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-

JTRES'11 September 26-28,2011, York, UK

dlers requests termination, upon which a cleanup phase is performed and the next mission is prepared. Mission execution is controlled by an application-defined sequencer.

The sequencer and the missions are implemented by subclasses of the `Mission` and `MissionSequencer` abstract classes of the SCJ API. A concrete subclass of `MissionSequencer` implements the `getNextMission()` method to determine the next mission to be run. A concrete subclass of `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 `AperiodicEventHandler`, `AperiodicLongEventHandler`, or `PeriodicEventHandler`. They override the `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 `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 cleared out at the end of each mission. Each release of a handler has an associated per-release memory area, cleared out at the end of the release. Additionally, during a release, a stack of temporary private memory areas can be used.

An account of the mission model in *Circus*, and of the memory model in the UTP, are briefly presented below.

2.2 Circus and the SCJ mission model

The key elements of *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 *Circus* process also encapsulates local state, hidden to other processes.

A process specification defines its state as a Z schema. Local actions operate on the state, while possibly interacting with the environment. The action notation is a mixture of 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 *BReqs* is in Figure 3. There, the state is defined by a schema *APState* whose single component is a collection *col* of type $\mathbb{P}$ char: it holds a set of characters. *Init* and *Insert* are local actions specified by Z operation schemas. *Init* defines the initial value of *col* to be the empty set. *Insert* adds an input $x?$ to *col*.

Additionally, *InsertS*, *BReq*1, *BReq*2 and *BReq*3 are actions that use (Timed) CSP elements, including synchronisations ($c \longrightarrow A$), input ($c\,?\,x \longrightarrow A(x)$) and output ($c\,!\,v \longrightarrow A$) prefixes, external choice ($\Box$), and recursion ($\mu\ X \bullet A(X)$). Like in CSP, interactions (that is, synchronisations, inputs, and outputs) are written as part of a prefixing operator ($\longrightarrow$). For example, in *BReq*2, we have a synchronisation on *out* prefixing an action, which is itself another prefixing with a synchronisation on *enable*. We, therefore, have that *BReq*2 is ready to synchronise on *out*, then on *enable*, then accept an input x through the channel *send*, and finally recurse.

The main action (at the end after the $\bullet$) involves two parallelims. A parallel composition ($A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$) of actions is parametrised by a synchronisation set *cs* and two disjoint sets of variables ns_1 and ns_2 that the parallel actions may modify. In (*BReq*2 $\llbracket \{\!|\ send\ |\!\} \rrbracket$ *BReq*3) we omit the name sets, as neither *BReq*2 nor *BReq*3 modify *col*; they synchronise on the channel *send*. This parallel action is, however, itself in parallel with *BReq*1, which modifies *col*. Like actions, we can also combine processes in *Circus* using the CSP operators, for example, sequentially ($P_1;\ P_2$), in parallel ($P_1 \llbracket cs \rrbracket P_2$) or in interleaving ($P_1 \,|||\, P_2$).

In *OhCircus*, we also have classes; an example is in Figure 4. These, like processes, have a state (specified in Z). Behaviour, however, is not specified by actions, but methods: data operations over the state, specified using either 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.

Circus Time includes also constructs in the style of Timed CSP. In a prefixing $c@t \longrightarrow A$, the variable t records the number of time units during which the communication c is available. In *BReq*1, for instance, $in?x@t \longrightarrow \mathbf{Skip}$ records the amount of time since the input through *in* is offered until it is taken and assigned to x. An action **wait** d terminates after d time units; also, **wait** $t_1 \,..\, t_2$ is a nondeterministic choice of a wait period d between t_1 and t_2. A timeout action $A_1 \overset{d}{\rhd} A_2$ preempts the execution of A_1 if it does not engage in some interaction in d time units, in which case the action A_2 takes over. Finally, we have two extra deadline operators: $A \blacktriangleright d$ asserts that A terminates within d times units, and $A \blacktriangleleft d$ asserts that A starts (via a visible interaction) within d time units. Data operations do not take time: any time properties need to be explicitly defined.

