Control Law Diagrams in *Circus*

Ana Cavalcanti[1] and Phil Clayton[2] and Colin O'Halloran[2]

[1] Department of Computer Science, University of York
York, England
[2] Systems Assurance Group, QinetiQ
Malvern, England

Abstract. Control diagrams are routinely used by engineers in the design of control systems. Yet, currently the formal verification of programs that implement the diagrams is a challenge. We present a strategy to translate block diagrams to *Circus*, a notation that combines Z, CSP, and a refinement calculus. This work is based on existing tools that produce Z and CSP specifications from discrete-time block diagrams. By using a combined notation, we provide a specification that considers both functional and behavioural aspects of the diagrams, and can cover a wider range of blocks. Moreover, the *Circus* refinement calculus can be used to derive or verify implementations, and reason about the block diagrams.

Keywords: Z, CSP, Simulink, refinement.

1 Introduction

A popular and intuitive representation for expressing control system specifications is that of block diagrams. In this notation, a system is modelled by a, possibly cyclic, directed graph of blocks interconnected by wires. This graph includes inputs and outputs to the system, which are signals carried by the wires. Roughly speaking, the blocks represent functions that determine how the outputs are calculated from the inputs. In a continuous-time model, signals continuously vary with time. In a discrete-time model, signals are sampled at discrete time intervals; input and output take place in cycles.

Due to the criticality of many control systems, analysis has been a major concern; numerical modelling and simulation are the established techniques. Recently, there have been efforts to use logic to capture the meaning of control diagrams and to support reasoning [4, 3, 10]. Our work has a different focus: derivation and verification of implementations, as opposed to validation of systems.

Discrete-time diagrams written using Simulink are considered in [2]. Simulink is a popular tool that is part of the Matlab environment [1]; its use in the avionics and automotive sectors is standard. In [2] we find the description a tool, ClawZ, that translates control law diagrams to Z. The translation is based on an extensive Z library that formalises the meaning of many of the blocks. The version of Z used is that implemented in the theorem prover ProofPower [11].

ClawZ has been extensively and successfully used at the Systems Assurance Group at QinetiQ in the proof of correctness of Ada programs with respect to

Simulink specifications. As described in [14], the output of ClawZ is used to construct a refinement conjecture (called a compliance argument) that can be formally verified using tools integrated with ProofPower.

In Z, reactivity and concurrency cannot be modelled directly; ClawZ captures only the functional behaviour of one cycle of a control system. Basically, the Z specification that it generates defines how the outputs of a cycle can be determined in terms of the inputs (and possibly, state information).

QinetiQ developed another tool, called ClaSP, to support the definition of a CSP [16] specification that captures the parallelism inherent in a control law diagram. In principle, the computation embedded in the blocks can be performed in parallel; order is imposed only by the wiring. ClaSP is used in the verification by model checking of distributed cyclic scheduling.

Circus [19, 6] is a combination of Z and CSP with a refinement calculus; it aims at the specification and design of state-rich reactive systems. *Circus* includes a theory and a technique of refinement that support the calculation of concurrent implementations from centralised specifications. The semantics is based on Hoare and He's unifying theories of programming [9].

In this work, we give a semantics to control diagrams using *Circus*, so that we can capture functionality and concurrency. We reuse ClawZ and ClaSP, which capture a partial semantics of these diagrams. Our semantics is a strategy to translate the outputs of extended versions ClawZ and ClaSP to a *Circus* specification: extensions are needed to enlarge the subset of the diagrammatic notation that is covered. Even so, the existing experience with ClawZ and ClaSP improves our confidence in the suitability of the *Circus* semantics.

Using *Circus*, we can model blocks whose output can be disabled or depends on the order of arrival of input signals. Moreover, the *Circus* specification can capture the behaviour of the system over any number of cycles; our model of a diagram is a process that proceeds recursively executing cycle after cycle.

With a *Circus* model, we are able to use refinement to reason about diagrams and their implementations. Separate analyses that consider functionality and concurrency independently are not needed. Properties that are based on both the functionality and the scheduling policies of an implementation can be handled.

In the next section, we present a brief introduction to Simulink control law diagrams. In Section 3 we describe ClawZ, ClaSP, and *Circus*; the extensions of ClawZ and ClaSP are described in Sections 4 and 5. Our translation strategy is presented in Section 6; refinement is discussed and exemplified in Section 7. In Section 8 we summarise our work and discuss future and related work.

