEvent` method in Figure 3 is not memory safe. The reference context in which `share` is declared ($MMem$), does not dominate the reference context of the new object ($PRMem(handler)$).

The rule for the `enterPrivateMemory` command is below. It states that the command executed in the private memory area must be safe when analysed in the reference context rc_2, which is calculated using *EnterPrivMemRC*.

$$\frac{mSafeCom_{\ e_1}\ (c_1, rc_2)}{mSafeCom_{\ e_1}\ (\texttt{enterPrivateMemory(}c_1\texttt{)}, rc_1)}$$

where
$rc_2 = EnterPrivMemRC\ rc_1$

In Figure 5, the call to `enterPrivateMemory` is in the `handleEvent` method; the reference context at this point is the per-release memory area of the handler ($PRMem(handler)$). The result of *EnterPrivMemRC* is the first temporary private memory area associated with the same handler ($TPMem(handler, 0)$).

A complete set of rules have been specified for *SCJ-mSafe*. An initial set defined in [10] have been updated in [9]. We have defined all functions to update the environment after the execution of *SCJ-mSafe* components.

6 Tool and experiments

We have developed a tool called *TransMSafe* for the automatic translation and checking of SCJ programs. The tool is an implementation of the translation strategy and checking technique we have defined in Z, and is an extension to the tool described in [11]. The existing tool is implemented in Java and uses third-party utilities and libraries including the compiler API to aid analysis and translation of SCJ programs; it is tailored for modifications and extensions.

The tool has been applied to a number of examples including the CDx, PapaBench, and an automotive cruise-control system (ACCS). The CDx is a flight collision detection algorithm that calculates the possible collisions of aircraft based on their position and movement, and is a benchmark for SCJ [12]. The PapaBench is a real-time benchmark adapted for SCJ [13]. The ACCS is a Level 1 cruise-control system [14] with implementation described in [11].

We are able to translate all of these examples into *SCJ-mSafe* automatically; each translation executes in 1 to 2 seconds on an Intel Core i5 650 at 3.20GHz with 8GB RAM. No code optimisation has been performed. We have also translated and checked the SCJ Checker duplicated class example in [5], demonstrating our ability to automatically check memory safety without duplication of classes or annotations. Further results of checking experiments are given in [9]. The output of the tool is a textual representation in *SCJ-mSafe* ; it displays the environment during the checking phase for each command.

The tool is available as part of the hiJaC project tool suite and is freely available to download at `http://www.cs.york.ac.uk/circus/hijac/tools.html`. Instructions on how to install and run *TransMSafe* are in the read-me file.

7 Conclusions

We have described and formalised an abstraction technique to verify memory safety of SCJ programs. We introduced *SCJ-mSafe*, which is tailored to ease memory-safety verification. *SCJ-mSafe* programs have a uniform structure that abstracts away from some of the complexities found in SCJ programs. Inference rules are defined for each component of *SCJ-mSafe* in order to determine what it means for each to be memory safe. We use environments to store information required throughout the checking phase. These allow us to check each command in a program and ensure no violations of the SCJ memory safety rules are possible.

Another technique to verify memory safety of SCJ programs is found in the SCJ Checker [5], which is an annotation checker. The annotations are used to describe scopes in which classes, fields, local variables, and methods reside and run. This technique sometimes requires code duplication when instances of classes are required in different scopes, however no false negatives are produced. Not all valid programs can be checked without modification. Our technique may also require refactoring of SCJ programs to implement the components of the SCJ paradigm (safelet, missions, and so on) in different classes, for example.

A bytecode analysis technique to find illegal assignments occurring in Level 0 and Level 1 programs is described in [6]. The approach is an automated static-checking technique and uses a stack of SCJ memory areas and a points-to relationship to check for potential violations. Like our approach, this also uses an over-approximation of possible mappings and may raise false negatives.

The model checking technique in [15] has been applied to Level 0 SCJ programs. The analysis of Level 1 programs and aperiodic event handlers, which includes concurrency, is limited because of the state explosion problem. Although techniques to try and reduce this explosion, such as symbolic execution, have been developed, they have not been applied yet. We avoid these problems by abstracting away from such complex issues that do not always affect memory safety, like the execution order of missions, for example.

The translation of SCJ programs has been automated; our goal is to extend *TransMSafe* to automatically check a wider range of SCJ programs. We aim to apply our technique to several more complicated case studies. Our target is to verify Level 1 SCJ programs, therefore, aperiodic event handlers and concurrency are two important components of SCJ that must be considered.

Our approach can raise false negatives, and until we apply our technique to further case studies, it is difficult to estimate the frequency of their occurrence. We believe, however, that coding practices for safety-critical systems impose restrictions that minimise the number of false negatives.

A distinguishing feature of our work is the precise definition of *SCJ-mSafe*, the strategy for translation from SCJ to *SCJ-mSafe*, and the inference rules. This paves the way to a proof of soundness based, for instance, on the SCJ memory model in [4]. We have yet to attempt this, and do not underestimate the difficulty considering the coverage of the language we have achieved. We will be unable to prove that the translation from SCJ to *SCJ-mSafe* is correct, since it does not preserve every property of the SCJ program. We aim to prove that given an SCJ program with memory-safety violations, our technique will find the errors.

References

1. The Open Group: SCJ technology specification (v0.94). Technical report (2013)
2. Bollella, G., Gosling, J.: The Real-Time Specification for Java. Computer **33** (2000) 47–54
3. Burns, A.: The ravenscar profile. ACM SIGAda Ada Letters **11** (1999) 49–52
4. Cavalcanti, A., Wellings, A., Woodcock, J.: The Safety-Critical Java memory model: A formal account. FM 2011: Formal Methods (2011) 246–261
5. Tang, D., Plsek, A., Vitek, J.: Static checking of Safety-Critical Java annotations. In: Proceedings of Java Technologies for Real-time and Embedded Systems, ACM (2010) 148–154
6. Dalsgaard, A.E., Hansen, R.R., Schoeberl, M.: Private memory allocation analysis for SCJ. In: Proceedings of Java Technologies for Real-time and Embedded Systems, ACM (2012) 9–17
7. Woodcock, J.C.P., Davies, J.: Using Z—Specification, Refinement, and Proof. Prentice-Hall (1996)
8. Schoeberl, M.: Nested Private SCJ example. (2013) `www.jopwiki.com/Download`.
9. Marriott, C.: The formalisation of *SCJ-mSafe* - Technical Report. The University of York, UK. (2013) `http://www-users.cs.york.ac.uk/marriott/`.
10. Marriott, C.: SCJ Memory Safety with *SCJCircus* - Technical Report. The University of York, UK. (2012) `http://www-users.cs.york.ac.uk/marriott/`.
11. Zeyda, F., Lalkhumsanga, L., Cavalcanti, A., Wellings, A.: Circus models for Safety-Critical Java programs. The Computer Journal (2013) bxt060
12. Kalibera, T., Hagelberg, J., Pizlo, F., Plsek, A., Titzer, B., Vitek, J.: CDx: a family of real-time Java benchmarks. In: Proceedings of Java Technologies for Real-time and Embedded Systems, ACM (2009) 41–50
13. Nemer, F., Cassé, H., Sainrat, P., Bahsoun, J.P., De Michiel, M.: Papabench: a free real-time benchmark. WCET **4** (2006)
14. Wellings, A.: Concurrent and real-time programming in Java. Wiley (2004)
15. Kalibera, T., Parizek, P., Malohlava, M., Schoeberl, M.: Exhaustive testing of Safety-Critical Java. In: Proceedings of Java Technologies for Real-time and Embedded Systems, ACM (2010) 164–174