We have defined a *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: *Safelet*, *MissionSequencer*, *Missions*, *AperiodicHandler*, and *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 (*FW*) that captures the SCJ framework behaviour, and an application-specific process (*App*) that models a class that implements the component. There is a process for each mission and event handler.

This *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 P model in Figure 2).

2.3 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 theory 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

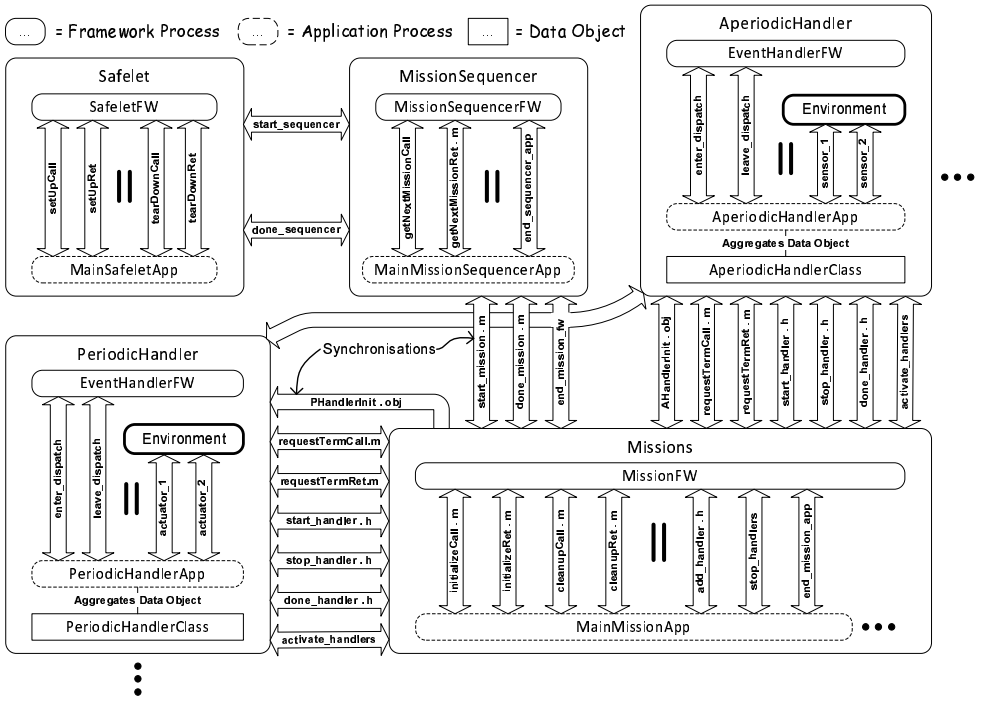

Figure 1: *Circus*-based model of an SCJ program

theory. The second is its signature: the syntax used to denote elements of the theory. The third is a set of healthiness conditions that determine membership of the theory.

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]; it 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

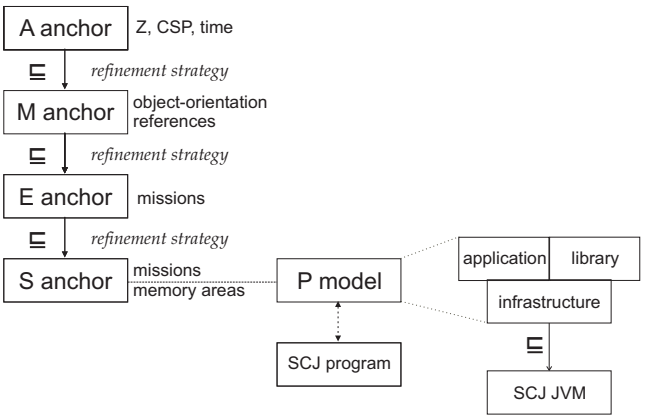

Figure 2: Our approach to development and verification

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 *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. REFINEMENT 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*

