Safety-Critical Java in Circus

Ana Cavalcanti, Andy Wellings, Jim Woodcock, Kun Wei, Frank Zeyda
University of York

ABSTRACT
This position paper proposes a refinement technique for the development of Safety-Critical Java (SCJ) programs. It is based on the Circus family of languages, which comprises constructs from Z, CSP, Timed CSP, and object-orientation. We cater for the specification of timing requirements, and their decomposition towards the structure of missions and event handlers of SCJ. We also consider the integrated refinement of value-based specifications into class-based designs using SCJ scoped memory areas. We present a refinement strategy, and a Circus variant that captures the essence of the SCJ paradigm independently from Java.

Categories and Subject Descriptors
G.4 [Mathematical Software]: Verification

1. INTRODUCTION
An international effort has produced a specification for a high-integrity real-time version of Java: Safety-Critical Java (SCJ) [11]. SCJ is based on a subset of Java augmented by the Real-Time Specification for Java (RTSJ) [25], which supplements Java’s garbage-collected heap memory model with support for memory regions [23] called memory areas. The execution model of an SCJ program is based on missions and event handlers. Additionally, SCJ restricts the RTSJ memory model to prohibit use of the heap, and define a policy for the use of memory areas. The SCJ design favours certification. The plan for the standardisation work includes the production of a reference virtual machine, but, of course, no particular design technique.

Circus is a family of languages for refinement [2] based on a flexible combination of elements from Z [28], CSP [19], and imperative commands from Morgan’s calculus [12]. Variants and extensions of Circus include Circus Time [22], which provides facilities from Timed CSP [17], and OhCircus [3], which is based on the Java approach to object-orientation. The semantics of the Circus languages is based on the Unifying Theories of Programming (UTP) [8]. This is a relational framework that supports refinement-based reasoning about a variety of paradigms. It has been used to define object-oriented [20] and time [22] constructs, for instance.

We have recently presented a Circus-based formalisation of the mission execution model of SCJ [30], and a UTP theory for the memory model [4]. SCJ is organised in Levels (0, 1, and 2), with a decreasing amount of restrictions to the mission execution model. Our work is on SCJ Level 1, which corresponds roughly to the Ravenscar profile for Ada [1].

Our existing formalisations of SCJ can be used to justify the soundness of verification techniques for SCJ. What we present here is a proposal for a refinement strategy that builds on these results on SCJ formal semantics. We also rely on previous results on Circus variants and UTP theories for object-orientation, time, and references [22, 7, 20, 3].

We propose an approach for stepwise development of SCJ programs based on abstract models that do not consider the details of either the SCJ mission or memory models. Four Circus specifications characterise the development steps: we call them anchors, as they identify intermediate targets for refinement, and the design aspects treated in each step.

Each anchor is written using a different combination of the Circus family of notations. The first anchor is the high-level specification. The last is so close to an SCJ program as to enable automatic code generation. It is written in SCJ/Circus, a new version of Circus extended with constructs that correspond to the components of the SCJ programming paradigm. They are syntactic abbreviations for definitions that use a combination of existing variants of Circus to cater for time, object-orientation, and the SCJ memory model.

In the next section, we present SCJ, the Circus family, the UTP, and our formalisations of the SCJ mission and memory models. Section 3 presents our refinement strategy. The applications of the last anchor are discussed in Section 4, and a more general discussion of the refinement strategy is in Section 5. We draw our conclusions in Section 6.

2. PRELIMINARIES
We present now the background to our work.

2.1 Safety-Critical Java
At SCJ Level 1, missions are executed in sequential order. An SCJ mission consists of a set of handlers that are either released periodically, or respond to aperiodic events. They are executed in parallel by a priority-based scheduler, and access to shared data has to be performed by synchronized methods and statement blocks to avoid race conditions. A mission normally continues to execute until one of its han-
The sequencer and the missions are implemented by subclasses of the \texttt{Mission} and \texttt{MissionSequencer} abstract classes of the SCJ API. A concrete subclass of \texttt{MissionSequencer} implements the \texttt{getNextMission()} method to determine the next mission to be run. A concrete subclass of \texttt{Mission} overrides methods that initialise the handlers of that mission and carry out cleanup tasks. Event handlers are derived from one of the abstract classes \texttt{AperiodicEventHandler}, \texttt{AperiodicLongEventHandler}, or \texttt{PeriodicEventHandler}. They override the \texttt{handleAsyncEvent()} method to provide the code that is executed when the handler is released. The entry point of a program is an implementation of the \texttt{Safelet} interface, which creates the mission sequencer.

Each component of this programming model has an associated memory area, whose lifetime is that of the component. An immortal area holds objects throughout the lifetime of the program: they are never deallocated. A mission area is associated memory area, whose lifetime is that of the component. The action notation is a mixture of atomic and in

The key elements of \textit{Circus} specifications are processes. They interact with the environment through atomic and instantaneous events: either simple synchronisations, or input and output communications. Unlike CSP, a \textit{Circus} process also encapsulates local state, hidden to other processes.