2 Control law diagrams

Our work is based on the Simulink notation; an example is presented in Figure 1. That diagram specifies a PID (Proportional Integral Derivative) controller that is being used to control a fuel metering valve of an aircraft. Each box in a diagram is called a block; the wires carry signals. The inputs and outputs of a

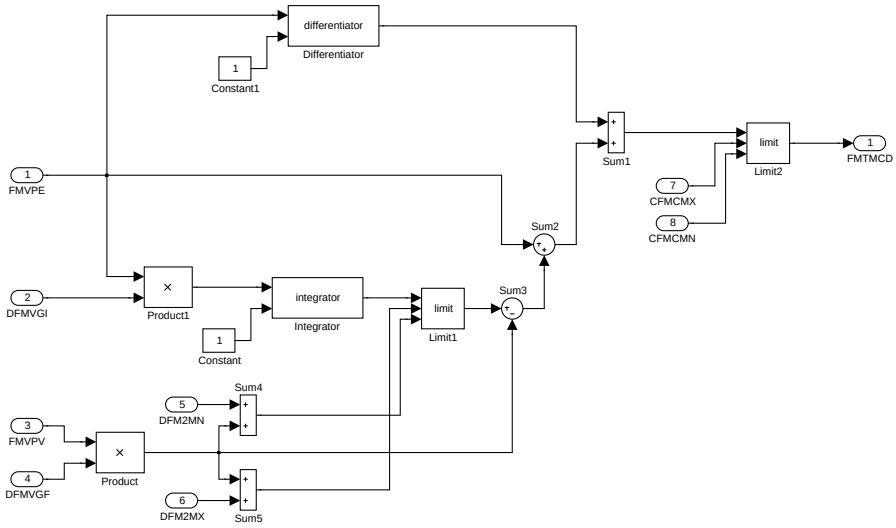

Fig. 1. PID (Proportional Integral Derivative) controller

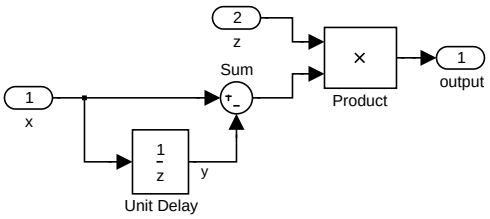

Fig. 2. PID Differentiator

system are represented by rounded boxes containing numbers. In our example, there are eight inputs and one output.

Typically, a block takes some input signals and produces some outputs according to a function determined by the kind of block in question. There are libraries of blocks, and they can also be user-defined.

The rectangular boxes without inputs output the constant value they display. The circles are sum blocks. Boxes enclosing names are subsystems; they denote control systems defined in other diagrams. For example, the diagram that corresponds to the Differentiator block is presented in Figure 2.

Blocks can have state. For example, blocks labelled $1/z$ are unit delay blocks. They store the value of the input signal, and output the value stored in the previous cycle. In each cycle, the output depends on the values of the inputs and of the state that may be held in the blocks, but other factors may be relevant.

For example, subsystems may be conditionally executed: an action subsystem has an activate input and is executed when it is true; an enabled subsystem has

an enabling input and is executed when its value is greater than zero. When a subsystem is not executed, its outputs can either be held at their previous value or reset to an initial value. Any state contained in blocks within the subsystem is held until the subsystem is about to be executed again, at which point the states can be held or reset to an initial value. Merge blocks take a number of inputs and produce one output: the most recently calculated input.

In the next section, we present two models for control diagrams provided by two tools. ClawZ uses Z to provide a relational model for blocks, which covers state, but not concurrency and the behaviour of conditionally executed subsystems and merge blocks. ClaSP, on the other hand, cannot capture functionality.

3 ClawZ, ClaSP, and *Circus*

ClawZ characterises each block of a Simulink diagram, including constants, as a set of bindings, typically defined as a schema. In the Z specification of a diagram, there is a set of bindings for each block, and a set of bindings corresponding to the whole diagram. Part of the output of ClawZ for the PID diagram in Figure 1 is presented in Figure 3; the Z notation is that adopted by ProofPower.

The schema *pidspec* declares the inputs and the outputs of the diagram, and includes (the schemas that specify) the blocks. The predicate of *pidspec* (omitted) specifies how the inputs and outputs of the diagram and of each of the blocks are connected. The type $\mathbb{U}$ is a universal type in ProofPower.

We present only the definition of the Differentiator; it is a schema that declares the inputs and outputs of the Differentiator block, and each of the blocks in its diagram (Figure 2). The predicate, which is similar to that of *pidspec*, equates, for instance, the inputs of the Product block to an input of the whole block and the output of the Sum block.

ClawZ includes a library of block definitions. The Product block of the Differentiator is defined in terms of the library block $Product\_M2$. The Unit Delay block specification uses $UnitDelay\_g$; it is a function that takes a binding that defines the initial value of the unit delay state, and gives a set of bindings. In ProofPower, there is support for real numbers: $0\ e\ 0$ is the real number 0.

ClaSP provides a simple characterisation of the wiring in a diagram; it ignores the calculations performed by the blocks. The output of ClaSP is not really a CSP specification, but a set of pairs that is used as argument for a CSP process that defines the concurrent behaviour of the diagram. The set includes one pair for each block in the diagram: the first element of the pair is the set of input signals of the block, and the second element is a sequence of output signals.

