Simulink Timed Models for Program Verification

Ana Cavalcanti[1], Alexandre Mota[2], and Jim Woodcock[1]

[1] Department of Computer Science, University of York, York, YO10 5DD, England
[2] Centro de Informática, Universidade Federal de Pernambuco, Brazil

Abstract. Simulink is widely used by engineers to provide graphical specifications of control laws; its frequent use to specify safety-critical systems has motivated work on formal modelling and analysis of Simulink diagrams. The work that we present here is complementary: it targets verification of implementations by providing a refinement-based model. We use *CircusTime*, a timed version of the *Circus* notation that combines Z, CSP, and Morgan's refinement calculus with a time model, and which is firmly based on Hoare & He's Unifying Theories of Programming. We present a modelling approach that formalises the simulation time model that is routinely used for analysis. It is distinctive in that we use a refinement-based notation and capture functionality, concurrency, and time. The models produced in this way, however, are not useful for program verification, due to an idealised simulation time model; therefore, we describe how such models can be used to construct more realistic models. This novel modelling approach caters for assumptions about the programming environment, and clearly establishes the relationship between the simulation and implementation models.

Keywords: Simulink, Z, CSP, *Circus*, time, refinement, modelling.

On the occasion of He Jifeng's 70th birthday.

1 Introduction

The use of Simulink diagrams [17] for the specification of control laws is pervasive in industry. Various approaches enrich the current Simulink facilities for analysis of diagrams with techniques based on formal methods [15, 2, 6]. Denney & Fischer [7], for example, propose the use of the AutoCert verification system to construct a natural language report explaining where code uses specified assumptions and why and how it complies with requirements (though, significantly, not for timing aspects). In contrast, our work recognises the need to verify implementations. Automatic code generation does not usually provide enough assurance: even when generators are reliable, restrictions on performance or resources often require changes to the code.

In previous work [4, 3], we covered functional and behavioural aspects of diagrams and implementations. We cater for the inherent parallelism in diagrams and the verification of complete programs, including their scheduling components. For modelling and reasoning, we use *Circus* [20], a flexible integration of

Z [30], CSP [23], and Morgan's refinement calculus [18], with formal foundations underpinned by Hoare & He's Unifying Theories of Programming (UTP) [13]. *Circus* is a mature notation with a sound semantics implemented in tools: it was first introduced in 2001 in [28], given a formal semantics in 2002 [29], and subsequently mechanised in ProofPowerZ, a HOL-based theorem prover, in [20, 21], and in Isabelle/HOL [8, 9].

We generate formal models automatically, and apply a refinement tactic in ProofPowerZ to prove that the model of the program conforms to (refines) the model of the diagram. Automation is enabled by knowledge of the structure of the automatically generated models, and of the correspondence between diagram and program components.

What we have not covered before is the time model embedded in the diagrams. In [4, 3], we use synchronisation to model the cycles of the diagram, which are in fact defined by simulation time parameters. In this approach, we cannot cater for multi-rate diagrams and, most importantly, have to consider partial program models that do not capture the use of timing primitives (like delay commands). To produce a model of a program, we consider a slice that removes all variables related to time control; this can potentially mask an error.

In this paper we present a novel modelling approach to cover the time properties of diagrams. Our approach uses *CircusTime* [27], a timed version of *Circus* with both timeout and deadline operators. *CircusTime* was first introduced in 2002 in [25], and given a complete formal semantics in UTP in [26, 27].

In our new approach, we capture the idealised-time model adopted in the Simulink simulator as well as its data-flow model, which embeds some calculations (functional properties) and concurrent behaviour. Since we are interested in software implementations, we consider only diagrams with a discrete-time model; but we can cover multi-rate diagrams as well.

The idealised-time model of the simulation is not implementable, since it involves infinitely fast computations. So we also provide a realistic model used by typical implementations that run on real-time computers. This programming model embeds assumptions about the environment; in particular, we consider the assumptions adopted in the standard Simulink code generator, but our approach can be adapted for different real-time computers. The timed programming model is the appropriate starting point for the verification of programs.

The programming model is written in terms of the simulation model, so that we formalise the way in which the assumptions made about the programming environment affect the simulation model. Engineers use the simulation model in the analysis and validation of diagrams and corresponding control laws, so it is important to understand the way in which it is reflected in programs. Additionally, the *CircusTime* model that captures the Simulink idealised-time model is in direct correspondence with the informal description of the simulator. Its use to define the programming model increases our confidence in its validity.

With the use of *CircusTime*, we can provide very faithful models of the diagram and of the assumptions about the environment; it is also possible to model programs in a direct way. All this reduces the risk of introducing modelling errors that compromise verification.

Ultimately, the simulation time model is defined by the solver, a component of the simulator that determines the simulation steps. For simplicity, we consider a fixed-step solver, where the step size is constant; this is the solver used to generate code for a real-time computer. It is not difficult to generalise our model for a variable-step solver. In this case, the step size changes with time, so that steps that do not present any changes to the output are omitted.

In the next section we give a brief overview of Simulink diagrams, *Circus*, and *CircusTime*. In Section 3, we present our approach to construct timed simulation models. The programming models are discussed in Section 4. Finally, in Section 5, we draw our conclusions and discuss related and future work.

2 Simulink, *Circus*, and *CircusTime*

This section describes the modelling notations used in our work.

2.1 Simulink

A control law diagram is a graph whose nodes are blocks that embed an input, an output, or some computation, and whose edges are wires that connect the input and output ports of the blocks to define data flow. The behavioural model embedded in a diagram is a cyclic execution, where in each iteration inputs are read and outputs are calculated and produced.

At the top of Figure 1, we present the Simulink diagram for a PID controller. (This is the same example used in [4, 3].) The rounded boxes are input and output blocks that represent the inputs and outputs of the system. Inputs are represented by outputs of input blocks, which work in the same way as any other block. Outputs are the inputs of the output blocks. In Figure 1, we have inputs E, Kp, Ki, and Kd, and output Y. The rectangles and circles are blocks that define particular calculations. The Sp block, for instance, is a simple multiplication.

Subsystem blocks are defined by other diagrams. In Figure 1, the blocks Diff and Int are defined by the diagrams at the bottom of Figure 1. A control law for a real system may reach hundreds of pages of hierarchical diagrams.

Blocks can have state. A Unit Delay, used in the diagrams in Figure 1, for instance, records in its state the value of its last input.

