Modelling the Haemodialysis Machine with *Circus*

Artur O. Gomes and Andrew Butterfield

Trinity College Dublin
School of Computer Science and Statistics
Lero, the Irish Software Research Centre
gomesa,butrfeld@tcd.ie

Abstract. We present a formal model of aspects of the haemodialysis machine case study using the *Circus* specification notation. We focus on building a model in which each of the software requirements (**R-1**–**36**) are represented by a *Circus* action. All of these act in concert with actions that model the collection of sensor data and the progress through the various therapy phases and activities. We then present how we model check the system using FDR. [1]

1 Introduction

This paper describes our experience in modelling the haemodialysis machine case study, that was issued for the ABZ 2016 conference[7]. We chose to do our modelling using *Circus*, a fusion of Z and CSP. We saw the case study as a way to assess how the ability to mix Z schemas with CSP-like processes would enable us to structure the model in a reasonably modular manner.

Our primary focus was on the software requirements (**R-1** through **R-36**) and our plan was to use a *Circus* process or action to model the behaviour implied by each of them. We also modelled some support services, such as clocks and sensor reading actions, as well as the control-flow prescribed for the various phases of a typical therapy session.

We make reference to the case-study document [7] using the shorthand [HMCS] or [HMCS, part X]. We present a quick overview of *Circus* in §2. We then give an overview of the approach and present some of the modelling infrastructure in §3, before describing how some of the software requirements were modelled as processes in §4, where we also discuss how they were then assembled to give the full model. An issue with *Circus* is the availability of tool support, and so we discuss in §5 how we translated our model by hand into machine-readable CSP (CSPm), so that we could use the FDR3 refinement-checker[3]. We talk about issues and inconsistencies spotting during the modelling process in §6, and then, in §7, we conclude.

[1] This work was funded by CNPq (Brazilian National Council for Scientific and Technological Development) within the Science without Borders programme, Grant No. 201857/2014-6, and partially funded by Science Foundation Ireland grant 13/RC/2094. The final publication is available at Springer via http://dx.doi.org/10.1007/978-3-319-33600-8_34.

2 Quick *Circus* Guide

Woodcock and Cavalcanti developed *Circus*[12,10], as a formalism which not only combines Z[13] and CSP[6], but also Dijkstra's guarded command language [2]. Its semantics is based on the Unifying Theories of Programming (UTP)[5] and it has a refinement calculus, developed by Oliveira[9] based on that of Morgan[8]. The thesis by Oliveria[9] is also the de-facto reference for *Circus*[2] .

A *Circus* script can be considered as a series of "paragraphs", which can be either Z paragraphs or CSP process definitions, or a hybrid mix of CSP with "commands" to produce actions. The key feature here is that *Circus* uses Z-schemas to declare variables and state invariants, and then allows CSP actions to refer to and modify those variables.

We shall present a simplified version of *Circus* here, focussing on those CSP aspects as used in this paper. We shall simply model state-changing operations by variable assignment, as proxy for the Z schema parts. In the rest of the paper we make use of proper Z schemas.

For both CSP and Circus, a typical description consists of a series of definitions of the form

$$N(v_1, v_2, \ldots, v_n) \mathrel{\widehat=} C$$

where N is a (process/action) name, the v_i are local parameters, and C is a process/action "construct" that may or may not refer to N and the v_i.

Basic building blocks include expressions over local state, and ways to describe events:

$N \in \mathit{Name}$ — Process Names
$k \in \mathit{Const}$ — Concrete Values
$v \in \mathit{Var}$ — Local Variables
$e \in \mathit{Expr}$ — Expressions over Local State
$a \in \mathit{Event}$ — Atomic Events
$c \in \mathit{Chan}$ — Event Channel
$c.k \in \mathit{Event}$ — Channel Data Event

Events are observable, atomic (they either happen "in full" or not at all) but can be composite objects. So a common idiom is to describe events — atomic ! — of the form $c.k$ which is to be interpreted as the atomic event consisting of the transfer of a value k along a channel c. A process is an entity that is willing to perform some events, but not others, depending on its current state. We consider all processes as interacting with an environment, also considered as a process. A process willing to perform an event can be said to be "offering" that event. Whether or not the event actually occurs depends on both the willingness of the environment to perform it, and the synchronisation requirements between the process and its environment.