$$
\begin{array}{l}
\textbf{process}\ BReqs \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ APState == [col : \mathbb{P}\,\mathrm{char}] \\
\quad Init == [APState' \mid col' = \varnothing] \\
\quad Insert == [\Delta APState;\ x? : \mathrm{char} \mid col' = col \cup \{x?\}] \\
\quad InsertS(w) \mathrel{\widehat=} (\textbf{wait}\,0..w\ ;\ Insert)\ \Box\ (send!(\#\,col)@t \longrightarrow InsertS(w-t)) \\
\quad BReq1 \mathrel{\widehat=} (in?x@t \longrightarrow InsertS(100-t)\ \Box\ send!(\#\,col) \longrightarrow \textbf{Skip})\ ;\ BReq1 \\
\quad BReq2 \mathrel{\widehat=} out \longrightarrow enable \longrightarrow send?x \longrightarrow BReq2 \\
\quad BReq3 \mathrel{\widehat=} send?x \longrightarrow disable \longrightarrow BReq3 \\
\quad \bullet\ \textbf{wait}\,0\,..\,3\ ;\ Init\ ;\ (\ BReq1\ \ [\![\ \{col\} \mid \{\!|\ send\ |\!\} \mid \{\}\ ]\!]\ \ (\ BReq2\ \ [\![\ \{\!|\ send\ |\!\}\ ]\!]\ \ BReq3\ )\ ) \\
\textbf{end} \\
\textbf{process}\ TReqs \mathrel{\widehat=} \textbf{begin} \\
\quad TReq1 \mathrel{\widehat=} ((in?x \longrightarrow \textbf{Skip}) \blacktriangleright 5 \ |||\ \textbf{wait}\,100)\ ;\ TReq1 \\
\quad TReq2 \mathrel{\widehat=} out \longrightarrow \textbf{wait}\,0\,..\,7\ ;\ enable \longrightarrow (disable \longrightarrow \textbf{Skip}) \blacktriangleright 15\ ;\ TReq2 \\
\quad \bullet\ TReq1 \ |||\ TReq2 \\
\textbf{end} \\
\textbf{system}\ AProtocol \mathrel{\widehat=} BReqs\ [\![\ \{\!|\ in, out, enable, disable\ |\!\}\ ]\!]\ TReqs
\end{array}
$$

Figure 3: Protocol: Anchor A

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 related by refinement ($\sqsubseteq$) 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 inputs received.

3.1 Anchor A: abstract model

In this anchor, we use the basic *Circus* notation and *Circus Time*, including deadline operators. In other words, we basically use the constructs of Z, CSP, Timed CSP, and Morgan's refinement calculus. Crucially, nothing is said about classes, or objects and their allocation. An A anchor defines 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 process *BReqs* specifies the data and the interactions of the system, 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 distinguished with the keyword **system**; just like **process**, it introduces a *Circus* process. We use **system** to emphasise its role in specifying the behaviour of the system to be implemented. Semantically, there is no distinction to **process**.

There are three behavioural requirements *BReq*1, *BReq*2, and *BReq*3; 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. *BReq*1 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 parameter w of *InsertS* determines the amount of time that *Insert* can take. It can be up to $100 - t$ time units, where t is the time between the offer to input, and the actual arrival of the input. Moreover, *InsertS* specifies that, if before *Insert* concludes, 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 *BReq*2, it is defined that, if there is a request for *out*put, then we must *enable* the line, and output via *send*. *BReq*3 states that once the data is sent, the line must be *disable*d.

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 *BReq*1, *BReq*2, and *BReq*3.

The process *TReqs* introduces two timing restrictions that are interleaved ($|||$): no need to synchronise. *TReq*1 states that the input must be read within 5 time units ($\blacktriangleright 5$), and that in total 100 time units (**wait** 100) must be elapsed before an input is taken again. *TReq*2 states that if an *out*put is requested, then after at most 7 time units, the system should ask for the line to be *enable*d. Moreover, once the line is enabled, it cannot be held for more than 15 time units.