A process specification defines its state as a \texttt{Z} schema. Local actions operate on the state, while possibly interacting with the environment. The action notation is a mixture of \texttt{Z}’s schema calculus, standard CSP constructs, and imperative commands from Morgan’s refinement calculus. A main action at the end specifies the process behaviour.

An example of a process \texttt{BReq} is in Figure 3. There, the state is defined by a schema \texttt{APState} whose single component is a collection \texttt{col} of type \texttt{P char}: it holds a set of characters. \texttt{Init} and \texttt{Insert} are local actions specified by \texttt{Z} operation schemas. \texttt{Init} defines the initial value of \texttt{col} to be the empty set. \texttt{Insert} adds an input \texttt{x}! to \texttt{col}.

Additionally, \texttt{InsertS}, \texttt{BReq1}, \texttt{BReq2} and \texttt{BReq3} are actions that use (Timed) CSP elements, including synchronisations (\texttt{c \rightarrow A}), input (\texttt{c?! x \rightarrow A(x)}) and output (\texttt{c?! v \rightarrow A}) prefixes, external choice (\texttt{\bigcirc}), and recursion (\texttt{\mu X \cdot A(X)}). Like in CSP, interactions (that is, synchronisations, inputs, and outputs) are written as part of a prefixing operator (\texttt{\rightarrow}). For example, in \texttt{BReq2}, we have a synchronisation on \texttt{out} prefixing an action, which is itself another prefixing with a synchronisation on \texttt{enable}. We, therefore, have that \texttt{BReq2} is ready to synchronise on \texttt{out}, then on \texttt{enable}, then accept an input \texttt{x} through the channel \texttt{send}, and finally recurse.

The main action (at the end after the \texttt{\bullet}) involves two parallelisms. A parallel composition (\texttt{A1 \parallel ns1 \parallel cs \parallel ns2 \parallel A2}) of actions is parametrised by a synchronisation set \texttt{cs} and two disjoint sets of variables \texttt{ns1} and \texttt{ns2} that the parallel actions may modify. In ( \texttt{BReq2 \parallel [\{ send \} \parallel BReq3}) we omit the name sets, as neither \texttt{BReq2} nor \texttt{BReq3} modify \texttt{col}: they synchronise on the channel \texttt{send}. This parallel action is, however, itself in parallel with \texttt{BReq1}, which modifies \texttt{col}.

Like actions, we can also combine processes in \textit{Circus} using the CSP operators, for example, sequentially (\texttt{P1; P2}), in parallel (\texttt{P1 || P2}) or in interleaving (\texttt{P1 \interleave P2}).

In \textit{OhCircus}, we also have classes; an example is in Figure 4. These, like processes, have a state (specified in \texttt{Z}). Behaviour, however, is not specified by actions, but methods: data operations over the state, specified using either \texttt{Z} or the guarded command language. In the example, methods are synchronised, so that there are never two methods being executed in parallel with the same object as a target.

\textit{Circus Time} includes also constructs in the style of Timed CSP. In a prefixing \texttt{c?t \rightarrow A}, the variable \texttt{t} records the number of time units during which the communication \texttt{c} is available. In \texttt{BReq1}, for instance, \texttt{in?x?t \rightarrow Skip} records the amount of time since the input through \texttt{in} is offered until it is taken and assigned to \texttt{x}. An action \texttt{wait d} terminates after \texttt{d} time units; also, \texttt{wait t1 \ldots t2} is a nondeterministic choice of a wait period \texttt{d} between \texttt{t1} and \texttt{t2}. A timeout \texttt{A1 \rightarrow A2} preempts the execution of \texttt{A1} if it does not engage in some interaction in \texttt{d} time units, in which case the action \texttt{A2} takes over. Finally, we have two extra deadline operators: \texttt{A\bullet d} asserts that \texttt{A} terminates within \texttt{d} time units, and \texttt{A\cdot d} asserts that \texttt{A} starts (via a visible interaction) within \texttt{d} time units. Data operations do not take time: any time properties need to be explicitly defined.

We have defined a \textit{Circus} modelling pattern that captures the SCJ framework behaviour, and also allows us to describe particular applications [30]. An overview of this pattern is in Figure 1. Here, we find the main components of the SCJ mission model: \texttt{Safelet}, \texttt{MissionSequencer}, \texttt{Missions}, \texttt{AperiodicHandler}, and \texttt{PeriodicHandler}. The overall specification of the system is their parallel composition. Each of them is specified as a process obtained by the parallel composition of a generic process (\texttt{FW}) that captures the SCJ framework behaviour, and an application-specific process (\texttt{App}) that models a class that implements the component. There is a process for each mission and event handler.