A block also has a sample time, characterised by a sampling period and an offset, which indicates when, during simulation, it produces outputs and updates its state, if any. Additionally, each port can have a different sample time. If the blocks do not all have the same sample time, we have a multi-rate diagram.

A simulation is described by a series of steps; a solver determines the time of the next simulation step. We assume that the default step size and offset determined by the fixed-step solver are used, since they guarantee that all sample times are covered. The default step size is the greatest common divisor of all

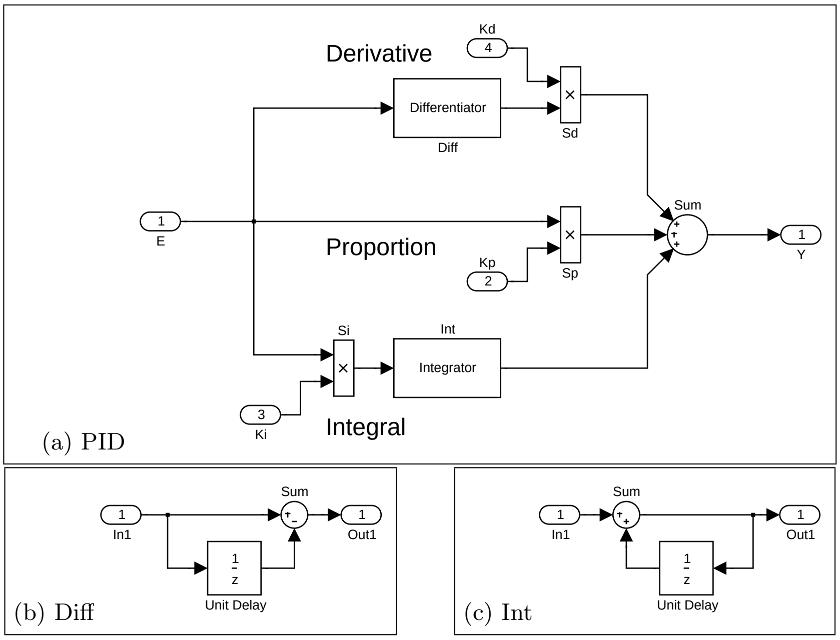

Fig. 1. PID (Proportional Integral Derivative) controller

sample times. A block's state is updated and a new output is generated only when the time of a step is a multiple of the sample time of the block.

In Section 3, we formalise this simulation time model using our novel approach based on *CircusTime*, which we describe next.

2.2 *Circus* and *CircusTime*

A *Circus* model is a sequence of definitions: Z paragraphs, channel declarations, and process definitions. Several examples are presented in the next section (see Figures 5, 6, 8, 2, and 3). After a simple example in this section, we explain details of the notation as it is used.

$$PIDTSpec \mathrel{\widehat{=}} \left(\begin{array}{l} \mathbf{Wait}\ 1 \,;\, (PID[E, Kp, Ki, Kd, Y := Ed, Kpd, Kid, Kdd, Yd]) \setminus \{\!| step |\!\} \\ \quad |[\, Internal \,]| \\ Interface(1) \end{array}\right) \setminus Internal$$

Fig. 2. Programming model of the PID

$$
\begin{array}{l}
\textbf{\textit{process}}\ \mathit{Interface} \mathrel{\widehat=} \mathit{stepSize} : \mathbb{R} \bullet \textbf{\textit{begin}} \\
\quad \textbf{\textit{state}}\ \mathit{IS} \mathrel{\widehat=} [\, \mathit{Ev}, \mathit{Kpv}, \mathit{Kiv}, \mathit{Kdv}, \mathit{Et}, \mathit{Kpt}, \mathit{Kit}, \mathit{Kdt}, \mathit{Yv} : \mathbb{R}\,] \\
\quad \mathit{EInp} \mathrel{\widehat=} \left(\begin{array}{l} (\sqcap d : 0 \mathinner{\ldotp\ldotp} \mathit{stepSize} \bullet \textbf{\textit{Wait}}\ d \mathbin{;} (E?x \rightarrow \mathit{Ev}, \mathit{Et} := x, d)\ \textbf{deadline}\ 0) \mathbin{;} \\ (\mu X \bullet E?x \rightarrow \mathit{Ev} := x \mathbin{;} X) \overset{\mathit{stepSize} - \mathit{Et}}{\triangleright} \mathit{Skip} \end{array} \right) \\
\quad \ldots \\
\quad \mathit{Input} \mathrel{\widehat=} \mathit{EInp} \interleave \mathit{KpInp} \interleave \mathit{KiInp} \interleave \mathit{KdInp} \\
\quad \mathit{YOut} \mathrel{\widehat=} (\sqcap d : 0 \mathinner{\ldotp\ldotp} \mathit{stepSize} \bullet \textbf{\textit{Wait}}\ d \mathbin{;} (\bigsqcup v : \mathbb{R} \bullet Y!v \rightarrow \mathit{Yv} := v)\ \textbf{deadline}\ 0) \\
\quad \mathit{Output} \mathrel{\widehat=} \mathit{YOut} \\
\quad \mathit{InputD} \mathrel{\widehat=} \mathit{Ed}!\mathit{Ev} \rightarrow \mathit{Skip} \interleave \mathit{Kpd}!\mathit{Kpv} \rightarrow \mathit{Skip} \interleave \mathit{Kid}!\mathit{Kiv} \rightarrow \mathit{Skip} \interleave \mathit{Kdd}!\mathit{Kdv} \rightarrow \mathit{Skip} \\
\quad \mathit{OutputD} \mathrel{\widehat=} \mathit{Yd}?x \rightarrow \{\mathit{Yv} = x\} \\
\quad \bullet (\mu X \bullet (\mathit{Input} \interleave \mathit{Output}) \mathbin{;} (\mathit{InputD} \interleave \mathit{OutputD}) \mathbin{;} X) \\
\textbf{\textit{end}}
\end{array}
$$

Fig. 3. *Interface* process for the PID

Just like in CSP, systems and their components are described by processes that communicate with each other and the environment using channels. In *Circus* and *CircusTime*, however, a process encapsulates some state (defined just like in Z). More precisely, in an explicit process definition, we define state components and invariants using a Z schema and define behaviour using actions. To specify an action, we use a mixture of CSP constructs, Z data operations, and guarded commands. In the case of *CircusTime*, we additionally use wait, timeout, and deadline operators in the style of Timed CSP [22].