[2] More details and publications about *Circus* can be found at `https://www.cs.york.ac.uk/circus/`

We shall first consider those constructs of *Circus* that are essentially the same as their CSP counterparts:

$$
\begin{array}{lll}
C ::= \mathbf{Skip} & & \text{— Termination} \\
\quad | \quad a \longrightarrow C & & \text{— Prefix} \\
\quad | \quad c?v \longrightarrow C & & \text{— Input} \\
\quad | \quad c!e \longrightarrow C & & \text{— Output} \\
\quad | \quad C \; ; \; C & & \text{— Sequential Composition} \\
\quad | \quad C \; [\![\; cs \;]\!] \; C & & \text{— Parallel Composition (CSP)} \\
\quad | \quad C \; ||| \; C & & \text{— Parallel Interleaving} \\
\quad | \quad C \; \Box \; C & & \text{— External Choice} \\
\quad | \quad e \;\; \& \;\; C & & \text{— Guarded Process} \\
\quad | \quad N(e_1, e_2, \ldots, e_n) & & \text{— Process Call} \\
\quad | \quad \mu \, X \bullet C & & \text{— Recursion}
\end{array}
$$

Briefly, *Skip* terminates immediately; Prefix $a \longrightarrow C$ performs event a and then behaves like C; Input $c?v \longrightarrow C$ performs a channel event $c.k$ where k is a valid value for local variable v, and then behaves like $C[k/v]$; and Output $c!e \longrightarrow C$ performs event $c.k$, where k is the current valuation of e, and then behaves like C. We also have sequential composition where $C_1 \; ; \; C_2$ behaves first like C_1 and then behaves like C_2. Another composition form is parallel, in which $C_1 \; [\![\; cs \;]\!] \; C_2$ runs both commands in parallel, synchronising on events in cs and interleaving others. External choice ($C_1 \; \Box \; C_2$) allows the environment to choose between the events offered by C_1 and C_2, so determining which runs. The guarded command $e \;\; \& \;\; C$ behaves like C if e evaluates to *true*, otherwise behaves like the canonical deadlocked process *Stop*, which is a unit for external choice.

Now we look at those constructs which have been added to CSP to produce *Circus*, as well as CSP constructs that require modification in order to "play nice" with the extended semantics.

$$
\begin{array}{lll}
C ::+ \; v := e & & \text{— Assignment} \\
\quad | \quad \mathbf{if} \; G \; [\!] \ldots [\!] \; G \; \mathbf{fi} & & \text{— Guarded Choice} \\
\quad | \quad C \; [\!|\![\; us \mid cs \mid vs \;]\!|\!] \; C & & \text{— Parallel Composition (\textit{Circus})} \\
G ::= \; e \longrightarrow C & & \text{— Guarded Command}
\end{array}
$$

We have assignment $v := e$, that updates a variable that can be local or global. We also have guarded commands ($\mathbf{if} \ldots \mathbf{fi}$) in the "Dijkstra Style"[2], where a single guarded command is denoted by $e \longrightarrow C$, e being a boolean-valued expression. The strong similarity between event prefix ($a \longrightarrow C$) and guarded command ($e \longrightarrow C$) is unfortunate, but is part of the official *Circus* syntax[9]. However, it has the same semantics as guarded processes, so, for clarity's sake, we shall use $e \;\; \& \;\; C$ in the sequel for both, to avoid confusion with prefixing.

For *Circus* we need a slightly more complicated notion of parallel composition because we have global state. In order to put C_1 and C_2 (say) in parallel, we require that the sets of variables modified by the two commands be disjoint, otherwise the parallel composition is not well-formed. We write $C_1 \; [\!|\![\; us \mid cs \mid vs \;]\!|\!] \; C_2$ to indicate that the variables modified by C_1 are contained in us, and

those modified by C_2 are in vs, with $us \cap vs = \varnothing$. As with CSP parallel, we also specify the events/channels (cs) on which both sides must synchronise. In addition, when the parallel composition starts, each side gets its own snapshot of the starting variable state, which it then subsequently uses to record its own variable updates. While either of C_1 or C_2 are still running, the state changes made by one are *not* visible to the other. At the end, once both have terminated, then their snapshots are merged to give the overall final state. This means that if C_1 wants to communicate a state change to C_2, while both are still running, then it must use input/output events to achieve this.

One key advantage that *Circus* has over CSP is its ease of handling a large collection of named state components, when most situations only require the update of a few of those components. For example, imaging a process $FIRST$ that waits for an event e and then increments a state component called s, which is one among a large number of such components, and then behaves like $NEXT$. The CSP definition of $FIRST$ and $NEXT$ would be something like the following:

$$\begin{aligned} FIRST(\dots, c, \dots) &\mathrel{\widehat=} e \longrightarrow NEXT(\dots, c+1, \dots) \\ NEXT(\dots) &\mathrel{\widehat=} \dots \end{aligned}$$

The *Circus* equivalent would be

$$\begin{aligned} FIRST &\mathrel{\widehat=} e \longrightarrow c := c + 1 \,;\; NEXT \\ NEXT &\mathrel{\widehat=} \dots \end{aligned}$$

3 Approach

In order to formalise the HD machine[4], a few decisions were made regarding the system environment, sensors and the kind of responses that are required. In this section, we present our decisions about how we deal with timing, the overall structure of the HD machine, as well as how we capture the sensing functionality, and how the system should respond to events according to the requirements.

3.1 Timing Properties

A first consideration we need to take into account is how we handle time, as there are several safety requirements for the machine that deal with time. For example, the software requirement **R-2** deals with the absence of blood flow in the machine for a period of 120 seconds. After that period is over, the machine should respond right away by stopping the blood flow and raising an alarm.