This \textit{Circus} model of an SCJ program provides a semantics for both the application and the virtual machine that executes it. It is the underlying model of the most concrete anchor of our refinement strategy (the \texttt{P} model in Figure 2).

\subsection{UTP and the SCJ memory model}

The UTP is part of a research agenda to understand the relationship between programming theories. Programming languages can be grouped by computational paradigm. An alternative classification treats each paradigm at different levels of abstraction, relating each computational model to an implementation technology. A final classification focuses on the method of presentation of the language definitions: denotational, algebraic, or operational. The UTP uses all three ways of classifying programming theories.

As well as providing an arena in which to study existing languages, the UTP can be used to define new ones. Having identified the major programming constructs, we can compose those we need in designing a new language. It is as though we were walking down the aisle of the technology supermarket, shopping for those features we need, confident that they can be plugged together to do what we want.

Each UTP theory is characterised by three things. The first is its alphabet: the set of observational variables of the
The variant used in SCJ-Circus differs from the existing Circus timing paths and to inform a schedulability analysis. If the deadline is viewed as a point in time, a deadline is said to be 

A theory recently presented formalises the SCJ memory model [4]. It has an alphabet variable for each memory area. In the case of the immortal, mission, and per-release (perR) areas, they associate references to object-values. In the case of a temporary private memory area (tPriv), we have a stack of such functions. Healthiness conditions guarantee, for instance, that objects cannot be deallocated from immortal.

Our structural model of SCJ memory areas is connected to a theory of object references [7] for pointers and hierarchical addressing as created by data types with recursive records. This theory has three observational variables: A, a set of hierarchical addresses; V, a partial function from addresses to values; and S, an equivalence relation on addresses. For a particular program, A describes all the legal names that can be constructed: all variables and field accesses. V maps the addresses of primitive (non-object) variables or fields to their values. Finally, S relates addresses that share a common location. Healthiness conditions guarantee, for example, that A is prefix closed (if a, b, c is valid, then so is a, b, and so on).

Our addition of a deadline operator to the Circus Time model in [22] is also relevant when reasoning about SCJ programs. It is similar to operators in existing process algebras and temporal logic [21, 9, 16]: we adopt a similar idea to that in [9]: failure to meet a deadline results in infeasibility in the model of the program. A deadline is viewed as an assertion for static analysis [26, 24]; is used to identify timing paths and to inform a schedulability analysis. If the deadline cannot be guaranteed, the program is rejected.

In each step of our refinement strategy, we use a different variant of Circus, and its underlying UTP theory. The variant used in SCJ-Circus is built from the following items in the UTP shopping-cart: nondeterministic imperative programming with specifications (based on Z) [27]; reactive processes with concurrency and communication (based on CSP) [15]; object orientation, with classes and inheritance (based on OhCircus) [20]; discrete real-time (based on Circus Time) [22, 24]; and an SCJ memory model [4]. The UTP agenda is far from complete, so some of these theories need to be brought to maturity and linked together.

3. REFINE MENT STRATEGY

In this section, we describe the steps of the refinement approach that we propose. Figure 2 shows the four Anchors A, M, E, and S, and other artefacts mentioned in the next section. The anchors are all written using different subsets and versions of Circus. The common UTP model of the Circus
variants makes it possible to establish a formal connection
between the anchors. It is the objective of the development
strategy that we present to guarantee that anchors are rel-
ated by refinement (□) as suggested in Figure 2. As an
example, we use a simple protocol, which repeatedly inputs
characters, and uses a serial line to output, when requested,
the number of different characters received.

3.1 Anchor A: abstract model

In this anchor, we use the basic Circus notation and Cir-
кус Time, including deadline operators. In other words,
we basically use the constructs of Z, CSP, Timed CSP, and Mor-
gan’s refinement calculus. Crucially, nothing is said about
classes, or objects and their allocation. An A anchor def-
ines the interaction pattern of the system in the style of
CSP. For that, it uses abstract data types in the style of
Z. Parallelism is used as a way of combining (conjoining)
requirements, rather than describing a concurrent design.

The A anchor for our example is in Figure 3. A first pro-
cess BReqs specifies the data and the interactions of the sys-
tem, that is, its behavioural requirements. Another process,
TReqs, defines the timing requirements. They are composed in parallel to define the system, AProtocol here.

The channel in is used to input characters, and out to
request an output: the number of different characters that
have been input (through in) so far. A synchronisation on
enable is used to enable a serial line for output; similarly,
synchronisation on disable releases the line connection. The
channel send is used to provide the output over the line.

The process AProtocol that defines the system is distin-
guished with the keyword system: just like process, it in-
roduces a Circus process. We use system to emphasise its
role in specifying the behaviour of the system to be imple-
mented. Semantically, there is no distinction to process.

There are three behavioural requirements BReq1, BReq2,
and BReq3; each is specified by a separate recursion. They
are composed in parallel in the main action to specify that
the system must satisfy all requirements. BReq1 defines
that the value x input in the communication in?x must be
inserted in the state collection, and that, when requested,
the size of the collection is to be output through send. The
action InsertS defines how the data operation Insert is used.