The output of ClaSP for the PID is shown in Figure 4. To make model checking practical, the CSP process that uses this set of pair determines an order of execution for the blocks; this is why the outputs are identified by sequences. The massive parallelism intrinsic in a block diagram leads to processes that have a large number of states and are difficult to model check.

Circus is a language for refinement; it includes specification constructs from Z and Morgan's refinement calculus [13], CSP constructs to model communica-

z

$$pidspec\_\_Differentiator\_\_Product \mathrel{\widehat=} Product\_M2$$

z

$$pidspec\_\_Differentiator\_\_UnitDelay \mathrel{\widehat=} UnitDelay\_g\ (X\,0 \mathrel{\widehat=} 0\ e\ 0)$$

z

pidspec__Differentiator

$In1? : U;\ In2? : U;$
$Product : pidspec\_\_Differentiator\_\_Product;$
$Sum : pidspec\_\_Differentiator\_\_Sum;$
$UnitDelay : pidspec\_\_Differentiator\_\_UnitDelay;$
$Out1! : U$

$Out1! = Product.Out1!;$
$Product.In1? = In2? \land Product.In2? = Sum.Out1!;$
$Sum.In2? = UnitDelay.Out1!;$
$UnitDelay.In1? = Sum.In1? = In1?$

z

pidspec

$In1? : U;\ In2? : U;\ In3? : U;\ In4? : U;\ In5? : U;\ In6? : U;\ In7? : U;\ In8? : U;$
$Constant : pidspec\_\_Constant;\ Constant1 : pidspec\_\_Constant1;$
$Differentiator : pidspec\_\_Differentiator;$
...
$Out1! : U$

...

Fig. 3. ClawZ output for the PID (ProofPower notation)

tion and concurrency, and Dijkstra's language of guarded commands. A *Circus* program is a sequence of paragraphs, just like in Z, but they also include channel and process declarations. Section 6 gives examples.

A process encapsulates state and exhibits behaviour. Like a *Circus* program, an explicit definition of a process is a sequence of paragraphs; Figure 6 has an example. A distinguished paragraph introduces the state schema. At the end, a main action specifies the behaviour of the process. Actions are (composed of) Z operations, CSP processes, and guarded commands. Typically, a process includes several paragraphs to define actions that are combined in the main action to specify the behaviour of the process. Processes can be combined using CSP operators: choice, parallelism, hiding, and others.

Communications are events, just like in CSP; if their occurrence entails a state change, a state operation needs to be used. If a Z operation is used outside its precondition, it diverges, just like in Z. Guards can be explicitly defined.

Parallelism is alphabetised; we can either define a synchronisation set or the alphabet of the parallel processes. A synchronisation set determines the channels for which communication requires synchronisation. The alphabet of a process is

$\{ (\{ FMVPE \}, \langle Differentiator\_out \rangle), (\{ FMVPE, DFMVGI \}, \langle Product1\_out \rangle),$
$(\{ FMVPE, Sum3\_out \}, \langle Sum2\_out \rangle), (\{ FMVPV, DFMVGF \}, \langle Product\_out \rangle),$
$(\{ Product1\_out \}, \langle Integrator\_out \rangle), (\{ DFM2MN, Product\_out \}, \langle Sum4\_out \rangle),$
$(\{ DFM2MX, Product\_out \}, \langle Sum5\_out \rangle),$
$(\{ CFMCMX, CFMCMN, Sum1\_out \}, \langle FMTMCD \rangle),$
$(\{ differentiator\_out, Sum2\_out \}, \langle Sum1\_out \rangle),$
$(\{ integrator\_out, Sum5\_out, Sum4\_out \}, \langle Limit1\_out \rangle),$
$(\{ Limit1\_out, Product1\_out \}, \langle Sum3\_out \rangle) \}$

Fig. 4. ClaSP output for the PID

the set of channels that it can use; synchronisation is required for the channels in the intersection of alphabetised parallel processes. In the case of actions, there is a concern about conflicting access to the state. The parallel composition of actions A_1 and A_2 with a synchronisation set cs is written $A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2$, where ns_1 and ns_2 are disjoint sets of names of state components. Both A_1 and A_2 have access to the initial value of all state components; however, A_1 can only modify the components named in ns_1, and A_2 can only modify those in ns_2. The same concerns apply for interleaving of actions.

A refinement calculus and strategy is available for *Circus* [6]. The strategy aims at calculating concurrent implementations from centralised specifications. Using the *Circus* refinement theory, we can implement and reason about the *Circus* model of a diagram. Examples are considered in Section 7.