To compose processes, we use CSP operators (for parallelism, choice, and hiding, for instance). Here, we use the alphabetised parallel operator [23], where the set of channels used by each process is defined and they are required to synchronise on the intersection of these sets. We also use hiding, which, just like in CSP, removes a channel, or set of channels, from the interface of a process.

The semantic model for *Circus* has been chosen to support a simple and intuitive notion of refinement: one process Q refines another P, providing that every behaviour of Q is also a behaviour of P. In this way, if specification P is refined by implementation Q, then there is nothing that Q could do that would be forbidden by P, and all its behaviours are specified behaviours. The semantics of *Circus* is a timed extension of the failures-divergences semantics of CSP [12] in the spirit of [22], and so the notion of refinement includes subtle testing of nondeterminism and timing properties.

To illustrate *CircusTime*, we use a timed version of a small example that first appeared in [28], where we used the untimed version of *Circus* to specify a Fibonacci series generator. This is a simple process that generates the Fibonacci series on a channel named “*out*”; the process is described in full in Figure 4. *Circus* processes have encapsulated state that is defined in Z; in the process *Fib*, the state is defined using the schema named *FibState*, which introduces two natural numbers, x and y.

The state is initialised in *InitFibState* to give both state components the initial value of 1. *InitFib* is an action, defined using CSP with embedded references to the state defined in Z. This action first initialises the state using *InitFibState*, then it outputs the first two numbers in the Fibonacci series. It does this with a deadline of 0, which makes the outputs occur instantaneously; after each output, the action pauses for one time interval.

The main work in generating the series is done in the action *OutFib*. First, we define *OutFibState*, which updates the state components: this operation changes the state ($\Delta FibState$) and has one output, defined in the Z convention as $next!$. The effect of this state operation is to set the output to be the same as y', the newest member of the series, which is merely the sum $x + y$, and to copy the value of y to x'.

OutFib itself updates the state using *OutFibState* and then outputs the value of *next* punctually, as before, and it does this repeatedly: the fixed-point operator "μ" introduces tail-recursive iteration. A local-variable block scopes the value of *next*.

After all these schema and action definitions, the real business begins: the main behaviour of the process *Fib* is defined as the composition $InitFib;\ OutFib$.

```
process Fib ≙
   begin
      state FibState == [ x, y : ℕ ]
      InitFibState == [ FibState′ | x′ = y′ = 1 ]
      InitFib ≙
         InitFibState;
         (out!1 → Skip) deadline 0; Wait 1;
         (out!1 → Skip) deadline 0; Wait 1
      OutFibState == [ ΔFibState; next! : ℕ | next! = y′ = x + y ∧ x′ = y ]
      OutFib ≙
         μ X •
            var next : ℕ •
               OutFibState; (out!next → Skip) deadline 0; Wait 1; X
   • InitFib; OutFib
   end
```

Fig. 4. A timed Fibonacci series generator.

3 Simulation models

In this section, we propose a novel approach to construct *CircusTime* simulation models of Simulink diagrams. Like the *Circus*-based strategy, it can be used to generate models automatically [31]. As already said, it produces richer models that cater also for the timing aspects of a larger set of diagrams. We have more faithful models, which we have demonstrated to be in direct correspondence with the simulator behaviour, and as a side effect we also get more compact models.

Our input is a diagram compiled using the fixed-step discrete solver. This means that there is no connection between blocks with incompatible sample times and the discrete sample time of all blocks used for simulation has been defined. (It is possible to leave the sample time of a block to be determined from the context; compilation determines all values.) This is the approach taken, for example, by the MATLAB code generator (Real-time Workshop).

The output of our modelling strategy is a *CircusTime* specification. Its first paragraphs declare channels. Inputs and outputs of the diagram and of the blocks are represented by channels. For the PID, we have the following declaration.

$$\textbf{\textit{channel}}\ E, Kp, Ki, Kd, Y, Si\_out, Diff\_out, Int\_out, Sd\_out, Sp\_out : \mathbb{U}$$

Basically, the channels that represent the inputs and outputs of the diagram are named after the corresponding blocks. The internal wires are represented by channels named after the block that has an output port connected to it. For instance, the wire that connects Diff to Sd in Figure 1 is represented by a channel $Diff\_out$. (As explained in [31], a few special cases need to be considered in the naming rules, but for the purpose of the discussion here, this view is enough.)

The blocks, the solver, and the diagram itself, are modelled by processes. The solver process synchronises with the block processes on a channel *step*. It is used to indicate the occurrence of a simulation step.

$$\textbf{\textit{channel}}\ step : \mathbb{R}$$

We have a discrete-time model, but the sample times and offset can be real numbers, so the type of *step* is $\mathbb{R}$. It is available in the HOL-based theorem prover ProofPower-Z.

The third paragraph of the model declares a type *SampleTime*.

$$SampleTime == [\, sP, o : \mathbb{R}\,]$$

It contains records whose components sP and o define a sample period and an offset. Each block process has a constant of this type to represent its sample time. Below, we explain how the diagram, block, and solver processes are defined.

3.1 Diagram

The processes that model the blocks and the process that models the solver are all composed in parallel to define the process that models the diagram. For our example, this process is sketched below; its name is that of the diagram.

$$\textbf{\textsf{process}}\ PID \mathrel{\widehat=} \left(\begin{array}{l} Si\ \{\!\mid E, Ki, Si\_out, step \mid\!\} \\ \qquad \parallel \\ Diff\ \{\!\mid E, Diff\_out, step \mid\!\} \\ \qquad \parallel \qquad \ldots \parallel \\ FixedStepDiscreteSolver(1,0)\ \{\!\mid step \mid\!\} \end{array}\right) \setminus \{\!\mid Si\_out, Diff\_out, \ldots \mid\!\}$$

The alphabet of each process that models a block includes the channels that are used to represent its inputs and outputs, besides *step*. This reflects the fact that the behaviour of each block is independent, but the way in which their inputs and outputs are connected defines a data flow. In the model, synchronisation on the shared channels between block processes establishes data flow.

The process *FixedStepDiscreteSolver* models the solver; it takes the step size and offset of the diagram as parameters. In our simple example, all blocks have sampling period 1 and offset 0, so the solver uses step size 1 and offset 0.