The process TReq specifies that, if before Insert conclud-
es, there is a need to send information, it must be done, but that decreases the amount of time w available for Insert. (Since wait 0, w is a timing property related to amount of time that a data operation may take, it is specified in BReqs, rather than TReqs.)

In BReq2, it is defined that, if there is a request for output,
then we must enable the line, and output via send. BReq3 states that once the data is sent, the line must be disabled.

The main action that specifies the behaviour of AProtocol
starts with a wait of up to 3 time units, to reflect the fact
that the system may take some time to initialise and start
accepting events. This is followed by the data initialisation
operation Init, and the parallelism that conjoins the three
behavioural requirements BReq1, BReq2, and BReq3.

The process TReqs introduces two timing restrictions that
are interleaved (||): no need to synchronise. TReq1 states
that the input must be read within 5 time units (•5), and
that in total 100 time units (wait 100) must be elapsed be-
fore an input is taken again. TReq2 states that if an output
is requested, then after at most 7 time units, the system
should ask for the line to be enabled. Moreover, once the
line is enabled, it cannot be held for more than 15 time units.

The variables col and z used in the UTP theories that de-
fine the main action of BReqs, for instance, are specification
variables. They denote values, not memory locations.

---

```
process BReqs ⊨ begin
  state APState ::= [col : P char]
  Init ::= [APState' | col' = 0]
  Insert ::= [ΔAPState; x : char | col = col ∪ {x}]
  InsertS(w) ::= (wait 0..w; Insert) □ (send!(# col)t → InsertS(w − t))
  BReq1 ::= (in?x& → InsertS(100 − t) □ send!(# col) → Skip); BReq1
  BReq2 ::= out → enable → send?x → BReq2
  BReq3 ::= send?x → disable → BReq3
  • wait 0..3; Init; (BReq1 [ ]{col} | {send} | {} | BReq2 [ ] {send} ] BReq3 )
end
process TReqs ⊨ begin
  TReq1 ::= ((in?x → Skip)▷5 )|| wait 100); TReq1
  TReq2 ::= out → wait 0..7; enable → (disable → Skip)▷15; TReq2
  • TReq1|| TReq2
end
system AProtocol ⊨ BReqs [ ] in, out, enable, disable ] TReqs
```

Figure 3: Protocol: Anchor A
3.2 Anchor M: Memory allocation

In this step, the target is an A anchor, which can use classes and objects. The object-oriented constructs are those of OhCircus. A variable of a class type denotes a reference to an object, and a variable of a Z type denotes a value. For our example, an M anchor is sketched in Figure 4. The process MBReqs is a refinement of BReqs. It implements col as an object of a class List. The Z data operations of the A anchor are implemented by methods of List.

Like in [3], we use data refinement to replace Z data types with class types. Since data refinement preserves the structure of actions, this affects just the data operations. In Figure 4, we show the definitions of BReq1 and InsertS in MBReqs; they are changed because they involve data operations. All other actions and processes from the A anchor are kept unchanged, and are therefore omitted in Figure 4.

In List, the method elems is logical: it does not need to be provided in the final implementation, but is used in the specification. With the use of elems, the specifications of insert and size are basically those of the corresponding Z operations. The public methods insert and size are declared to be synchronized: this follows the SCJ paradigm.

The UTP theory for the Circus language used in this step includes a combination of those in [20, 7]. A declaration of a variable of a class type, like l or next, enriches the set of addresses (in A). An assignment l := new List, for instance, also enriches A. The memory model here is very much that of Java, where memory deallocation is unspecified.

3.3 Anchor E: Execution model

The objective of this step is to introduce the parallel design of missions and handlers. It has four phases. The first, CP, removes the parallelism used in the A anchor to specify requirements. The second, SH, defines how variables are shared between missions and handlers. The third, MH, introduces the sequences and parallelisms that reflect the architecture of the missions and handlers. The final Phase AR uses algorithmic refinement to derive the implementation of the methods and handling routines.

3.3.1 Phase CP: Collapse parallelism

In the CP phase, we reduce the parallelism of recursions in the M (and A) anchor to a single recursion. As mentioned before, parallelism is used in an A anchor to conjoin requirements, not to model concurrency. In this phase, we eliminate that parallelism, which is convenient only to specify requirements. What we obtain is a sequential account of the specification that is useful as a normalised starting point to introduce the concurrent design of the program.

The result of this phase for our example is sketched in Figure 5; we name the refined system process EProtocol. After the initialisation, the behaviour of EProtocol is given by the action InPending(0, 5), whose parameters t and d are the time since the start of the in cycle, and the deadline on the communication through in. InPending(t, d) defines that, at the beginning of the cycle, a communication through either in or out is possible. After an input in?x@u, the behaviour is defined by AfterInPinset(t + u, 100 – (t + u), x); and after a request out@u, by InAfterOut(t + u, d – u, 7).