We did not feel that the timing issues in the specification warranted the complexities of using a timed variant of *Circus*. Also, the precise times are not that important for our model —rather than waiting for a model time to elapse corresponding to 120 seconds, we would simply treat It more symbolically. All we really need to be able to distinguish is between a time when haven't reached such a limit, and that limit time.

We defined a *Circus* process called *SysClock* that starts with the *ResetClock* process that initialises the *time* variable and then calls *Clock*. On its turn, *Clock* repeatedly issues *tick* events, and increments a state component called *time*, storing the current time. The current *SysClock* time is made available through the channel *getCurrentTime*.

$$Clock \mathrel{\widehat=} \mu\, X \bullet \left(\begin{array}{l} tick \longrightarrow time := time + 1 \\ |||\, getCurrentTime!time \longrightarrow \mathbf{Skip} \end{array} \right) ;\ X$$
$$ResetClock \mathrel{\widehat=} time := 0 ;\ Clock$$

Another feature we need to have in our model is a wait period in order to comply with requirements such as **R-16** that specifies a period of time between two phases of the therapy. We therefore define a *Circus* process *Wait* that counts n *tick* cycles. The variable n is decremented after each *tick* and the entire process ends when the value of n reaches zero, here defined as a **Skip** action.

$$Wait \mathrel{\widehat=} \mathbf{var}\, n : \mathbb{Z} \bullet \\ \big(\mathbf{if}\ n > 0 \longrightarrow (tick \longrightarrow Wait(n-1)) \,[]\, n = 0 \longrightarrow \mathbf{Skip}\, \mathbf{fi} \big)$$

3.2 State Components

The notion of machine state is essential in order to record key values of the various components of the system. Reading the software requirements section led us to identify over 20 state components used during the execution of the system. These are related to sensor measurements, sensor limits and switches that allows the physician to adjust the parameters for the therapy.

We also take into account some components that we decided to include in our specification as state components. These are used, for example, to register the activity and therapy phases of the HD machine. We also create records of the time in the system.

In our model, we define the Z schema *HDGenComp*, composed of the state components that we identified whilst reading the system requirements. For instance, we identify the *airVol* and *airVolLimit* parameters from **R-28**–**32**, as detailed in Section 4.

HDGenComp

$$airVolLimit : \mathbb{Z};\ airVol : \mathbb{Z};\ alarm : SWITCH;\ \ldots$$

describing with the description of many components of the system. For instance, [HMCS, Table 2] describes the rinsing parameters that are defined and entered into the machine during therapy. These parameters have specific ranges of values. We model the content of Table 2 as the *RinsingParameters* schema with its components. Moreover, we define the value ranges of the components as state invariants. For example, the range for the *Filling BP rate* is $0 - 6000mL$ and is modelled in Z as the state variable *fillingBPRate*, with an invariant of

$fillingBPRate \in \{ x : \mathbb{Z} \bullet 0 \leq x \leq 6000\}$. The overall state component *RinsingParameters* is illustrated below.

RinsingParameters

$fillingBPRate : \mathbb{Z};\ rinsingBPRate : \mathbb{Z};\ \ldots$

$fillingBPRate \in \{ x : \mathbb{Z} \mid 0 \leq x \leq 6000\}$
$rinsingBPRate \in \{ x : \mathbb{Z} \mid 50 \leq x \leq 300\}$
$\ldots$

The entire state of the HD machine is modelled as the schema *HDState*, which is composed by the above described *HDGenComp* schema, along with the *RinsingParameters* and all the other schemas, *DFParameters*, such as *UFParameters*, *PressureParameters*, and *HeparinParameters*, modelling the variables detailed in the [HMCS, Tables 3–6].

$$HDState \mathrel{\widehat=} HDGenComp \land RinsingParameters \land DFParameters \land UFParameters \land PressureParameters \land HeparinParameters$$

We also initialise the *HDGenComp* components modelled as the *HDGenCompInit* schema that basically sets the numerical values to zero, along with switching off the alarm and closing sensors and tubes.

$$HDGenCompInit \mathrel{\widehat=} [\Delta HDState \mid airVolLimit' = 0 \land airVol' = 0 \land \ldots]$$

As part of our HD machine model, we want to detect the values from the various sensors in the system and update these into the state components described above. In order to achieve that task, we define the *Circus* process *SensorReadings* that watches a number of *Circus* channels each of them responsible for sensing a specific value arising from the machine sensors. These values are then stored in the state components, as described by *HDGenComp*.

$$\begin{aligned} &SensorReadings \mathrel{\widehat=} \\ &\quad senApTransdPress?apTransdPress \longrightarrow SensorReadings \\ &\quad \Box\ senInfVol?infVol \longrightarrow SensorReadings \\ &\quad \Box\ \ldots \end{aligned}$$

In addition, this process also makes the current recorded readings available

$$\begin{aligned} &SensorReadings \mathrel{\widehat=} \\ &\quad \ldots \\ &\quad \Box\ repApTransdPress!apTransdPress \longrightarrow SensorReadings \\ &\quad \Box\ repInfVol!infVol \longrightarrow SensorReadings \\ &\quad \Box\ \ldots \end{aligned}$$

The process *SensorReadings* is basically a large external choice over all sensor readings and reading reports, that repeats endlessly.