The variables *col* and x used in the UTP theories that define the main action of *BReqs*, for instance, are specification variables. They denote values, not memory locations.

class $List \mathrel{\widehat=}$

state $LState == [val : \mathrm{char};\ next : List;\ empty : Bool \mid empty = true \Leftrightarrow next = null]$

initial $Init == [LState' \mid empty' = true]$

synchronized public *insert*

$\Delta LState;\ x? : \mathrm{char}$

---

let $col == \mathbf{self}.elems();\ col' == \mathbf{self}'.elems() \bullet col' = col \cup \{x?\}$

logical $elems \mathrel{\widehat=}$ **res** $col : \mathbb{P}\,\mathrm{char} \bullet$
  if $empty = true \longrightarrow col := \varnothing$
  $[]$ $empty = false \longrightarrow col := next.elems() \cup \{val\}$
  fi

synchronized public $size == [\Xi LState;\ s! : num \mid \mathbf{let}\ col == \mathbf{self}.elems() \bullet s! = \#\, col\,]$

end

process $MBReqs \mathrel{\widehat=}$ **begin**

state $MPState == [l : List]$

$Init \mathrel{\widehat=} (l := \mathbf{new}\ List)$

$InsertS(w) \mathrel{\widehat=} (\mathbf{wait}\ 0..w\ ;\ l.insert(x))\ \Box\ (send!(l.size())@t \longrightarrow InsertS(w-t))$

$BReq1 \mathrel{\widehat=} (in?x@t \longrightarrow InsertS(100-t)\ \Box\ send!(l.size()) \longrightarrow \mathbf{Skip})\ ;\ BReq1$

...

Figure 4: Protocol: Anchor M

3.2 Anchor M: memory allocation

In this step, the target is an M 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 *BReq*1 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 $AfterInPinsert(t+u, 100-(t+u), x)$; and after a request $out@u$, by $InAfterOut(t+u, d-u, 7)$.

AfterInPinsert 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 ($\sqcap$) defines the amount of time d actually taken. Before d time units have elapsed, *AfterInPinsert* offers a synchronisation on *out*; if it happens, its behaviour is that of *AfterOutPinsert*. 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*, *AfterOutPinsert*, and *AfterIn*, and of the other

$$
\begin{array}{l}
\textbf{system}\ \mathit{EProtocol} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{MPState} == [l : \mathit{List}] \\
\quad \mathit{Init} \mathrel{\widehat=} (l := \textbf{new}\ \mathit{List}) \\
\quad \mathit{InPending}(t,d) \mathrel{\widehat=} (\mathit{in}?x@u \longrightarrow \mathit{AfterInPinsert}(t+u, 100-(t+u), x)) \blacktriangleleft d \\
\qquad\qquad\qquad\qquad \Box \\
\qquad\qquad\qquad\qquad \mathit{out}@u \longrightarrow \mathit{InAfterOut}(t+u, d-u, 7) \\
\quad \mathit{AfterInPinsert}(t, \mathit{wins}, x) \mathrel{\widehat=} \\
\qquad \bigsqcap d : 0 \mathinner{..} \mathit{wins} \bullet (\mathit{out}@u \longrightarrow \mathit{AfterOutPinsert}(t+u, d-u, 7, x)) \overset{d}{\rhd} l.\mathit{insert}(x);\ \mathit{AfterIn}(t+d) \\
\quad \ldots \\
\quad \bullet\ \textbf{wait}\, 0 \mathinner{..} 3\ ;\ \mathit{Init}\ ;\ \mathit{InPending}(0,5) \\
\textbf{end}
\end{array}
$$