The channels that represent internal wires (in our example, Si_out, $Diff\_out$, and so on) are hidden. In this way, the channels in the interface of the diagram process are only those that represent inputs and outputs of the system, and *step*.

3.2 The blocks

A block process is defined explicitly, independently of whether the block is simple, like Sd in our example, or a subsystem, like Diff. We consider here blocks with a single sample time; port-based sample time is addressed in Section 3.4.

Figure 5 sketches the model for Diff, a process named *Diff*. An explicit process definition is composed of a sequence of paragraphs. In a block process, we first declare a constant *st* of type *SampleTime* and define the value of its fields. This is defined in the diagram: although it does not (necessarily) appear in its graphical representation, it appears in its textual representation produced by Simulink.

A distinguished paragraph declares the state of the process using a schema named after it; in our example, $Diff\_State$. The state components record the state of the block, and the last calculated output value(s). In our example, we have $pid\_\_Diff\_\_UnitDelay\_state$, which records the state of the Unit Delay block used in the Diff diagram, and $Out1$, corresponding to the single output of Diff.

A schema *Init* defines the state initialisation in the standard Z way. A declaration like $pid\_\_Diff\_State\,'$ introduces dashed versions of the state components to represent their values after initialisation. The components that represent the block state are initialised as determined in the block definition (included in the textual representation of the diagram). In our example, the initial value of the Unit Delay state, represented by $pid\_\_Diff\_State\,'$, is 0. The components that correspond to block outputs, like $Out1$ in our example, are not initialised.

The action at the end of a process definition (after the $\bullet$) is the main action that specifies its behaviour. In a block process, we have a call to *Init*, followed by a recursion (introduced by the μ operator). Each of its iterations models a

$\textbf{\textit{process}}\ Diff \mathrel{\widehat=} \textbf{\textit{begin}}$

$$\begin{array}{|l} st : SampleTime \\ \hline st.sP = 1 \land st.o = 0 \end{array}$$

$\textbf{\textit{state}}\ Diff\_State == [\, pid\_\_Diff\_\_UnitDelay\_state : \mathbb{R};\ ;\ Out1 : \mathbb{R}\,]$

$$\begin{array}{|l} \underline{\ Init\ } \\ pid\_\_Diff\_State' \\ \hline pid\_\_Diff\_\_UnitDelay\_state' = 0 \\ \hline \end{array}$$

$Calculate\_pid\_\_Diff == \ldots$

$$\bullet \left(\begin{array}{l} Init\ ; \\ \left(\begin{array}{l} \mu\, X \bullet \\ \left(\begin{array}{l} step?cT \rightarrow \\ \left(\begin{array}{l} \textbf{\textit{var}}\ In1 : \mathbb{U} \bullet \\ \quad E\,?x \rightarrow In1 := x\ ; \\ \quad \textbf{\textit{if}}\ \ (cT - st.o \geq 0) \land ((cT - st.o) \mathbin{\mathsf{mod}} st.sP = 0) \rightarrow \\ \qquad\quad Calculate\_pid\_\_Diff \\ \quad [\!]\ \ (cT - st.o < 0) \lor ((cT - st.o) \mathbin{\mathsf{mod}} st.sP \neq 0) \rightarrow Skip \\ \quad \textbf{\textit{fi}}\ ; \\ \quad Diff\_out\,!Out1 \rightarrow Skip \end{array} \right) \\ \textbf{deadline}\ 0 \end{array} \right) ; X \end{array} \right) \end{array} \right)$$

$\textbf{\textit{end}}$

Fig. 5. *CircusTime* model of the block Diff

simulation step. It starts with a communication $step?cT$ on the channel *step* to input the current time cT, followed by an interleaving of the inputs, which are recorded in local variables $In1$, $In2$, and so on. In our example, we have just one input $E?x$, and the associated assignment of the input value x to $In1$.

A conditional compares the current and sampling times. If the block offset is over $(cT - st.o \geq 0)$ and we have a sample time hit $((cT - st.o) \mathsf{mod}\ st.sP = 0)$, we calculate the outputs and update the state. Otherwise, nothing happens; the *Skip* action terminates immediately without changing the state.

The required calculations and updates are determined by the functionality of the block (or its diagram, in the case of a subsystem block like Diff). We rely on an industrial tool, namely ClawZ [1], to produce a Z specification for that.[3] (This is the same approach that we take in [3, 4].) ClawZ deals with sequential behaviour; we have extended this to deal with concurrency and timing aspects. For each block, ClawZ produces a Z schema, which we use to define the *CircusTime* process. In our example, we have the *Calculate_pid__Diff* schema, whose definition is constructed using ClawZ. We omit it here, since these details are not relevant for our discussion. In the main action of *Diff*, when there is a sample time hit, we call *Calculate_pid__Diff*.

In any case, the (last calculated) outputs are communicated in interleaving. These are either the outputs that have just been calculated, or those calculated in the previous sample time hit. In our example, we have a single output: we communicate $Out1$ through the channel *Diff_out*.

All this is carried out instantaneously, that is, with **deadline** 0. This captures the idealised-time model of the simulation, where the system is quiescent between the simulation steps, but all the inputs, calculations, updates, and outputs are performed instantaneously (and infinitely fast), when there is a time hit. This is, of course, not an approach that can be taken by a program.

As explained previously, the state components that correspond to an output, like $Out1$, are not initialised. If the solver takes a simulation step before the first sample time hit of the block, the value of the output is arbitrary. For Simulink diagrams whose simulation does not generate any errors, however, such a situation does not arise. Here, as also already said, we are only concerned with compiled diagrams that do not produce simulation errors.

In Section 4, we discuss how the model presented here can be used to construct a model compatible with the restrictions of a real-time computer.

3.3 Solver

To ensure correct timing, the solver process uses the channel *step* to communicate the current time to all blocks in each simulation step. The model of the solver embeds a clock that is indirectly used by all the blocks.

[3] See `www.lemma-one.com/clawz_docs/` for more information about the ClawZ tool, including a user guide to the tool with a simple complete worked example of a Simulink model file, its corresponding Ada code, and a proof of correctness.