This approach is fine for sensor readings for which the time at which they occur is not important and so we are happy for an interested process to poll the relevant state component at regular intervals. Some sensor readings require some form of timestamping with possible timeouts, and some of these are handled separately, as explained later.

In order to capture the transition between the therapy phases of the HD machine, we provide a *Circus* process called *StatePhase*, that changes the value of the state variable *hdMachineState* depending on signals received during the therapy. Basically, when the process for a specific therapy phase begins, it immediately uses a special event to announce its commencement. Process *StatePhase* monitors these and updates state variables accordingly. For example after a signal *preparationPhase*, produced by the *Circus* process *TherapyPreparation*, the state variable *hdMachineState* is changed to *prepPhase* and will be used by the software requirements as described in the Section 4.

$$
\begin{array}{l}
StatePhase \mathrel{\widehat=} \\
\qquad \mu\, X \bullet \left(\begin{array}{l}
preparationPhase \longrightarrow hdMachineState := prepPhase \\
\Box connectingToPatient \longrightarrow \\
\qquad\qquad hdMachineState := connectThePatient \\
\Box therapyInitiation \longrightarrow hdMachineState := initPhase \\
\Box therapyEnding \longrightarrow hdMachineState := endPhase
\end{array} \right) \; ; \; X
\end{array}
$$

3.3 Response to the requirements

Whilst modelling the software requirements, we were able to identify twelve different kinds of behaviours that are expected among the 36 listed requirements. For each of them, we needed to provide an action that is equivalent to the intended behaviour of the requirement. For example, the expected behaviour for the requirement **R-1** to be satisfied is to stop the blood flow and raise an alarm. We formalise those two responses as Z schemas *StopBloodFlow* and *RaiseAlarm*: the former produces a signal *stopBloodFlow* stopping the current flow and the latter sets the *alarm* to *ENABLED* and then triggers the buzzer of the system.

$$
\begin{array}{l}
StopBloodFlow \mathrel{\widehat=} stopBloodFlow \longrightarrow \mathbf{Skip} \\
RaiseAlarm \mathrel{\widehat=} [\, \Delta\, HDState \mid alarm' = ENABLED \,]; \\
\qquad\qquad produceAlarmSound \longrightarrow \mathbf{Skip}
\end{array}
$$

This two process capture a common behaviour when a requirement error condition arises.

4 Model Development

In this section we give details of the modelling that resulted from our chosen approach: namely to model each of the safety requirements **R-1** to **R-36** as

a *Circus* Action that "enforces" that requirement. It allows us to show how the sensors are integrated into our model and how the system should behave accordingly. We then present an overview of the therapy phases of the machine, describing how we structure our model with respect to the activities performed.

4.1 Software Requirements

A first example of how we model the requirement is illustrated by the requirement **R-1**. According to the description, during the application of arterial bolus, the system monitors the volume of saline infusion and if the volume exceeds 400ml, the system should stop the blood flow and raise an alarm signal.

$$
\begin{array}{|l}
\hline
PreR1 \\
\Delta\, HDState \\
\hline
hdActivity \in \{applicationArterialBolus\} \land infSalineVol > 400 \\
\hline
\end{array}
$$

We define a schema $PreR1$ for the alarm-state precondition, and if it is satisfied, the system will perform *StopBloodFlow* and *RaiseAlarm*. If the precondition is not satisfied ($\neg\, PreR1$), it waits for a defined period of time (parameter *CheckInterval*) and checks again. This illustrates the general approach here for many of these monitoring requirements. They check state variables at regular intervals and raise alarms if required. This decouples them from the process of doing sensor readings and recording the results.

$$
\begin{aligned}
R1 \mathrel{\widehat=} & (PreR1 \;;\; (StopBloodFlow \;|||\; RaiseAlarm)) \\
& \lor \neg\, PreR1 \;;\; Wait(CheckInterval) \;;\; R1
\end{aligned}
$$

The second software requirement, **R-2**, monitors the blood flow and in the event that no flow is detected for a period longer than 120 seconds, the system should raise an alarm and stop. We formalise the requirement with help of two interleaved processes, *NoFlowWatchDog* and *BloodFlowSample*.

$$
R2 \mathrel{\widehat=} NoFlowWatchDog \;|||\; BloodFlowSample
$$

The former process monitors the time interval during which no blood flow occurs. if this exceeds the timeout, then the system will stop the blood pump *StopBP* and *RaiseAlarm*.

$$
\begin{aligned}
& NoFlowWatchDog \mathrel{\widehat=} getCurrentTime?time \longrightarrow \\
& time - lastNonZeroBF > 120000 \;\&\; tick \longrightarrow StopBP \;;\; RaiseAlarm \\
& \Box\, time - lastNonZeroBF \leq 120000 \;\&\; tick \longrightarrow NoFlowWatchDog
\end{aligned}
$$