Figure 5: Protocol: phase CP

$$
\begin{array}{l}
\textbf{system}\ \mathit{EProtocol} \mathrel{\widehat=} \textbf{begin} \\
\quad \ldots \\
\quad \mathit{AfterInPinsert}(t, \mathit{wins}, x) \mathrel{\widehat=} \\
\qquad \bigsqcap d : 0 \mathinner{..} \mathit{wins} \bullet (\mathit{out}@u \longrightarrow \mathit{AfterOutPinsert}(t+u, d-u, 7, x)) \overset{d}{\rhd} \mathit{insertLC}!x \longrightarrow \mathit{insertLR} \longrightarrow \mathit{AfterIn}(t+d) \\
\quad \ldots \\
\quad \mathit{System} \mathrel{\widehat=} \mathit{InPending}(0,5) \\
\quad \mathit{MArea} \mathrel{\widehat=} \left(\begin{array}{l} \textbf{var}\ l : \mathit{List} \bullet \textbf{wait}\, 0 \mathinner{..} 3\ ;\ \mathit{Init}; \\ \qquad \left(\mu\ X \bullet \left(\begin{array}{l} \mathit{insertLC}?x \longrightarrow l.\mathit{insert}(x)\ ;\ \mathit{insertLR} \longrightarrow X \\ \Box \\ \mathit{sizeLC} \longrightarrow \mathit{sizeLR}!(l.\mathit{size}(x)) \longrightarrow X \end{array} \right) \right) \end{array} \right) \\
\quad \bullet\ (\mathit{MArea} \llbracket \{\!| \mathit{insertLC}, \mathit{insertLR}, \mathit{sizeLC}, \mathit{sizeLR} |\!\} \rrbracket \mathit{System}) \setminus \{\!| \mathit{insertLC}, \mathit{insertLR}, \mathit{sizeLC}, \mathit{sizeLR} |\!\} \\
\textbf{end}
\end{array}
$$

Figure 6: Protocol: phase SH

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 $\mathit{InPending}(t, d)$ 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 (μ) 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 *AfterInPinsert*, 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 ($\setminus$). 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

$$
\begin{array}{l}
\textbf{system}\ EProtocol \mathrel{\widehat=} \textbf{begin} \\
\quad Handler1 \mathrel{\widehat=} ((in?x @ t \longrightarrow \textbf{wait}\, 0..(100 - t)\ ;\ insertLC!x \longrightarrow insertLR \longrightarrow \textbf{Skip}) \blacktriangleleft 5 \mid\mid\mid \textbf{wait}\, 100)\ ;\ Handler1 \\
\quad Handler2 \mathrel{\widehat=} \\
\qquad out \longrightarrow sizeLC \longrightarrow sizeLR?x \longrightarrow \textbf{wait}\, 0 \ldots 7\ ;\ enable \longrightarrow (send!x \longrightarrow disable \longrightarrow \textbf{Skip}) \blacktriangleright 15\ ;\ Handler2 \\
\quad Mission \mathrel{\widehat=} (Handler1 \mid\mid\mid Handler2) \\
\quad System \mathrel{\widehat=} Mission \\
\quad MArea \mathrel{\widehat=} \ldots \\
\quad \bullet\ (MArea \llbracket \{\!| insertLC, insertLR, sendLC, sendLR |\!\} \rrbracket System) \setminus \{\!| insertLC, insertLR, sendLC, sendLR |\!\} \\
\textbf{end}
\end{array}
$$

Figure 7: Protocol: Anchor E

in interleaving; our E anchor is sketched in Figure 7. We omit the definition of *MArea*, 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 *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* AR*: algorithmic refinement*

In the final Phase AR, we mostly carry out algorithmic refinement using [5]. In our case, we just need to refine the Z schemas that specify the methods *insert* and *size* in the class *List* (see Figure 4). The method *elems* is logical and can be eliminated after the refinement of *insert* and *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 S: Safety-Critical Java

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

As already explained, the S anchor can be written using the same *Circus* variant used in the M and E anchors, with the addition only of constructs to use the SCJ memory areas. Instead, however, we use paragraphs for declaration of safelets, 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 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 *List*.

Since the only *List* object l is in the *MArea* action in the E anchor, the objects created in the *insert* method are allocated in the mission memory. For that, we use assignments of the form $next := \textbf{newm}\, List$, 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 *MArea*.