4 Extensions to ClawZ

The translation of diagrams to *Circus* is based on the output of extended versions of ClawZ and ClaSP. ClawZ is extended to include action and enabled subsystems, and merge blocks; they are representative in the treatment of conditional execution and order of arrival of inputs. In the translation of an action subsystem, we need a record of the enabling condition and the value of its outputs separately. The schema that records the enabling condition is named after the block with the suffix *_Enabling*. Schemas with suffix *_Enabled* and *_Disabled* define the values of the outputs in the case the system is enabled and in the case the system is disabled. The schema that defines the subsystem combines these schemas. For enabled subsystems, the strategy is similar.

The definition of a merge block is nondeterministic, and requires information about whether the inputs have been computed or not, and their order of arrival. Below, we present the definition of a merge block with two inputs $In1?$ and $In2?$. Two extra inputs $In1Computed?$ and $In2Computed?$ determine whether the values input have been freshly calculated or are just default or held values. The boolean type $BOOL$ is available in ProofPower, although it is not part of Standard Z. The component $arrOrder$ is a sequence of input indexes that defines the order of arrival of the inputs. The single output is $Out1!$.

If a block has a state, its Z specification would typically involve three schemas to define the state, the initial state, and the calculation of outputs. The ClawZ

library, however, includes many block definitions, and, for clarity and simplicity, it groups the definition of each block in a single schema. Components *state*, $state'$, and $initial\_state$ record the value of the state at the beginning of each cycle, and its initial value. This is the approach we adopt in *Merge*2.

*Merge*2

$$
\begin{array}{l}
In1?, In2? : \mathbb{U} \\
In1Computed?, In2Computed? : BOOL \\
arrOrder : \operatorname{seq} 1 \ldots 2 \\
state, state', initial\_state : \mathbb{U} \\
Out1! : \mathbb{U} \\
\hline
initial\_state = (0\ e\ 0) \\
In1Computed? \land \neg\ In2Computed? \Rightarrow Out1! = In2? = state' \\
In2Computed? \land \neg\ In1Computed? \Rightarrow Out1! = In1? = state' \\
\neg\ In1Computed? \land \neg\ In2Computed? \Rightarrow Out1! = state = state' \\
In1Computed? \land In2Computed? \Rightarrow \\
\quad last\ arrOrder = 1 \Rightarrow Out1! = In1? = state' \land \\
\quad last\ arrOrder = 2 \Rightarrow Out1! = In2? = state'
\end{array}
$$

The extra information (*In*1*Computed*?, *In*2*Computed*?, and *arrOrder*) required by *Merge*2 is determined in the *Circus* specification.

5 Extensions to ClaSP

ClaSP is extended to incorporate a more elaborate view of blocks, since it considers that a block produces all its outputs once it receives all its inputs. There are, however, even basic blocks, like the unit delay, which can produce its output before it receives its input. (This is currently handled by assuming some arbitrary input.) Although ClaSP models all the possible flows of execution, it cannot show the relationship between the order of input signals and an output value. This means that some information about parallelism in a Simulink diagram can be lost making automated verification impossible in some circumstances.

We use Z to characterise the form of the output of the extended version of ClaSP. Again, it is not actually a CSP process, but information about the structure of the diagram that is used to define the *Circus* specification.

We use given sets *NAME*, *Signal*, and *Block* to represent the valid specification names, and the sets of signal and block names used in the diagram. For a given diagram, the output produced by ClaSP gives the name of the diagram, its inputs and outputs, and a characterisation of each of its blocks.

ClaSPOutput

$$
\begin{array}{l}
spec : NAME \\
inputs, outputs : \mathbb{P}\ Signal \\
blocks : Block \rightarrow BlockWiring
\end{array}
$$

The wiring of a block defines its inputs, outputs, and the dependencies between

them; these determine the independent flows of execution that can arise to calculate different outputs.

Values of a free type *Enabled* are used to record whether a flow of execution is *always* enabled or enabling depends on the values of some special input signals: $Enabled ::= always \mid esigs << \mathbb{P}\, Signal >>$. In a flow, the order in which the signals are received may be relevant. We also need to know the signals that a flow requires (*rinps*), and the outputs that it produces (*pouts*).

$$Flow \mathrel{\widehat=} [\, enabled : Enabled;\ ordered : BOOL;\ rinps, pouts : \mathbb{P}\, Signal\,]$$

The block wiring information includes the order of the inputs and outputs to establish a correspondence between the inputs and outputs of the ClawZ schema that defines the functionality of the block and the signals in the diagram.

BlockWiring

$$inps, outs : \operatorname{seq} Signal$$
$$flows : \mathbb{P}\, Flow$$

$$\forall f : flows \mid f.enabled \in \operatorname{ran} esigs \bullet (esigs^{\sim} f.enabled) \subseteq \operatorname{ran} inps$$
$$(\forall f : flows \bullet f.rinps \subseteq \operatorname{ran} inps) \land \bigcup \{ f : flows \bullet f.pouts \} = \operatorname{ran} outs$$
$$\forall f_1, f_2 : flows \bullet f_1 \neq f_2 \Rightarrow f_1.pouts \cap f_2.pouts = \varnothing$$

The invariant establishes that the enabling signals and the required inputs of a flow are inputs of the block, and every output of the diagram is an output of a flow. For inputs, we do not have the same restriction, as there may be inputs that are not required to produce outputs; a unit delay block is a simple example. Finally, different flows should produce distinct outputs.