The model of the solver is the same for all diagrams; it is presented in Figure 6. The step size sS and offset o are taken as parameters, which are instantiated appropriately for each diagram. The state, defined by the schema *Clock* contains a single component cT to record the current time.

$$\textbf{\textit{process}}\ \mathit{FixedStepDiscreteSolver} \mathrel{\widehat{=}} sS, o : \mathbb{R} \bullet \textbf{\textit{begin}}$$

$$\quad \textbf{\textit{state}}\ \mathit{Clock} == [\, cT : \mathbb{R} \,]$$

$$\quad \bullet \left(\begin{array}{l} cT := o \,;\, \textbf{\textit{Wait}}\ o \,; \\ (\mu\, X \bullet (\mathit{step}!cT \to \mathit{Skip})\ \textbf{deadline}\ 0 \,;\, cT := cT + sS \,;\, \textbf{\textit{Wait}}\ sS \,;\, X) \end{array} \right)$$

$$\textbf{\textit{end}}$$

Fig. 6. *CircusTime* model of a fixed-step discrete solver

In the main action, cT is initialised to o, since nothing happens in the simulation before the diagram offset time. Next, after a wait period of o time units, there is an iteration corresponding to simulation steps. In each iteration, an instantaneous communication (with deadline 0) over the channel *step* outputs the current time cT. Afterwards, cT is increased by the step size sS, and there is a waiting period of sS time units.

3.4 Multi-rate diagrams

Our modelling approach caters for multi-rate diagrams. As already explained, if the blocks have different sample times, the step size and offset of the solver guarantee that they are covered. In the model, at each simulation step, all block processes read inputs and produce outputs, but output calculations and state updates occur only when there is a hit. For blocks with port-based sample times, however, we need to define the block processes differently.

Port-based sample times occur in rate-transition blocks, which have one input and one output port with different sample times, and custom blocks defined by programs. In what follows, we explain our treatment of rate-transition blocks. Custom blocks can be handled in a similar way, but the behaviour when there is a sample-time hit is defined programmatically. Modelling such blocks requires modelling, for example, a C program instead of using a ClawZ schema.

For rate-transition blocks, we need to consider whether its input port is slower (has a longer sampling period) or faster than the output port, as this determines its behaviour when there is a hit. If the input is faster, then the behaviour is that of a zero holder: the block holds its input until there is a hit for the output port. If the input is slower, then the behaviour is that of a unit delay: it outputs the input from a previous hit.

A diagram involving a rate-transition block is provided in Figure 7. In this example, all offsets are 0, but the period of the input port of the rate-transition block (and of the Sine-Wave block) is 0.2, and that of the output port (and of the Gain and Out blocks) is 0.1. The diagram step size is therefore 0.1.

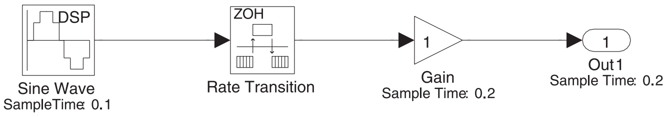

Fig. 7. Multi-rate diagram: rate transition block

The construction of the model of this diagram can proceed much as before, the only difference concerning the rate-transition block process, which is described in Figure 8. First, it records two sample times st_1 and st_2, one for each of its ports.

```
process $Rate\_Transition \mathrel{\widehat{=}}$ begin
  | $st_1, st_2 : SampleTime$
  |-----------------------------
  | $st_1.sP = 0.1 \land st_1.o = 0 \land st_2.sP = 0.2 \land st_1.o = 0$

  state $rt\_\_Rate\_Transition\_State = [\, In1, Out1 : \mathbb{U} \,]$
  . . .

  • ( $\mu X \bullet$
      ( $step?cT \rightarrow$
          ( $Sine\_Wave\_out?x \rightarrow$
              ( if $(cT - st_1.o \geq 0) \land ((cT - st_1.o) \bmod st_1.sP = 0) \rightarrow$
                       $In1 := x$
                [] $(cT - st_1.o < 0) \lor ((cT - st_1.o) \bmod st_1.sP \neq 0) \rightarrow$
                       $Skip$
                fi
              ) ;
            ( if $(cT - st_2.o \geq 0) \land ((cT - st_2.o) \bmod st_2.sP = 0) \rightarrow$
                     $rt\_Rate\_Transition$
              [] $(cT - st_2.o < 0) \lor ((cT - st_2.o) \bmod st_2.sP \neq 0) \rightarrow Skip$
              fi
            ) ;
            $One\_out\,!Out1 \rightarrow Skip$
          )
        deadline 0
      ) ;
      $X$
    )
end
```

Fig. 8. *CircusTime* model of a rate-transition block: zero-order hold

Just as before, in each simulation step, the input is taken and the output is produced. Due to the lack of synchrony between inputs and outputs, we do not keep only the most recently output value from one step to the next, but also the most recent input. We have both $In1$ and $Out1$ as state components.

If a simulation step is a hit for the sample time of the input, $In1$ is updated, otherwise the input taken is ignored. If there is a hit for the sample time of the output, then it is calculated. In the case of a zero-order hold block, the calculation, as defined by ClawZ, updates $Out1$ to the value of the most recent input $In1$. This is defined in the (omitted) schema $rt\_Rate\_Transition$.

For a rate-transition block with a slower input we have a block process whose main action is shown in Figure 9. In this case, we have an additional state component $state$ as indicated by ClawZ, and the $Init$ action initialises the $state$ as defined in the block properties, and also captured by ClawZ.

$$
\left(\begin{array}{l}
Init\,; \\
\left(\begin{array}{l}
\mu X \bullet \\
\left(\begin{array}{l}
step?cT \rightarrow \\
\left(\begin{array}{l}
Sine\_Wave\_out?x \rightarrow \\
\quad \left(\begin{array}{l}
\textbf{\textit{if}}\ \ (cT - st_1.o \geq 0) \land ((cT - st_1.o) \mathsf{\ mod\ } st_1.sP = 0) \rightarrow \\
\qquad\quad rt\_Rate\_Transition \\
[\!]\ \ (cT - st_1.o < 0) \lor ((cT - st_1.o) \mathsf{\ mod\ } st_1.sP \neq 0) \rightarrow \\
\qquad\quad Skip \\
\textbf{\textit{fi}}
\end{array}\right) ; \\
One\_out\,!Out1 \rightarrow Skip
\end{array}\right) \\
\textbf{deadline}\ 0
\end{array}\right) ; \\
X
\end{array}\right)
\end{array}\right)
$$

Fig. 9. *CircusTime* model of a rate-transition block: unit delay (main action)

The output calculations and state updates are determined by the sample-time hit of the slow port. Here, it is the input, so we do not need to check for hits of the output nor record the inputs in the state. The $rt\_Rate\_Transition$ schema defines that $Out1$ is assigned the current value of the state, which becomes the freshly input value x. This is the standard definition of a unit delay.