As discussed in Section 2.2, a safelet, for example, is described by the parallel composition of two *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 S anchor define these parallelisms, and the complete *Circus* specification. The paragraph **safelet**, for instance, defines the `setUp` and `tearDown` methods. In our simple example, they are empty (modelled as **Skip**), and so the **safelet** paragraph is omitted. The *Circus* specification defined by the 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 **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, **getNextMission** defines the body of the `getNextMission` method, and the corresponding *Circus* action in the mission-sequencer process. The assignment to the special variable **ret** defines the mission to be returned. (The variable **ret** is an implicit result parameter of the method.)

For each mission, 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 *Handler*1 and *Handler*2. Finally, a **mission** paragraph defines the `cleanup` method. In our example, the definition is trivial: **Skip**.

The definition of a *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 *in*. The second handler is aperiodic, and performs three synchronisations on *out*, *enable*, and *disable*, and one output through *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-

sequencer *MainMissionSequencer* $\widehat{=}$ **begin**
  state *MainMissionSequencerState* == [*mission_done* : *Bool*]
  initial $\widehat{=}$ *mission_done* := *false*
  getNextMission $\widehat{=}$ **if** *mission_done* = *false* $\longrightarrow$ *mission_done* := *true*; **ret** := *ProtocolMission*
    [] *mission_done* = *true* $\longrightarrow$ **ret** := *null*
    fi
end

mission *ProtocolMission* $\widehat{=}$ **begin**
  state *MState* == [*l* : *List*]
  initialize $\widehat{=}$ *l* := **new***List* ; (**newHandler***Handler*1(*l*)) ; (**newHandler***Handler*2(*l*))
  cleanup $\widehat{=}$ **Skip**
end

periodic(100) **handler** *Handler*1 $\widehat{=}$ **begin**
  state *Handler*1_*State* == [*l* : *List*]
  initial *Handler*1_*Init* == [*Handler*1_*State*′; *list*? : *List* | *l*′ = *list*?]
  handleAsyncEvent(*x*, *w*) $\widehat{=}$ **wait** 0..*w* ; *l*.*insert*(*x*)
  dispatch (*in*?*x*@*t* $\longrightarrow$ **handleAsyncEvent**(*x*, 100 − *t*))◀5
end

aperiodic handler *Handler*2 $\widehat{=}$ **begin**
  state *Handler*2_*State* == [*l* : *List*]
  initial *Handler*2_*Init* == [*Handler*2_*State*′; *list*? : *List* | *l*′ = *list*?]
  handleAsyncEvent $\widehat{=}$
    var *size* : ℕ • *size* := *l*.*size*() ; **wait** 0 . . 7 ; *enable* $\longrightarrow$ (*send* ! *size* $\longrightarrow$ *disable* $\longrightarrow$ *Skip*)▶15
  dispatch (*out* $\longrightarrow$ **handleAsyncEvent**())
end

Figure 8: Sorter: Anchor S

ence to the list *l* in the mission area. The class constructor, defined in the **initial** paragraph, takes it as a parameter. The handler method `handleAsyncEvent` is captured by the **handleAsyncEvent** paragraph. In our example, it is just a call to the relevant method of *List*. The **dispatch** paragraph captures the behaviour in each release of the handler (typically using a call to **handleAsyncEvent**). In the underlying *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 **aperiodic** paragraph is similar in structure.

In the **safelet** and **sequencer** paragraphs, use of **new** leads to allocation in immortal memory, and in the **mission** and **initial** paragraphs of a **handler**, to allocation in the mission memory. In our example, *l* := **new** *List* in **initialize** creates *l* in mission memory. In the **handleAsyncEvent** and **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 *Circus*-based (and, therefore, UTP-based) specification: the P model in [30]. This means that we can *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 safelet, 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/~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

[1] A. Burns. The Ravenscar Profile. *Ada Letters*, XIX:49 – 52, 1999.

[2] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *FACJ*, 15(2 - 3):146 — 181, 2003.

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. Unifying Classes and Processes. *SoSyM*, 4(3):277 – 296, 2005.