AfterInPinset takes as parameters the time t since the start of the in cycle, and the amount of time wins available to insert the third parameter x in the list. An internal choice (!) defines the amount of time d actually taken. Before d time units have elapsed, AfterInPinset offers a synchronisation on out; if it happens, its behaviour is that of AfterOutPinset. If d time units are over before a synchronisation on out, a time out ensures that the insertion is completed, and the behaviour is defined by AfterIn.

For conciseness, in Figure 5 we omit the definitions of InAfterOut, AfterOutPinset, and AfterIn, and of the other
local actions that they use. They characterise a sort of state machine that defines the behaviour of the system. Step laws that evaluate the parallelism of actions can be used to calculate the definition of \( InPending(t, d) \) and of the other local actions. A collection of such laws are available for Circus. Extra laws needed here are the subject of our future work.

### 3.3.2 Phase SH: sharing

In the SH phase, we define the object-valued state components that are to be allocated in immortal, in mission, and in per-release or temporary private memory areas. Those to be allocated in immortal memory remain as state components.

The components to be allocated in mission memory become local to a new action \( MArea \). To call their methods, we use new channels. In our example, \( MArea \) models \( l \), and the state of \( EProtocol \) becomes empty (see Figure 6). For each method, we have a channel to signal its call and another to signal the return; in our example we have \( insertLC \) and \( insertLR \), for instance. Method calls are recursively \((\mu)\) offered over and over again in choice, since they refer to synchronised methods (see Figure 4). In Figure 6, we show how the definition of an action like \( AfterInPin\), which uses data in the mission memory, is changed to interact with \( MArea \). We omit the definition of the other actions of \( EProtocol \), some of which are affected in a similar way.

In the new main action, \( MArea \) is in parallel with another new action \( System \). It defines the behaviour of the system as that specified by the main action obtained in Phase CP after initialisation, but using the variables via \( MArea \). If there are variables to be allocated in the immortal area, they can be modified by \( System \) directly, but not by \( MArea \). In our example, there are no such variables, so that the name sets of the parallelism are empty and omitted.

The new method channels are internal, and so hidden \((\backslash)\). Communications through them, therefore, do not take up any time when \( System \) is ready to synchronise. This ensures that they do not interfere on the timing properties of the system. In the next step of refinement, they are refined to commands that use the SCJ memory areas.

The state components to be allocated in per-release or temporary private areas become local to \( System \). (This refinement is trivial.) In the next step, they are further localised in new actions that represent the handlers.

### 3.3.3 Phase MH: missions and handlers

In this phase, we introduce one action for each mission, and one action for each handler. A refined \( System \) action defines the sequence of missions. A mission action is the parallel composition of the actions that model its handlers.

In our example, we have a single mission, with two handlers.
in interleaving; our $\mathbf{E}$ anchor is sketched in Figure 7. We omit the definition of $\mathbf{M Area}$, which is that in Figure 6.

Guidance and automation for this step of refinement has to rely on the design decisions of how to decompose the program into missions, and each mission into handlers. For each handler, it is necessary to define the events that it handles. If an event is the concern of more than one handler, synchronisation between them is required.

The refinement laws to be used here are transformation laws that introduce parallelism. They are part of the basic $\mathbf{Circus}$ refinement strategy in [2, 3]. The techniques presented there are too general to allow automation. In the case of the strategy that we propose here, automation is possible due to the restricted architecture of missions.

3.3.4 Phase $\mathbf{AR}$: algorithmic refinement

In the final Phase $\mathbf{AR}$, we mostly carry out algorithmic refinement using [5]. In our case, we just need to refine the $\mathbf{Z}$ schemas that specify the methods $\text{insert}$ and $\text{size}$ in the class $\mathbf{List}$ (see Figure 4). The method $\text{elems}$ is logical and can be eliminated after the refinement of $\text{insert}$ and $\text{size}$ removes the calls to it. If extra classes are needed to provide a better design, the techniques in [3] can be used. It is in this phase that the burden of proof lies.

3.4 Anchor $\mathbf{S}$: Safety-Critical Java

An $\mathbf{E}$ anchor embeds the structure of missions and memory areas. In this step, we produce the $\mathbf{S}$ anchor, which describes them in terms of the framework presented in Section 2.2.

As already explained, the $\mathbf{S}$ anchor can be written using the same $\mathbf{Circus}$ variant used in the $\mathbf{M}$ and $\mathbf{E}$ anchors, with the addition only of constructs to use the SCJ memory areas. Instead, however, we use paragraphs for declaration of safelies, mission sequencers, missions, and handlers. These are just abbreviations, but they highlight the main components of an SCJ program. Reasoning in this step can be based on refinement laws for these constructs proved in terms of laws for classes and processes.