In the following section, we describe how these simulation models can be used to define models appropriate for program verification.

4 Programming models

The idealised simulation model requires the calculations and communications to take place infinitely fast. For program verification, we need a model that captures the assumptions that allow us to conclude that an implementation is correct, from the timing as well as the functional point of view, even though it is restricted by the performance of the real-time computer on which it runs.

All calculations and communications take place instantaneously at each simulation time step. For programs, we expect a time line where all calculations and communications take place during the intervals defined by the hits. This is, for example, the view adopted by the MATLAB code generator. The (implicit) assumption adopted in the default configuration of this code generator is that the simulation steps define execution cycles. Additionally, the environment keeps the inputs constant and available during each cycle, and is ready to accept an output at any point during the cycle.

Here, we explain how the simulation model can be used to construct a model for program verification. The result is a new *CircusTime* process; for our PID example, this is *PIDTSpec*, presented in Figure 2. Roughly, it is defined by composing the simulation model of the diagram, in our case, *PID*, in parallel with a process *Interface* that handles the inputs and outputs of the system to capture the assumptions about the environment.

We need to adapt the simulation model of the diagram in three ways. First, we need to address the fact that the simulation provides outputs already at the first hit, even if it is at time 0, while the program needs to take some time before it can produce results. In the program verification model, therefore, the simulation model is used after a wait period. In this way, the simulation time line is shifted, and during that initial period the computations can start. We use the step size as the wait period; in our example, this is 1. The assumption is that, for all blocks, one step is enough to calculate the outputs and make the state updates. This is again the view taken by the MATLAB code generator.

A second, most important observation is that the inputs and outputs of the simulation model correspond to those of the system. It is, however, the *Interface* process that needs to handle these communications. For this reason, we use the simulation diagram process obtained by renaming the input and output channels. The new channels are used for internal communication with *Interface*. For the *PID*, we declare the channels *Ed*, *Kpd*, *Kid*, *Kdd*, and *Yd*, and in the definition of *PIDTSpec*, we use $PID[E, Kp, Ki, Kd, Y := Ed, Kpd, Kid, Kdd, Yd]$. We also define a channel set *Internal* to include all the new channels, which are hidden in the programming model (see Figure 2).

A final observation is that the *step* channel is used to mark the simulation steps, and has no role in the program. So, it is hidden as well.

Figure 3 sketches the definition of the *Interface* process used in the specification of *PIDTSpec*. In all cases, *Interface* takes the diagram step size as a parameter. If the offset of the diagram is not 0, then it is preceded by a corresponding wait in the programming model. For the PID, this is not necessary.

The state of *Interface* includes two components for each input of the diagram, and one for each output. For an input E, for instance, the component *Ev* records the last value input, and *Et* records the time the input was first read in the current cycle. The output components hold the values output by the program. For the PID, the inputs give rise to state components *Ev*, *Kpv*, *Kiv*, and *Kdv*, and *Et*, *Kpt*, *Kit* and *Kdt*, and the output to a component *Yv*.

The behaviour of *Interface* is characterised by iterations that correspond to the simulation steps of the diagram. During each of them, *Interface* interacts with the environment; it reads the inputs one or more times, and produces the outputs as required. At the end of each step, it interacts with the simulation process to provide its inputs and take its outputs.

Correspondingly, in the specification of *Interface*, the main action's iterations correspond to the simulation steps. A sequence (;) splits each iteration into two parts, corresponding to the period before the simulation step and to its end, which is the exact moment of a simulation time step. During each iteration, the behaviour is defined by the interleaving ($|||$) of actions *Input* and *Output*, which interact with the environment. At the simulation time step, the behaviour is given by the interleaving of *InputD* and *OutputD*, which interact with the simulation diagram model.

As detailed later on in this section, the values of the state components *Ov* that are output to the environment in *Output* are chosen angelically. In our example, we have a single component *Yv* corresponding to an output, and its value is chosen angelically. An angelic choice is resolved in a way that ensures, if at all possible, that the program does not abort. In programming terms, it provides a backtracking mechanism. In *OutputD*, the value x provided by the simulation model for the output is compared to that of the corresponding state component *Ov* in an assumption $\{Ov = x\}$. In our example, we have $\{Yv = x\}$. An assumption $\{Ov = x\}$ is an action that aborts if *Ov* is different from x, but otherwise skips. Since the value of *Ov* is angelically chosen, the assumption is guaranteed to hold; effectively, it forces the value *Ov* to be chosen correctly.

Angelic nondeterminism is typically used as a specification construct. This is certainly the case here. Refinement of our models to feasible programs leads to implementations that make the appropriate calculations to determine the value to be output. Use of backtracking is not really practical or necessary.

Since we have an assumption that the values of the inputs are constant during a cycle, each input can be read any number of times during each iteration, but at least once. For each input, we define an action that specifies this behaviour.

For the input E of the PID model, for instance, we have the action *EInp*. The internal choice ($\sqcap$) over a delay d allows the first input to happen at any time during the iteration. More precisely, the wait of d time units followed by the instantaneous communication over E specifies that the input occurs exactly after d time units. Additionally, the internal choice of d in the specification model means that a program can choose a value for d freely: it can carry out the input at any time, whenever needed, during the iteration. In the specification model, that time is recorded in the state component *Et*. The value input itself is recorded in *Ev*. After that first communication, additional inputs on E can happen any number of times, during the rest of the iteration. After $stepSize - Et$ time units, however, the iteration finishes, and so does the input action. A timeout (operator $\overset{d}{\triangleright}$) interrupts its recursive execution in favour of *Skip*.

The inputs are all independent, so the action *Input* that specifies the program inputting behaviour is defined by the interleaving ($|||$) of the actions that handle each of the diagram inputs. In Figure 3, we omit the definitions of *KpInp*, *KiInp*, and *KdInp*, which are similar to that of *EInp*.

Each output is produced just once, but at any time, during the iteration. For each output, we have an action in *Interface*. In our example, we have just *YOut*, because we have only one output. Like in an input action such as *EInp*, in an output action we use an internal choice to leave open the choice of when the output is produced. As already mentioned, an angelic choice ($\bigsqcup$) determines the value v to be output and recorded in the corresponding state component.