[4] A. L. C. Cavalcanti, A. Wellings, and J. C. Woodcock. The Safety-critical Java Memory Model: a formal account. In *FM*, volume 6664 of *LNCS*, pages 246 – 261. Springer, 2011.

[5] A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *FACJ*, 10(3):267—289, 1999.

[6] G. Haddad, F. Hussain, and G. T. Leavens. The Design of SafeJML, A Specification Language for SCJ with Support for WCET Specification. In *JTRES*. ACM, 2010.

[7] W. Harwood, A. L. C. Cavalcanti, and J. C. P. Woodcock. A Theory of Pointers for the UTP. In *ICTAC*, volume 5160 of *LNCS*, pages 141 – 155. Springer, 2008.

[8] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[9] I. J. Hayes and M. Utting. A sequential real-time refinement calculus. *Acta Informatica*, 37(6):385 – 448, 2001.

[10] I. T. Kassios. The dynamic frames theory. *FACJ*, 23(3):267 – 288, 2011.

[11] D. Locke, B. S. Andersen, B. Brosgol, M. Fulton, T. Henties, J. J. Hunt, J. O. Nielsen, K. Nilsen, M. Schoeberl, J. Tokar, J. Vitek, and A. Wellings. *Safety Critical Java Specification, First Release 0.76*. The Open Group, UK, 2010. `jcp.org/aboutJava/communityprocess/edr/jsr302`.

[12] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[13] P. Mukherjee and V. Stavridou. Decomposition in Real-Time Safety-Critical Systems. *RTS*, 14:183 – 202, 1998.

[14] E. Olderog and H. Dierks. Decomposing Real-Time Specifications. In *COMPOS'97*, pages 465 – 489. Springer, 1998.

[15] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *FACJ*, 21(1-2):3 – 32, 2009.

[16] S. Qin, J. S. Dong, and W. N. Chin. A Semantic Foundation for TCOZ in Unifying Theories of Programming. In *FME*, volume 2805 of *LNCS*, pages 321 – 340, Springer, 2003.

[17] G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *TCS*, 58:249–261, 1988.

[18] J. Reynolds. Separation logic: a logic for shared mutable data structures. In *LICS*, pages 55 – 74. IEEE Computer Society, 2002. Invited Paper.

[19] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall, 1998.

[20] T. L. V. L. Santos, A. L. C. Cavalcanti, and A. C. A. Sampaio. Object Orientation in the UTP. In *UTP*, volume 4010 of *LNCS*, pages 18 – 37. Springer, 2006.

[21] D. Scholefield, H. Zedan, and He Jifeng. A specification-oriented semantics for the refinement of real-time systems. *TCS*, 131:219 – 241, 1994.

[22] A. Sherif, A. L. C. Cavalcanti, He Jifeng, and A. C. A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *FACJ*, 22(2):153 – 191, 2010.

[23] M. Tofte and J.-P. Talpin. Region-based memory management. *Information and Computation*, 132(2):109 – 176, 1997.

[24] K. Wei, J. C. P. Woodcock, and A. Burns. A Timed Model of *Circus* with the Reactive Design Miracle. In *SEFM*, pages 315 – 319. IEEE Computer Society, 2010.

[25] A. Wellings. *Concurrent and Real-Time Programming in Java*. Wiley, 2004.

[26] J. C. P. Woodcock. The Miracle of Reactive Programming. In *UTP*, volume 5713 of *LNCS*. Springer, 2010.

[27] J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In *IFM*, volume 2999 of *LNCS*, pages 40 – 66. Springer, 2004. Invited tutorial.

[28] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.

[29] F. Zeyda, A. L. C. Cavalcanti, and A. Wellings. A Simple Protocol - Safety-Critical Java Program and Its *Circus* Model. Technical report, 2011. Available at `www.cs.york.ac.uk/circus/techreports`.

[30] F. Zeyda, A. L. C. Cavalcanti, and A. Wellings. The Safety-critical Java Mission Model: a formal account. In *ICFEM*, LNCS, 2011.