The $\mathbf{S}$ anchor for our example is presented in Figure 8. It includes a sequencer, a mission, and two handler paragraphs. Although omitted in Figure 8, the classes are also included; in our example, we have $\mathbf{List}$.

Since the only $\mathbf{List}$ object $l$ is in the $\mathbf{M Area}$ action in the $\mathbf{E}$ anchor, the objects created in the $\text{insert}$ method are allocated in the mission memory. For that, we use assignments of the form $\text{next} := \text{newm List}$, whose semantics is given using the theory in [4]. The separation enforced by the SCJ memory area model ensures that it is a correct implementation of the access policy defined by $\mathbf{M Area}$.

As discussed in Section 2.2, a safelie, for example, is described by the parallel composition of two $\mathbf{Circus}$ processes: one representing the SCJ framework and another representing the corresponding application class. The same applies to the other components of the SCJ paradigm. The paragraphs of an $\mathbf{S}$ anchor define these parallelisms, and the complete $\mathbf{Circus}$ specification. The paragraph $\text{safelie}$, for instance, defines the $\text{setUp}$ and $\text{tearDown}$ methods. In our simple example, they are empty (modelled as $\text{Skip}$), and so the $\text{safelie}$ paragraph is omitted. The $\mathbf{Circus}$ specification defined by the $\mathbf{S}$ anchor in Figure 8, along with the code of an SCJ program that implements our protocol, is in [29].

In the sequencer paragraph, the $\text{state}$ clause declares the fields of the SCJ mission-sequencer class, which determine the state components of the mission-sequencer application process. In the initial clause, we have the SCJ class constructor, and so the initialisation operation of the process. Finally, $\text{getNextMission}$ defines the body of the $\text{getNextMission}$ method, and the corresponding $\mathbf{Circus}$ action in the mission-sequencer process. The assignment to the special variable $\text{ret}$ defines the mission to be returned. (The variable $\text{ret}$ is an implicit result parameter of the method.)

For each handler, we have a mission paragraph. Its state contains the field variables of the SCJ mission; it is that of the corresponding mission application process. The initialize clause defines the initialize method of the mission. It creates the mission fields, in our example, $l$, and defines the handlers of the mission. These are declared by a newHandler command, whose body uses the constructor of a type defined by a handler paragraph. In our example, we have two handlers defined using $\text{Handler1}$ and $\text{Handler2}$. Finally, a mission paragraph defines the cleanup method. In our example, the definition is trivial: $\text{Skip}$.

The definition of a $\mathbf{Circus}$ process that models a handler depends on whether it is periodic or aperiodic. In our simple example, we have one periodic handler that handles one input through the channel $\text{in}$. The second handler is aperiodic, and performs three synchronisations on $\text{out}$, $\text{enable}$, and $\text{disable}$, and one output through $\text{send}$.

A periodic paragraph defines a periodic handler, whose period is defined in the declaration of the paragraph. In our example, it is 100. The state paragraph defines the fields of the SCJ handler class. In our example, it is a refer-
sequence \texttt{MainMissionSequencer} \triangleq \textbf{begin}
  \texttt{state} \texttt{MainMissionSequencerState} \triangleq [\texttt{mission\_done} : \text{Bool}]
  \texttt{initial} \triangleq \texttt{mission\_done} := \textbf{false}
  \texttt{getNextMission} \triangleq \textbf{if} \ \texttt{mission\_done} = \textbf{false} \rightarrow \texttt{mission\_done} := \textbf{true}; \ \text{ret} := \texttt{ProtocolMission}
  \texttt{cleanup} \triangleq \begin{cases} \texttt{mission\_done} = \textbf{true} \rightarrow \text{ret} := \textbf{null} \\ \text{false} \end{cases}
\textbf{end}

\textbf{mission} \texttt{ProtocolMission} \triangleq \textbf{begin}
  \texttt{state} \texttt{MState} \triangleq [l : \text{List}]
  \texttt{initialize} \triangleq \begin{cases} \texttt{l} := \texttt{newList}; \ \texttt{(newHandler(Handler1(l)))}; \ \texttt{(newHandler(Handler2(l)))} \\ \text{false} \end{cases}
  \texttt{cleanup} \triangleq \textbf{Skip}
\textbf{end}

\textbf{periodic}(100) \textbf{handler} \texttt{Handler1} \triangleq \textbf{begin}
  \texttt{state} \texttt{Handler1\_State} \triangleq [l : \text{List}]
  \texttt{initialize} \triangleq \texttt{l} := \texttt{newList}; \ \texttt{(newHandler(Handler1(l)))}; \ \texttt{(newHandler(Handler2(l)))}
  \texttt{cleanup} \triangleq \textbf{Skip}
\textbf{end}