Part of the extended ClaSP output for the PID diagram is in Figure 5. The blocks are very simple: they have one flow, which is always enabled, and whose output does not depend on the input order. The constants are also blocks, with no inputs, and just one output. Even though blocks like the Differentiator represent a diagram, from the point of view of the PID, it is just a block. The internal communications that take place inside the Differentiator are ignored.

This does not mean, however, that ClaSP does not need to inspect the subsystems to determine the model of a diagram. A subsystem can, for example, have several flows of execution, or have a behaviour that depends on the order of the inputs are received. This information can only be determined by applying ClaSP to the blocks of the subsystem.

6 Translation strategy

The starting points for the translation are a *ClaSPOutput* which we call clasp, and a Z specification, called clawz, produced by the extended version of ClawZ. We refer to a definition D in clawz as clawz.D.

The *Circus* specification of a diagram first declares all signals as channels. It also declares a synchronisation channel *end_cycle*; after taking all its inputs

$$
\begin{array}{l}
\langle\!| \mathit{spec} \mapsto \mathit{pidspec},\\
\quad \mathit{inputs} \mapsto \{\, \mathit{FMVPE}, \mathit{DFMVGI}, \mathit{FMVPV}, \mathit{DFMVGF},\\
\qquad\quad \mathit{DFM2MN}, \mathit{DFM2MX}, \mathit{CFMCMX}, \mathit{CFMCMN} \,\},\\
\quad \mathit{output} \mapsto \{\, \mathit{FMTMCD} \,\},\\
\quad \mathit{blocks} \mapsto \{\, \mathit{Differentiator} \mapsto \langle\!| \mathit{inps} \mapsto \langle\, \mathit{FMVPE}, \mathit{Constant1\_out} \,\rangle,\\
\qquad\qquad\qquad\qquad \mathit{outs} \mapsto \langle\, \mathit{Differentiator\_out} \,\rangle\\
\qquad\qquad\qquad\qquad \mathit{flows} \mapsto \{\, \langle\!| \mathit{enabled} \mapsto \mathit{always}, \mathit{ordered} \mapsto \mathit{false}\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{rinps} \mapsto \{\, \mathit{FMVPE}, \mathit{Constant1\_out} \,\},\\
\qquad\qquad\qquad\qquad\qquad\qquad \mathit{pouts} \mapsto \{\, \mathit{Differentiator\_out} \,\} |\!\rangle \,\} |\!\rangle,\\
\qquad\quad \mathit{Constant1} \mapsto \langle\!| \mathit{inps} \mapsto \langle\,\rangle, \mathit{outs} \mapsto \langle\, \mathit{Constant1\_out} \,\rangle\\
\qquad\qquad\qquad\quad \mathit{flows} \mapsto \{\, \langle\!| \mathit{enabled} \mapsto \mathit{always}, \mathit{ordered} \mapsto \mathit{false}\\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{rinps} \mapsto \{\,\},\\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{pouts} \mapsto \{\, \mathit{Constant1\_out} \,\} |\!\rangle \,\} |\!\rangle,\\
\qquad\quad \mathit{Sum1} \mapsto \langle\!| \mathit{inps} \mapsto \langle\, \mathit{Differentiator\_out}, \mathit{Sum2\_out} \,\rangle, \mathit{outs} \mapsto \langle\, \mathit{Sum1\_out} \,\rangle\\
\qquad\qquad\quad \mathit{flows} \mapsto \{\, \langle\!| \mathit{enabled} \mapsto \mathit{always}, \mathit{ordered} \mapsto \mathit{false}\\
\qquad\qquad\qquad\qquad\quad \mathit{rinps} \mapsto \{\, \mathit{Differentiator\_out}, \mathit{Sum2\_out} \,\},\\
\qquad\qquad\qquad\qquad\quad \mathit{pouts} \mapsto \{\, \mathit{Sum1\_out} \,\} |\!\rangle \,\} |\!\rangle, \ldots \,\} |\!\rangle
\end{array}
$$

Fig. 5. Extended ClaSP output for the PID

and producing all its outputs, each block of a diagram waits to synchronise on *end_cycle* before proceeding to the next cycle. In this way, all blocks are kept in phase. The *Circus* specification corresponding to the PID starts as follows.

$$
\begin{array}{l}
\textbf{channel}\ \mathit{FMVPE}, \mathit{Differentiator\_out}, \ldots, \mathit{CFMCMX}, \mathit{CFMCMN}, \ldots : \mathbb{U}\\
\textbf{channel}\ \mathit{end\_cycle};
\end{array}
$$

Next, the *Circus* specification includes the ClawZ library, which is used in clawz.