If there are several outputs, the *Output* action is defined as their interleaving. In the case of the PID, we have just one output, so *Output* is just *YOut*.

In the interaction between *Interface* and the simulation process, the inputs are produced in interleaving; this is defined by the action *InputD*. Similarly, *OutputD* takes all outputs in interleaving. The values received from the simulation model, however, are compared to those previously output. In Figure 3, after reading the value x through the internal channel *Yd*, we have the assumption $\{Yv = x\}$. As explained above, it determines the angelic choice in *Output*.

The action *OutputD* interleaves all communications to receive outputs from the simulation process and their associated assumptions. For the PID, which has just one output, no interleaving is needed. The deadlines in the simulation model guarantee that all communications in *InputD* and *OutputD* are instantaneous.

The step size of the simulation diagram process, as defined in the instantiation of the solver process, and that of the *Interface* process should be the same. This can be easily ensured when the models are generated automatically.

The external channels of *PID* and *PIDTSpec* are the same. This holds in general for simulation and programming models constructed as described here.

5 Conclusions

In this paper, we propose a modelling strategy to use *CircusTime* to capture both timing properties embedded in the simulation model of a Simulink diagram and the timing assumptions embedded in a typical programming environment. The models produced capture functional, behavioural, and timing aspects of diagrams. The use of a refinement notation, and the consideration of programming concerns, make the models useful for program verification.

The simulation model is in direct correspondence with the description of the Simulink simulator. Using *CircusTime* and angelic nondeterminism, we construct a programming model that records the environment assumptions, but uses the simulation model to specify functionality and data flow. We overcome three challenges in establishing the connection between the two models: in the simulation model, outputs can be produced immediately at time 0, infinitely fast at each time hit, and simulation steps do not have a role in a programming model.

We do not take into account the possibility of overflow of timers. Run-time exceptions need to be handled separately.

For validation, we have checked classical properties (deadlock and livelock freedom, and absence of nondeterminism) and analysed timing aspects. The validation has consisted in initially converting *CircusTime* to CSP to use the FDR2 model checker, following a strategy similar to [19], where timing aspects are captured according to [24].[4] Using the CSP animator, we have observed that at time 0 all inputs are performed in any possible order. After that, internal calculations take place generating the observable output through channel Y. At this point, time (represented by a *tock* event) has to pass before the previous input and output behaviour occurs again. This pattern repeats itself as expected.

As far as we know, there is no report in the literature of support for formal program verification that takes into account the way in which the time model of Simulink diagrams are adapted. Moreover, we are not aware of any formalisation of the typical assumptions embedded in the programming timing model of implementations of control law diagrams.

Timed models of Simulink diagrams are also considered in [14], where a notation called SPI is used to capture control flow. SPI is based on communicating processes; it does not incorporate data operations, but supports the specification of timing restrictions. Since the idealised model of Simulink is not relevant to implementations, the proposed approach is the formalisation of timing requirements after the translation to SPI. It allows the specification of (mode-dependent) data rates and latency times. Using the timed model, it is possible to use static analysis to tackle scheduling. Here, we propose the automatic generation of models.

The approach in [16] uses an extension of Simulink to specify real-time interactions. It is based on a programming language called Giotto. The extended model is translated to Simulink, and then to a program that combines the result of the Simulink code generator with a Giotto program that handles the scheduling. This program runs in an embedded machine that is platform dependent. In our approach, assumptions related to real-time programming are captured using *CircusTime* constructs, and they are uniformly specified (by an *Interface* process) for all applications to be deployed in a particular platform.

The combined use of UML and Simulink is supported by the work in [10], a technique to verify real-time properties of a distributed design compositionally using model checking. It is also part of the trend to verify models and designs, and rely on code generators for the automatic production of programs [11].

The work in [6] also proposes the characterisation of timing requirements based on a calculated model of Simulink diagrams. In that case, the modelling language is TIC (Timed Interval Calculus), a Z-based notation that supports the definition of total functions of continuous time, and (sets of) time intervals. Blocks are specified by schemas, like in ClawZ, but their components include functions from time that define how inputs vary with time, and how outputs are related to inputs over time intervals. Both continuous and discrete times are considered, but not concurrency. The objective of the work is the analysis of diagrams; tool support based on PVS is provided.

A first important piece of future work is the extension of the tool in [31] to automate the generation of our *CircusTime* models and enable significant case studies. For refinement, we will investigate a strategy that transforms the timed

[4] FDR2 is a refinement-checking software tool that can check whether one CSP process refines another, or that a process is free from deadlock, livelock, or nondeterminism. More details about the tool are available from `www.fsel.com/`.

models described here into synchronisation-based models similar to those used in [4]. In that way, we can reuse the verification approach in that work.

A more foundational piece of future work is related to our use of angelic nondeterminism. The semantics of *Circus* and *CircusTime* is defined using Hoare and He's Unifying Theories of Programming (UTP) [13]. In [5], we describe a UTP model for angelic nondeterminism. It remains for us to investigate the consequences of the integration of that model with the *CircusTime* model with deadlines, and to propose and prove laws to support a refinement strategy.

Acknowledgements

Ana Cavalcanti is grateful for the support of EPSRC (grant EP/E025366/1). The work of Alexandre Mota was partially supported by INES11, funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, by CNPq grant 482462/2009-4 and by the Brazilian Space Agency (UNIESPACO 2009).