The second helper process is *BloodFlowSample*, which monitors the blood flow arising from the channel *senBloodFlowInEBC*. Whenever the value of *bloodFlowInEBC* is different, the state component *lastNonZeroBF* is updated

with the current time in the system.

$$
\begin{array}{l}
BloodFlowSample \mathrel{\widehat=} getCurrentTime?time \longrightarrow \\
\quad senBloodFlowInEBC?bloodFlowInEBC \longrightarrow \\
\left(\begin{array}{l} \left(\begin{array}{l} \textbf{if } bloodFlowInEBC \neq 0 \ \ \& \ \ lastNonZeroBF := time \\ [] \, bloodFlowInEBC = 0 \ \ \& \ \ \textbf{Skip fi} \end{array} \right) ; \\ BloodFlowSample \end{array} \right)
\end{array}
$$

This is an example of a sensor reading that is not handled by *SensorReadings*, because it needs to record a timestamp for the most recent non-zero reading.

Requirement **R-9** is an example of how we create helper processes in order to capture the intended behaviour of the system. During the phase *connecting the patient*, the machine should monitor the pressure at the VP transducer and if the value measured exceeds 450mmHg for more than 3 seconds, the machine should respond by stopping the blood pump and raising an alarm signal.

In our model, we first create a helper process *TrackTimervpTransdPressR*9 that monitors such pressure values through the channel *senvpTransdPress*, after one *tick* event and updates the timer interval with the following condition: when the sensed VP transducer pressure is higher than 450, the timer interval is incremented; otherwise, the timer is reset until the condition is satisfied again.

$$
\begin{array}{l}
TrackTimervpTransdPressR9 \mathrel{\widehat=} \\
\quad \left(\begin{array}{l} \left(\begin{array}{l} tick \longrightarrow senvpTransdPress?x \longrightarrow \\ \left(\begin{array}{l} \textbf{if } x > 450 \ \ \& \ \ timerIntervalR9 := timerIntervalR9 + 1 \\ [] \, x \leq 450 \ \ \& \ \ timerIntervalR9 := 0 \, \textbf{fi} \end{array} \right) \end{array} \right) ; \\ TrackTimervpTransdPressR9 \end{array} \right)
\end{array}
$$

The next step is to define a Z schema *PreR*9, in which we define the alarm precondition for the requirement itself. The requirement is specified for use during the initiation phase and is satisfied if the value of *vpTransdPress* is higher than 450 for a period of 3 seconds, captured by the *timerIntervalR*9 state variable with a value higher than 3000 ms.

*PreR*9

$$
\begin{array}{|l}
\hline
\Delta HDState \\
\hline
hdMachineState \in \{connectThePatient\} \\
vpTransdPress > 450 \land timerIntervalR9 > 3000 \\
\hline
\end{array}
$$

Should the precondition of the **R-9** requirement be satisfied, the system does *StopBP* and *RaiseAlarm*. Otherwise, it waits for a predefined time interval before checking again.

$$
\begin{array}{l}
R9 \mathrel{\widehat=} \left(\mu X \bullet \left(\begin{array}{l} (PreR9 \ ; \ (StopBP \ ||| \ RaiseAlarm)) \\ \lor \neg \, PreR9 \, ; \ Wait(CheckInterval) \, ; \ X \end{array} \right) \right) \\
\qquad ||| \, TrackTimervpTransdPressR9
\end{array}
$$

During the *connecting the patient* phase, **R-16** specifies a time interval of 310 seconds for that phase. When the specified time ends, the machine should change to

the *initiation phase*. We formalise that requirement through a signal *conToPatient*, followed by a wait period of 310000 ticks, and ended with a signal *therapyInit* that is triggered at the beginning of that phase.

$$R16 \mathrel{\widehat=} conToPatient \longrightarrow Wait(310000) \ ; \ therapyInit \longrightarrow \mathbf{Skip}$$

4.2 Therapy Processes

We now describe how we model the therapy phases of the HD machine—the top level "workflow", so to speak. According to the requirements, the system starts with the preparation phase, followed by the initiation phase, and an ending phase:

$$MainTherapy \mathrel{\widehat=} TherapyPreparation \ ; \ TherapyInitiation \ ; \ TherapyEnding$$

The initiation phase also contains the "perform therapy" phase for some reason that is unclear to us[3] . Each of these phases is further broken down.

For the therapy preparation phase, we define a *Circus* process that starts with a signal *preparationPhase*, followed by a sequence of activities according to [HMCS,§3.2]. A key idea here is each phase signals that it has started, on a channel, so that requirements and activities that are phase-dependent can ascertain when they should be active.

We capture the steps that compose the therapy preparation phase, each one, with a *Circus* process that behaves accordingly.

$$\begin{array}{l} TherapyPreparation \mathrel{\widehat=} \\ \quad preparationPhase \longrightarrow AutomatedSelfTest; \\ \quad ConnectingTheConcentrate \ ; \ SetRinsingParameters; \\ \quad InsertingRinsingTestingTubSystem \ ; \ PrepHeparinPump; \\ \quad SetTreatParameters \ ; \ RinsingDialyzer \end{array}$$

A similar pattern is used for the other phases.