\textbf{aperiodic} \textbf{handler} \texttt{Handler2} \triangleq \textbf{begin}
  \texttt{state} \texttt{Handler2\_State} \triangleq [l : \text{List}]
  \texttt{initialize} \triangleq \texttt{l} := \texttt{newList}; \ \texttt{(newHandler(Handler2(l)))}; \ \texttt{(newHandler(Handler2(l)))}
  \texttt{dispatch} \ (\texttt{in} \ ? x @ t \rightarrow \textbf{handleAsyncEvent}(x, 100 - t))\triangledown
\textbf{end}

\textbf{Figure 8:} Sorter: Anchor S

ence to the list \texttt{l} in the mission area. The class constructor, defined in the \textit{initial} paragraph, takes it as a parameter. The handler method \texttt{handleAsyncEvent} is captured by the \texttt{handleAsyncEvent} paragraph. In our example, it is just a call to the relevant method of \texttt{List}. The \texttt{dispatch} paragraph captures the behaviour in each release of the handler (typically using a call to \texttt{handleAsyncEvent}). In the underlying \texttt{Circus} definition of the handler, we have a class and an associated process that defines the communications, and how the handler interacts with the SCJ framework. The \texttt{aperiodic} paragraph is similar in structure.

In the \texttt{safeflet} and \texttt{sequencer} paragraphs, use of \texttt{new} leads to allocation in immortal memory, and in the \texttt{mission} and \texttt{initial} paragraphs of a \texttt{handler}, to allocation in the mission memory. In our example, \texttt{l} := \texttt{newList} in \texttt{initialize} creates \texttt{l} in mission memory. In the \texttt{handleAsyncEvent} and \texttt{dispatch} paragraphs, allocation occurs in the relevant per-release memory by default. Use of the temporary private memory areas has to be explicitly indicated.

Given its structure, it is simple to generate code from an S anchor automatically. Other uses are discussed next.

4. FROM THE S ANCHOR

As said above, the semantics of an S anchor is a \texttt{Circus}-based (and, therefore, UTP-based) specification: the P model in [30]. This means that we can \texttt{Circus} and the UTP to reason about aspects of the program modelled by the S anchor.

Complementarily, it is possible to generate S models from SCJ programs that follow an organised pattern of programming. It requires, for example, the use of separate classes to define the safeflet, the mission sequencer, the missions, and each of the handlers. As far as we can see, this does not impose any serious practical restrictions. At the moment, however, examples of SCJ programs are few and far between, as the technology has not yet reached maturity. An SCJ version of the PapaBench (a real-time embedded systems benchmark) has been developed (d3s.mff.cuni.cz/\texttt{malohlava/projects/jpapabench}) and can provide a basis for further experiments along with the examples in rtjava.blogspot.com/2011/05/safety-critical-java-case-studies.html.

Generation of S models from programs can in itself be used as a proof technique. For example, every program written in the UTP theory for the SCJ memory model is memory safe. So, any SCJ program that can be described using an S model is memory safe. With a larger number of examples, it is also possible to investigate refactoring strategies to enforce the design patterns required for the automatic generation of S models. At this stage, however, our purpose is to identify good programming practice that facilitates reasoning.

Design patterns are at the heart of our proposed development and verification approach. The constrained structure of an S anchor guides our refinement strategy. Its detailed account and proof of soundness, and its application to major case studies, are our main challenges ahead.

Regarding time, our major concern is decomposition via refinement. There are a number of works along this line. Mukherjee et al. [13] have developed a framework consisting of a real-time specification language (NewThink) based on VDM-SL, an implementation language (NewSpeak) designed for real-time safety-critical systems, and a collection of rules for decomposing specifications into code. Specifications in NewThink are expressed using pre and postconditions with a time clause. As in our approach, correctness of the decomposition is guaranteed by the rules. Their focus
is on sequential systems, but we believe that Circus rules similar to theirs might be useful in our work.

Instead of decomposing time specifications into code, the work in [14] supports decomposition of a system specified in Duration Calculus into an untimed system communicating with some timers. The idea of separating time aspects from untimed behaviours might be also useful in reasoning about time constraints when localising high-level requirements in SCJ handlers during our refinement procedure.

Our refinement strategy is inspired by the work in [9], in which time is introduced into Morgan’s calculus. The advantage of this approach is that derivation of code from specifications is similar to that for untimed specifications. In our approach the requirements in the A anchor are localised in the handlers in the E and S anchors. The S anchor is, in this way, annotated with the machine-independent timing requirements that every correct implementation (in a specific platform) needs to satisfy. Verifying that they do may require, for instance, schedulability analysis.

This approach complements that in [6], where worst-case timing analysis is carried out based on annotations and information about particular processors. While [9] has concentrated on a sequential language, for SCJ we need laws to decompose the timing requirements over the parallel structure of missions and their handlers.

Finally, the framework components of the P model are a high-level specification for an SCJ implementation. It can be used either to facilitate the development of an SCJ implementation or to aid in the proof of the correctness of an existing implementation (for example, the Reference Implementation for SCJ [11] or the forthcoming Level 1 implementation for Open Safety Critical Java (ovmj.net/oscj)). For that, we need to construct a Circus model of the API.