6.1 The diagram

Blocks and diagrams are defined as processes. The whole diagram is a process called clasp.spec, which is defined as the parallel execution of all the blocks.

$$
\begin{array}{l}
\textbf{process}\ \mathsf{clasp.spec} \mathrel{\widehat=}\\
\quad (\ \|\| \mathsf{B} : \mathsf{Block} \bullet \mathsf{B}) \setminus (\ \mathsf{Signal} \setminus (\mathsf{clasp.inputs} \cup \mathsf{clasp.outputs})\ )
\end{array}
$$

The alphabet of each block includes its inputs and outputs, and *end_cycle*. For conciseness, we use sets and sequences of signals to define channel sets in *Circus*.

$$\alpha\mathsf{B} = \mathrm{ran}(\mathsf{clasp.blocks\ B}).\mathsf{inps} \cup \mathrm{ran}(\mathsf{clasp.blocks\ B}).\mathsf{outs} \cup \{\, \mathit{end\_cycle} \,\}$$

The synchronisation required by the parallelism determines the possible flows of execution for the diagram. For the *PID*, we have the process sketched below.

$$
\begin{array}{l}
\textbf{process}\ \mathit{pidspec} \mathrel{\widehat=}\\
\quad (\mathit{Differentiator}\ \{|\mathit{FMVPE}, \mathit{Constant1\_out}, \mathit{Differentiator\_out}, \mathit{end\_cycle}|\}\\
\qquad\quad ||\\
\qquad \mathit{Sum1}\ \{|\ \mathit{Differentiator\_out}, \mathit{Sum2\_out}, \mathit{Sum1\_out}, \mathit{end\_cycle}\ |\} \ldots)\\
\qquad\quad \setminus\ \{|\ \mathit{Constant1\_out}, \mathit{Differentiator\_out}, \mathit{Sum2\_out}, \ldots, \mathit{end\_cycle}\ |\}
\end{array}
$$

The processes that represent the Differentiator and the Sum1 blocks are required

to synchronise on the channels *Differentiator_out* and *end_cycle* (the intersection of their alphabets); the processes for Sum1 and Limit2 are required to synchronise on *Sum*1*_out* and *end_cycle*; and so on. Because the internal channels are hidden, in an implementation, we do not need to have a separate process for each block; refinement can lead to combination and splitting of blocks.

6.2 The blocks

The process that corresponds to a block B is defined explicitly, independently of whether the block is simple, like *Sum*1, or a subsystem, like *Differentiator*. In clasp we have a record the outputs of a subsystem that may be produced independently and in parallel, but not of internal communications. For example, to model the interaction between the blocks of the Differentiator in Figure 2, we need to translate that diagram; the translation of the PID diagram in Figure 1 does not include them. In the next section we discuss the relation between the *Circus* process that models the Differentiator in the translation of the PID and the *Circus* process obtained by translation the Differentiator diagram.

We first consider the translation of a block whose flows are always enabled and do not depend on the order of the inputs. The state of the B process includes a component for each component named *state* used in the definition of B in clawz.

process B $\widehat{=}$ **begin**

state B_*State*

def1_*state* : T1; ...defn_*state* : Tn

Each defi is a definition in clawz such that clawz.B involves defi, and defi is a set of bindings with a component of type Ti called *state*. We define formally what it means for clawz.B to involve defi.

Definition 1. *A type T_1 involves a type T_2 if and only if (i) $T_1 = T_2$; or (ii) exits a type T_3 such that $T_1 = \mathbb{P}\, T_3$, and T_3 involves T_2; or (iii) there are types $T_3, \ldots, T_n$, such that $T_1 = T_3 \times \ldots T_n$, and any of the T_i involves T_2; or (iv) T_1 is a schema with a component whose type involves T_2.*

For example, the schema *pidspec_Differentiator* characterises the PID Differentiator; it has a component *UnitDelay* of type *pidspec_Differentiator_UnitDelay*, which is a set of bindings with a component called *state* defined by *UnitDelay_g*. So, the process *pidspec_Differentiator* defined in Figure 6 has a state component called *pidspec_Differentiator_UnitDelay_state*.

After the state declaration, we include clawz.B and all the definitions in clawz that it uses. The initialisation of the state is based on the clawz specification.

Init

B_*State*′

$\exists\, b$: defi • defi_*state*′ = b.*initial_state*

A component defi_*state*, corresponding to a *state* component of a definition defi

process $pidspec\_Differentiator \mathrel{\widehat=}$ **begin**

state

$pidspec\_Differentiator\_State \mathrel{\widehat=} [\, pidspec\_Differentiator\_UnitDelay\_state : \mathbb{U} \,]$

$pidspec\_Differentiator\_UnitDelay$ from Figure 3 and other definitions it uses.