As an example of a phase activity, we show the specification *SetRinsingParameters*, which collects parameter settings from the clinician, here modelled as a Z schema with inputs:

SetRinsingParameters

$$\begin{array}{l} setFBPRate? : \mathbb{Z}; \ setRBPRate? : \mathbb{Z} \\ setRTime? : \mathbb{Z}; \ setUFRFRinsing? : \mathbb{Z} \\ setUFVFRinsing? : \mathbb{Z}; \ setBFFCPatient? : \mathbb{Z} \\ \Delta HDState \\ \hline fillingBPRate' = setFBPRate? \land rinsingBPRate' = setRBPRate? \\ rinsingTime' = setRTime? \land ufVolForRinsing' = setUFVFRinsing? \\ ufRateForRinsing' = setUFRFRinsing? \\ bloodFlowForConnectingPatient' = setBFFCPatient? \end{array}$$

[3] Particularly, because it should be the phase that lasts longest!

4.3 Putting it all together

We conclude this section by detailing how we put all the pieces together. We start the main process of the HD machine, the *HDMachine* process, with the schema *HDGenCompInit* that initialises the state variables of the system. Then, the system is modelled as a parallelism between the *MainTherapy* process and the *SoftwareRequirements* process.

A second parallelism is required between the above and the *StatePhase* Circus process. The latter is a process that watches the changes of state, through signals like *preparationPhase* and *therapyInitiation*, and after these, the state variable *hdMachineState* are set accordingly, in order to be used by the requirements defined for the *SoftwareRequirements* process. Then the components are put in parallel with the *SensorReadings* process, used for example, to update the values of the state components.

$$
\begin{array}{l}
HDMachine \mathrel{\widehat=} HDGenCompInit; \\
\left(\left(\left(\begin{array}{l} MainTherapy \\ [\![HDGenCompStChanSet]\!] \; SoftwareRequirements \end{array} \right) \right) \right. \\
\left. \begin{array}{l} [\![TherapyPhaseChanSet]\!] \; StatePhase \\ [\![SensorReadingsComm]\!] \; SensorReadings \end{array} \right)
\end{array}
$$

Finally, the entire system is put in parallel with the *SysClock* process, synchronising on the channels *tick* and *getCurrentTime*, denoting the time elapsed, and the output of the time value for the rest of the therapy, respectively.

$$HDMachine \; [\![\{\!| tock, getCurrentTime |\!\}]\!] \; SysClock$$

5 Checking the Model

Currently, there is limited tool support for Circus, and nothing that can be used for direct model-checking of machine-readable Circus. Limited support can found as part of the Community Z Tools (CZT) project[1], as extensions to the Z support there. This is facilitated by the way that the machine-readable syntax of Circus takes the form of LaTeX documents in that same way as that of Z.

We were able to use the Circus extension to CZT, which include a parser and type-checker to assess our model. After minor revisions correcting typos and small type errors, we obtained a model that satisfied both the parser and type-checker.

The current approach to model-checking Circus is to translate it into machine-readable CSP (CSP_M), and use FDR3[3] to do the model checking. Unfortunately, there is no automated way to do this, so such translations have to be done by hand. Fully-, or even semi-, automatic translation from Circus to CPSm is difficult, but is an active research topic. FDR3 itself is described as a refinement checker, which basically means is that it is a model checker, where the models are labelled transition systems derived from the operational semantics of CSP, and the properties to be checked are assertions about the existence of a refinement relation between two distinct models, one for the specification, the other for the implementation.

Manual Translation We manually translated our *Circus* specification into CSP_M, in order to do some basic checks, particularly regarding deadlock freedom. Here we give a brief description of the translation and the challenges we encountered, most notably that of avoiding state-space explosion.

For most of the *Circus* constructs, we have a pretty straightforward translation into CSP_M, as we know that *Circus* is derived from CSP. For example, a lot of the type and channel declarations are very simple, so the following *Circus* fragment:

$$STATEPHASE ::= connectThePatient \mid initPhase \mid prepPhase \mid endPhase$$
$$\textbf{channel}\ preparationPhase, therapyInitiation$$

would become the following CSPM fragment:

```
datatype STATEPHASE
  = connectThePatient | initPhase | prepPhase | endPhase
channel preparationPhase, therapyInitiation
```

We then translate Z schemas into CSP_M as tuples. Each component of a Z schema is translated as a nametype represented by a tuple. For example, the following state schema fragment

RinsingParameters

$$fillingBPRate : \mathbb{Z};\ rinsingBPRate : \mathbb{Z}$$
$$fillingBPRate \in \{0 \,..\, 6000\} \land rinsingBPRate \in \{50 \,..\, 300\}$$

is translated into a tuple where, for example, the first component of the tuple is the range of values for the *fillingBPRate*, of type $\mathbb{N}$, restricted to the values 0 up to 6000.

```
nametype RinsingParameters = ({0..6000},{50..300})
```

5.1 Translation of *Circus* processes containing state

In *Circus*, when we want to manipulate the values of a component of a state component, we can freely access it, even as an assignment, as it is within the context of the *Circus* process. However that is not possible in CSP_M.