References

1. R. Arthan, P. Caseley, C. M. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *3rd IEEE International Conference on Formal Engineering Methods, ICFEM 2000*, 4–7 September 2000, York, Proceedings, IEEE Computer Society, 169–176, IEEE Press, 2000.
2. Pontus Boström, Lionel Morel, and Marina A. Waldén. Stepwise development of Simulink models using the refinement calculus framework. In Cliff B. Jones, Zhiming Liu, and Jim Woodcock, *Theoretical Aspects of Computing—ICTAC 2007, 4th International Colloquium, Macau, September 26–28, 2007*, Springer, LNCS, 4711, 79–93, 2007.
3. Ana Cavalcanti, Phil Clayton, and Colin O'Halloran. Control law diagrams in *Circus*. In John A. Fitzgerald, Ian J. Hayes, and Andrzej Tarlecki, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe*, Newcastle, July 18–22, 2005, Springer, LNCS, 3582, 253–268, 2005.
4. A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. From control law diagrams to Ada via *Circus*. *Formal Aspects of Computing* 23(4):465–512 2011.
5. A. L. C. Cavalcanti, J. C. P. Woodcock, and S. Dunne. Angelic nondeterminism in the Unifying Theories of Programming . *Formal Aspects of Computing* 18(3):288–307 2006.
6. C. Chen, J. S. Dong, and J. Sun. A formal framework for modeling and validating Simulink diagrams. *Formal Aspects of Computing* 21(5):451–484 2009.
7. Ewen Denney and Bernd Fischer. Generating customized verifiers for automatically generated code. In Yannis Smaragdakis and Jeremy G. Siek (eds), *Generative Programming and Component Engineering, 7th International Conference, GPCE 2008, Nashville, October 19–23, 2008, Proceedings*, ACM, 77–88, 2008.
8. Abderrahmane Feliachi, Marie-Claude Gaudel, and Burkhart Wolff. Isabelle/Circus: a process specification and verification environment. In Rajeev Joshi, Peter Müller, and Andreas Podelski (eds), *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012*, Philadelphia, January 28–29, 2012. Springer, LNCS, 7152, 243–260, 2012.

9. Abderrahmane Feliachi, Burkhart Wolff, and Marie-Claude Gaudel. Isabelle/Circus. *Archive of Formal Proofs*, `afp.sourceforge.net/entries/Circus.shtml`, 2012.
10. Holger Giese and Martin Hirsch. Modular verification of safe online-reconfiguration for proactive components in Mechatronic UML. In Jean-Michel Bruel (ed.), *Satellite Events at the MoDELS 2005 Conference, MoDELS 2005 International Workshops, Doctoral Symposium, Educators Symposium*, Montego Bay, October 2–7, 2005, Revised Selected Papers, Springer, LNCS, 3844, 67–78, 2006.
11. Susanne Graf, Sébastien Gérard, Øystein Haugen, Iulian Ober, and Bran Selic. Modelling and analysis of real time and embedded systems—Using UML. In Thomas Kühne (ed.), *Models in Software Engineering, Workshops and Symposia at MoDELS 2006*, Genoa October 1–6, 2006, Reports and Revised Selected Papers, Springer, LNCS, 4364, 126–130, 2007.
12. C. A. R. Hoare. *Communicating Sequential Processes.* Prentice Hall International, Series in Computer Science, 1986.
13. C. A. R. Hoare and Jifeng He. *Unifying Theories of Programming.* Series in Computer Science. Prentice Hall, 1998.
14. Marek Jersak, Dirk Ziegenbein, Fabian Wolf, Kai Richter, Rolf Ernst, Frank Cieslog, Jürgen Teich, Karsten Strehl, and Lothar Thiele. Embedded system design using the SPI Workbench. In *Proceedings of the 3rd International Forum on Design Languages*, 2000.
15. Anjali Joshi and Mats Per Erik Heimdahl. Model-based safety analysis of Simulink models using SCADE Design Verifier. In Rune Winther, Bjørn Axel Gran, and Gustav Dahll (eds), *Computer Safety, Reliability, and Security, 24th International Conference, SAFECOMP 2005*, Fredrikstad, Norway, September 28–30, 2005, Springer, LNCS, 3688, 122–135, 2005.
16. Christoph M. Kirsch, Marco A. A. Sanvido, Thomas A. Henzinger, and Wolfgang Pree. A Giotto-based helicopter control system. In Alberto L. Sangiovanni-Vincentelli and Joseph Sifakis (eds), *Embedded Software, Second International Conference, EMSOFT 2002*, Grenoble, October 7–9, 2002, Springer, LNCS, 2491, 46–60, 2002.
17. The MathWorks,Inc. *Simulink.* `www.mathworks.com/products/simulink`.
18. C. C. Morgan. *Programming from Specifications.* Prentice Hall, 2nd edition, 1994.
19. A. C. Mota and A. C. A. Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. *Science of Computer Programming* 40:59–96 2001.
20. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP semantics for *Circus*. *Formal Aspects of Computing* 21(1-2):3–32 2009.
21. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in ProofPower-Z. *Formal Aspects of Computing* 25(1):133–158 2013.
22. G. M. Reed and A. W. Roscoe. A timed model for communicating sequential processes. *Theoretical Computer Science* 58:249–261 1988.
23. A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice Hall Series in Computer Science, 1998.
24. S. Schneider. *Concurrent and Real-time Systems: The CSP Approach.* Wiley, 2000.
25. Adnan Sherif and Jifeng He. Towards a time model for *Circus*. In Chris George and Huaikou Miao (eds), *Formal Methods and Software Engineering, 4th International Conference on Formal Engineering Methods, ICFEM 2002*, Shanghai, October 21–25, 2002, Springer, LNCS, 2495, 613–624, 2002.
26. Adnan Sherif, Jifeng He, Ana Cavalcanti, and Augusto Sampaio. A framework for specification and validation of real-time systems using *Circus* actions. In Zhiming Liu and Keijiro Araki (eds), *Theoretical Aspects of Computing—ICTAC 2004,*

First International Colloquium, Guiyang, September 20–24, 2004, Revised Selected Papers, Springer, LNCS, 3407, 478–493, 2005.

27. 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. *Formal Aspects of Computing* 22(2):153–191 2010.
28. Jim Woodcock and Ana Cavalcanti. A concurrent language for refinement. In Andrew Butterfield, Glenn Strong, and Claus Pahl (eds), *5th Irish Workshop on Formal Methods, IWFM 2001*, Dublin, 16–17 July 2001, BCS, Workshops in Computing, 2001.
29. Jim Woodcock and Ana Cavalcanti. The semantics of Circus. In Didier Bert, Jonathan P. Bowen, Martin C. Henson, and Ken Robinson (eds), *ZB 2002: Formal Specification and Development in Z and B, 2nd International Conference of B and Z Users*, Grenoble, January 23–25, 2002, Springer, LNCS, 2272, 184–203, 2002.
30. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof.* Prentice-Hall, 1996.
31. Frank Zeyda and Ana Cavalcanti. Mechanised Translation of Control Law Diagrams into Circus. In Michael Leuschel and Heike Wehrheim (eds), *Integrated Formal Methods, 7th International Conference, IFM 2009*, Düsseldorf, February 16–19, 2009, Springer, LNCS, 5423, 151–166, 2009.