Init

$pidspec\_Differentiator\_State'$

$\exists\, b : pidspec\_Differentiator\_UnitDelay \bullet$
$\quad pidspec\_Differentiator\_UnitDelay\_state' = b.initial\_state$

Calculate_pidspec_Differentiator

$\Delta pidspec\_Differentiator\_State;\ In1?, In2?, Out1! : \mathbb{U}$

$\exists\, b : pidspec\_Differentiator \bullet$
$\quad b.In1? = In1? \land b.In2? = In2? \land$
$\quad b.UnitDelay.state = pidspec\_Differentiator\_UnitDelay\_state \land$
$\quad b.UnitDelay.state' = pidspec\_Differentiator\_UnitDelay\_state' \land$
$\quad b.Out1! = Out1!$

$Calculate\_pidspec\_Differentiator\_out \mathrel{\widehat=}$
$\quad Calculate\_pidspec\_Differentiator \setminus (pidspec\_Differentiator\_UnitDelay\_state') \land$
$\quad \Xi pidspec\_Differentiator\_State$

$Execute\_Differentiator\_out \mathrel{\widehat=}$
$\quad$ **var** $In1, In2 : \mathbb{U} \bullet$
$\quad\quad (FMVPE?x \rightarrow In1 := x) \llbracket \{In1\} \mid \{In2\} \rrbracket (Constant1\_out?x \rightarrow In2 := x);$
$\quad\quad$ **var** $Out1 : \mathbb{U} \bullet$
$\quad\quad\quad Calculate\_pidspec\_Differentiator\_out;\ Differentiator\_out!Out1 \rightarrow Skip$

$Calculate\_pidspec\_Differentiator\_State \mathrel{\widehat=}$
$\quad Calculate\_pidspec\_Differentiator \setminus (Out1!)$

$StateUpdate \mathrel{\widehat=}$
$\quad$ **var** $In1, In2 : \mathbb{U} \bullet$
$\quad\quad (FMVPE?x \rightarrow In1 := x) \llbracket \{In1\} \mid \{In2\} \rrbracket (Constant1\_out?x \rightarrow In2 := x);$
$\quad\quad Calculate\_pidspec\_Differentiator\_State;$

$\bullet\ Init;$
$\quad \mu\, X \bullet (Execute\_Differentiator\_out \llbracket \{\}$
$\quad\quad\quad \mid \{\!| FMVPE, Constant1\_out |\!\} \mid$
$\quad\quad\quad \{ pidspec\_Differentiator\_UnitDelay\_state \} \rrbracket StateUpdate);$
$\quad\quad end\_cycle \rightarrow X$

end

Fig. 6. *Circus* process for the block Differentiator

in clawz, is initialised with the value of the component *initial_state* of that definition. We identify a binding b of type defi, whose value for *initial_state* defines the initial value of defi_*state*. For example, if defi is a unit delay, defi is a set whose bindings all have the same value for *initial_state*: that in the diagram.

The main action starts with the initialisation, and recursively proceeds in parallel to execute each of the flows and update the state, before synchronising on *end_cycle*. The flows proceed independently, but the block can only start a new cycle when all the flows, (and all the blocks of the diagram) have finished.

$$\bullet\ Init;\ \mu X \bullet (\ Flows\ [\![\ \{\ \} \mid \mathsf{rInps} \mid \{\!|\alpha \mathsf{B\_}State|\!\}\ ]\!]\ StateUpdate\ );\ end\_cycle \rightarrow X$$
end

The flows do not update the state, and so the action *Flows* is associated with the empty set of state component names; on the other hand, *StateUpdate* is associated with the set B_State including all state components. When an input is received, it needs to be made available to the flows and to the action that updates the state, and so they synchronise. The set rInps contains all the inputs required by at least one flow of B.

$$\mathsf{rInps} \mathrel{\widehat=} \bigcup \{\, \mathsf{f} : (\mathsf{clasp.blocks\ B}).\mathsf{flows} \bullet \mathsf{f.rinps} \,\}$$

As already observed, not all inputs are required by a flow; the input of a unit delay block is a simple example.

The action *Flows* executes the flows in (clasp.blocks B).flows in parallel.

$$Flows \mathrel{\widehat=} |||\, \mathsf{f} : (\mathsf{clasp.blocks\ B}).\mathsf{flows}\, \{\ \} \mid \mathsf{f.rinps} \cup \mathsf{f.pouts} \bullet Execute\_\mathsf{f}$$

They do not change any of the state components; they only produce outputs. Their alphabets are the required inputs and the produced outputs.

In the Differentiator, there is only one flow, so the interleaving in *Flows* is reduced to a single process *Execute_Differentiator_out* (Figure 6). It synchronises with the action *StateUpdate* on the inputs *FMVPE* and *Constant*1_*out*.