We need to adopt a different approach for the translation. Basically we concert each state schema into a CSP process that has *get* and *set* events for each state component, so that, for example, the *Circus* action

$$x := y + 1$$

would become something like

```
getY?myY -> setX!(myY+1) -> Skip
```

All the actions accessing global state would run in parallel synchronising on the relevant *get* and *set* events.

5.2 Checking of the model using FDR

We checked our CSP_M model of the HD Machine using FDR3, with assertions regarding deadlock and livelock freedom. These helped us re-factor our model, mainly to avoid deadlocks that occurred because of errors in specifying synchronisation events. As a result of this we are very confident that our *Circus* model is deadlock-free.

A big issue we had to deal with was the fact that our original *Circus* model had a huge state-space: we had a clock that ticked every millisecond, together with a timeout in one of the requirements of 310 seconds. We also had checks for values in large numeric ranges, typically for fluid volumes. We had to carefully decide how to shrink the state-space by reducing the range of values as low as possible without having an impact on the integrity of the model.

Fortunately, most of the uses for numbers are to specify limits outside of which special action needs to be taken. Also, in many cases the number values are specific to just one requirement or a small coherent group. This makes is easy to shrink the range of values for one such requirement without worrying about its effect on another.

For example, **R-20–21** talks about measures that span the range of normal human body temperature, but are only explicit about two boundaries, one at $33^\circ C$, the other at $41^\circ C$. Error conditions arise if the temperature outside those bounds. This defines three regions of interest, but we do not need to model temperature values in the range $32 \ldots 42$ (say), but simply have three values that denote: "too cold", "too hot" and "just right".

We were able to check small clusters of requirement processes against the full machine model, and show their interaction was deadlock free. Even running the tests on a virtual machine cluster with 16 cores and 32GB of RAM we found that we handle at most about 3 requirements at a time. However as they are all independent, it didn't prevent us form checking them all.

Further ways to attempt overcome the state-space problem are described by Roscoe *et al.* [11], suggesting the use of compression techniques in order to model-check larger CSP specifications in FDR, allowing the reduction of both the number of states and the transitions to be visited. We will explore this as part of future work.

5.3 Back-annotation

Where analysis with FDR3 exposed any structural issues, we modified the *Circus* version, as would be expected. However we have not, at this stage, made the changes to number ranges needed to make model-checking feasible. Simply put, since there are no model-checkers for *Circus*, or automated translation to any other modelling notation such as CSP_M, we felt that we would keep the "full story" in the model. Clearly this would need to be addressed should automatic checking become possible.

6 Observations

One of the often-touted advantages of building formal models is that the rigour, level of detail and completeness that they require, results in the exposure of a lot of ambiguities and incompletenesses in the informal requirements and specifications. Here we collect a number of such issues that arose as part of our rigorous, detailed analysis of the case study.

We did look at the safety requirements and future work will look at formalising those with a view of being able to check the requirements model against the safety one. We shall start with a few observations regarding safety:

- **S-4** talks about draining saline solution to a bag/bucket attached to the venous connector. Presumably this means the patient isn't connected here. But this is when the patient is connected to EBC, and **S-1** requires both arterial and venous connectors are connected simultaneously.
- **S-5** makes a very ambiguous use of the phrase "can be connected". This can refer to a state, of being connected (up) to something or to a process, that involves making the connection with something (which will of course result in being in the state of being connected). We believe that what **S-5** intended to state was that the process (connecting to) could only occur during initiation, but that the state (now connected to) would continue to hold during the main therapy portion.
- **S-7** talks about power-loss. We are given no information about how the machine might cope, or the hardware's power-down state. How might an alarm be raised without power? Also, surely **S-7** should be about blood flow stopping for any reason, not just power failure?
- **S-9** talks about the difference between actual and measured blood flow, mentioning low or negative AP as an issue. Is there a well-defined relationship linking actual flow to measured flow and AP?

Next, issues that arose while looking at requirements and activities:

- **State Components:** We identified a bunch of parameters and sensors that are described in the *software requirements* section. Some of these are not mentioned elsewhere in the entire text and therefore we do not know what the restrictions are regarding expected values for them. We modelled these items based on our limited understanding, defining types for each of them. These are modelled as components of the *HDGenComp* schema, as part of the *HDState*. For instance, the pressure at the VP transducer and AP transducer are modelled as *vpTransdPress* and *apTransdPress* respectively.
- **Alarm:** In several requirements, the expected behaviour of the system is to raise an alarm. However, we don't know precisely what is the overall behaviour of the system for most of these requirements. Once the alarm is raised, what is the expected behaviour of the system? Does the system stop entirely until the alarm is acknowledged, or are some functions not available?
- **R-35** is not formalised in our current version of the HD machine model. There seems to be a safety issue in this requirement. The system *"shall*

monitor the net fluid removal volume and if the net fluid removal volume exceeds (UF set volume + 200 mL)", the machine goes into bypass and the alarm is raised. However, once the alarm is acknowledged by the user, the requirement says that *"the software shall increase the UF set volume by 200 mL"*. Is it really the intention to set a limit monitored by an alarm where the response mandate for the user is to raise the limit to the level were the alarm signal is (just) disabled?