5. DISCUSSION

The technique proposed here is an idealised development approach based on correctness by construction. Together with its soundness justification, the technique provides a solid programming theory for the SCJ paradigm. A number of additional practical issues can be addressed in that context; they are part of our agenda for future work.

Programming with pointers (or references) is error-prone. The chief culprit is aliasing, which happens when a memory location is addressed by more than one name. A write access using one of these names implicitly modifies the values associated with all the aliases, and this makes reasoning difficult. One solution is a holistic approach, in which we keep control of every variable and memory location in the program. If we develop the program in a top-down fashion, with no use of libraries, as suggested so far, this is effective.

This, however, breaks down if we want to use library components relying on their interfaces. As an example, we consider a component for calculating the mean of a sequence of numbers. Its abstract specification uses a sequence s with two operations: one to add an element to the sequence, and one to calculate the mean. Its implementation uses just two memory locations to hold the sum of the numbers and the size of the sequence. To reason about the system using the interface of the component, it is important to say that the storage used to implement s is not corrupted by anything else in the system. We cannot, however, refer to sum and size without breaking encapsulation, and giving up on modular reasoning. This degenerates into requiring a holistic view. If, however, we do not mention the implementation’s memory locations at all, then our reasoning is unsound in the presence of aliasing. The problem is to control aliasing without knowing which memory locations are being used.

Research to solve this problem is dominated by two approaches: separation logic and dynamic frames. Separation logic is an axiomatic technique [18]. Dynamic frames [10], on the other hand, is model oriented and fits in well with Circus. Dynamic frames are abstract variables whose values are sets of memory locations; specifications determine the frame required for the methods and how this changes at runtime. The user of an abstract interface can rely on the implementation accessing only the locations in the interface’s frame, but must guarantee to use a region of memory disjoint from that frame. The SCJ memory model can be used to implement this balance, making any sharing deliberate and explicit.

One of the advantages of Java is the availability of a wide range of class libraries. They, however, present a challenge both for our refinement strategy, as described above, and for use with the SCJ memory model. Java libraries generally assume that memory allocation occurs on the heap, and unused objects are garbage collected. Outside this context, they are likely to generate errors. It is necessary to determine how memory is used internally. Alternatively, SCJ-aware libraries must be developed; they need to be constrained to be useable in any memory area.

To evaluate an approach to translate S anchors to SCJ programs, we have analysed some of the larger SCJ case studies using a tool. This revealed certain aspects that we did not consider here, for instance, static methods and fields, static and instance initialiser blocks, inheritance and interfaces, parts of the SCJ API such as Clocks and Time, and the interaction with a library. For some of these points we have already sketched out solutions. A compositional and formalised translation strategy that can cope with all valid SCJ programs that fulfil the structural requirements on the class architecture is work in progress.

One important assumption of our technique is embedded in CSP and Circus: the events identified in the A anchor can be realised as atomic and virtually instantaneous interactions with the program environment. In our example, for instance, the aperiodic communication out is realised by an interrupt that gets enabled and disabled. If this assumption fails to hold, a separate argument of non-interference needs to be made; that is likely to be convoluted and non-compositional. We plan to propose a catalogue of programming patterns that satisfy our assumption.

6. CONCLUSIONS

The goal of the SCJ specification is to enable the development of safety-critical Java applications using a restricted infrastructure, in such a way that the system is amenable to certification under DO-178B, Level A, and other standards. The assumption is that a small and highly predictable virtual machine and libraries enhance certifiability and permit meeting tight performance requirements.

It is our view that the SCJ specification embeds two elements: a programming paradigm, and its implementation on a Java-based platform. Our work identifies the novel paradigm in the design of SCJ-Circus. It establishes a route to generate programs that are guaranteed to satisfy the SCJ specification. On the other hand, in following this route we lose some of the generality afforded by writing such programs
directly, taking advantage of all facilities of Java retained in SCJ, like inner classes, for example.

More experience with examples will tell us if and where we need to generalise our theories and techniques. On the other hand, patterns of specification and programming are essential to enable automation of refinement, and generalisations come at the expense of automation. A compromise needs to take into account scalability of the techniques, a central issue to be further explored in our future work.

Our work addresses correctness in the sense of CSP (and Circus): program safety (trace inclusion), liveness, and divergence-freedom, with respect to an abstract specification. We do not address, however, system safety and certification. This more general concern involves correctness, but we need to integrate our results in a certification approach.

We also do not address the issue of resources. There is nothing in our refinement technique that favours one design or another based on their use of memory or their performance. For that, we need resource-aware programming theories. They can be accommodated in the UTP, as discussed in [8], but this topic has not been further developed yet.

7. ACKNOWLEDGMENTS

This work is funded by EPSRC grant EP/H017461/1. Chris Marriott has contributed with useful discussions.

8. REFERENCES