7 Conclusions

We started this *Circus* case-study as a response to the HD machine case study, proposed for the ABZ 2016 conference[7]. We summarise our approach thus: we capture the communication between the sensors and the system and also define a structure for the data used around the system; we make use of Z schemas in order to define the model state and also operations that change that state, all related to the various sensor readings and parameters defined for the therapy.

We have around a thousand lines of *Circus* specification for the model, with about sixty state variables and forty events, and we have around ninety processes/actions used for modelling the requirements and the overall therapy procedures. We found that the ability to read and write small state components by name (using small Z schemas and assignment) coupled with the usual ability to structure CSP as small parallel processes made it easy to restrict any formal text to only parts of the system that were immediately relevant.

In our investigation, we were able to model almost all the software requirements with exception of **R-35** whose description leads us to see a contradiction that may be a safety issue, as discussed in Section 6.

Due to the current lack of tool support for direct checking of *Circus* specifications, we needed to translate our model, by hand, into CSP in order to be able to perform model checking using FDR3[3]. In our translation, we had to adapt the *Circus* model for CSP_M because *Circus* programs has explicit state-based features (such as assignment), which are not present in CSP_M, which instead relies on process parameter-lists to handle state. The equivalent specification written in CSP_Mhas around 24 hundred lines, more than twice the size of the *Circus* version, due to the inclusion of auxiliary functions and new channels for communicating with the new state-modelling processes. We were able to perform model checking using FD3R and perform checks that the requirements could run in parallel with the therapy model, synchronising on common events, without any deadlocks.

For future work, we intend to build the corresponding model of the safety requirements, and link it to the requirements model in order to look for inconsistencies. Other possible avenues of investigation would include deriving a formal software specification structured around the architecture and functional decomposition of a realistic implementation design. We would expect this to be structured differently to the model we have just derived from the require-

ments, and it would raise interesting issues regarding their refinement relation and verification.

Another interesting piece of future work would be to derive software prototypes from our *Circus* model to allow us to simulate the system execution. In the long term, we have interest in working with theorem proving for the refinement of *Circus* programs and possible proofs of test-cases. It is also in our plans to explore the development of tools that allows us to perform model-checking directly with *Circus* programs.

Acknowledgments

We would like to thank Thomas Gibson-Robinson for his help in assisting us in achieving the state-space reduction we needed, and the anonymous reviewers for their perceptive comments and pointed questions, which have help to improve this paper. Finally we re-iterate our thanks to our sponsors, CNPq of Brazil, and Science Foundation Ireland.

References

1. Community Z Tools Project: CZT: Community Z Tools (Sep 2015), `http://czt.sourceforge.net/manual.html`, checked Mar 14th, 2016
2. Dijkstra, E.W.: Guarded Commands, Nondeterminacy and Formal Derivation of Programs. Commun. ACM 18, 453–457 (August 1975)
3. Gibson-Robinson, T., Armstrong, P., Boulgakov, A., Roscoe, A.W.: FDR3 — A Modern Refinement Checker for CSP. In: TACAS 2014. Proceedings (2014)
4. Gomes, A.O., Butterfield, A.: HD-Machine Case Study Repository (2016), `https://bitbucket.org/artur1109/hdmachine/`
5. He, J., Hoare, C.A.R.: Unifying Theories of Programming. In: Orlowska, E., Szalas, A. (eds.) RelMiCS. pp. 97–99 (1998)
6. Hoare, C.A.R.: Communicating Sequential Processes. Computer Science, Prentice-Hall International, Englewood Cliffs, N.J (1985)
7. Mashkoor, A.: The Haemodialysis Machine Case Study. Software Competence Center Hagenberg GmbH (SCCH) (2015), `http://www.cdcc.faw.jku.at/ABZ2016/HD-CaseStudy.pdf`
8. Morgan, C.C.: Programming From Specifications, 2nd Edition. Prentice Hall International series in computer science, Prentice Hall (1994)
9. Oliveira, M.V.M.: Formal Derivation of State-Rich Reactive Programs using *Circus*. Ph.D. thesis, Department of Computer Science - University of York, UK (2005)
10. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP semantics for *Circus*. Formal Asp. Comput. 21(1-2), 3–32 (2009)
11. Roscoe, A.W., Gardiner, P.H.B., et al.: Hierarchical Compression for Model-Checking CSP or How to Check 1020 Dining Philosophers for Deadlock. In: TACAS. pp. 133–152. Springer (1995)
12. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B. pp. 184–203. ZB '02, Springer-Verlag, London, UK, UK (2002)
13. Woodcock, J., Davies, J.: Using Z, Specification, Refinement, and Proof. Prentice Hall International Series in Computer Science, Prentice Hall (1996)