For each flow f, the action *Execute*_f takes the required inputs, and then calculates and produces the outputs.

$$\begin{aligned} Execute\_\mathsf{f} \mathrel{\widehat=} \ & \mathbf{var}\ \mathsf{Ini} : \mathbb{U} \bullet \\ & |||\, \mathsf{inp} : \mathsf{f.rinps}\, \{\, \mathsf{Ini} \,\} \bullet \mathsf{inp}?x \rightarrow \mathsf{Ini} := x; \\ & \mathbf{var}\ \mathsf{Outj} : \mathbb{U} \bullet \\ & \quad \mathsf{CalculateOutputs};\ |||\, \mathsf{out} : \mathsf{f.pouts} \bullet \mathsf{out!Outj} \rightarrow Skip \end{aligned}$$

First, *Execute*_f declares local variables to record the values of the inputs; we declare a variable Ini when the i-th input is required by the flow: (clasp.blocks B).inps i $\in$ f.rinps. Similarly, to calculate the outputs, *Execute*_f declares variables Outj for each output produced by f: those in f.pouts. In *Execute_Differentiator_out* there are two input variables $In1$ and $In2$, and one output variable $Out1$.

The inputs are received in any order, through each of the channels inp in f.rinps. The value x of the input is recorded in the corresponding variable Ini.

Similarly, outputs are sent in any order through the channels in f.pouts. In our example, since there is only one output, the interleaving is reduced to one action.

The definition clawz.B specifies the state changes and the outputs of B, but it is not an operation over the state B_*State*. We define a schema *Calculate*_B that lifts clawz.B to B_*State*. It includes the input and output variables; Z decorations are used, since *Circus* allows us to keep the Z style and refer to local variables as inputs or outputs. In *Calculate*_B, we identify a binding b of type clawz.B using the input values in Ini to determine the value of the Ini? components of b, and the _*state* components to determine the value of the corresponding components of b. The new value of the state and the output are defined by b.

*Calculate*_B

$$\Delta \mathsf{B\_}State;\ \mathsf{Ini?}, \mathsf{Outj!} : \mathbb{U}$$

$$\exists b : \mathsf{clawz.B} \bullet b.\mathsf{Ini?} = \mathsf{Ini?} \land b.\mathsf{defi}.state = \mathsf{defi\_}state \land b.\mathsf{defi}.state' = \mathsf{defi\_}state' \land b.\mathsf{Outj!} = \mathsf{Outj!}$$

If B has a state component defi_*state*, it is because clawz.B includes a component defi with a *state* component. To define the schema CalculateOutputs, we hide the final value of the state in *Calculate*_B, and conjoin the result with Ξ B_*State* to establish that no state component is modified.

The action that updates the state takes all the inputs.

$$StateUpdate \mathrel{\widehat=} \mathbf{var}\ \mathsf{Ini} : \mathbb{U} \bullet \|\!|\ \mathsf{inp} : (\mathsf{clasp.blocks\,B}).\mathsf{inps}\{\ \mathsf{Ini}\ \} \bullet \mathsf{inp?}x \rightarrow \mathsf{Ini} := x; \mathsf{CalculateState};$$

In principle, all the inputs in (clasp.blocks B).inps are needed. The definition of CalculateState uses *Calculate*_B; it simply hides the output variables. An example is presented in Figure 6: *Calculate_pidspec_Differentiator_State*.

6.3 Enabling conditions and order of inputs

For flows that have enabling conditions or depend on the order of the inputs, *Execute*_f needs to be changed. For lack of space, we do not present the definitions in detail. To capture the order of the inputs, the interleaving in *Execute*_f needs to be replaced with a recursive action that takes any of the outstanding inputs at each step and records its value and index in a sequence. It terminates once all inputs have been received. The resulting sequence of indexes is used as an extra parameter for the calculation of outputs and state updates.

The presence of action and enabled subsystems leads to the possibility that some outputs are not computed. In this case, for every output signal o, we need two channels: o, as explained before, and oComputed of type *BOOL*. The communication of outputs in *Execute*_f needs to be defined as follows.

$$\|\!|\ \mathsf{o} : \mathsf{f.pouts} \bullet \mathsf{o!Outj} \rightarrow \mathsf{oComputed!}true \rightarrow Skip$$

If o is an internal channel, so should be oComputed. If f is a flow that is not

always enabled, it needs to use the *_Enabling* schema produced by ClawZ to determine whether an output should be computed or not. Blocks that need that information should declare oComputed in its alphabet.

7 Refinement

In the translation of a diagram, a block that corresponds to a subsystem is regarded mostly as a black box. As already said, even though we consider flows of execution and requirements to record the order of arrival of the inputs of a subsystems, we do not model its internal communications. We can, however, translate the diagram that corresponds to a subsystem. For example, in the PID diagram, Differentiator is a block; in the translation of the PID, it is defined as a the single process (Figure 6). If, on the other hand, we consider the diagram that specifies this block (Figure 2), we get the following *Circus* output.

$$
\begin{array}{l}
\textbf{process}\ \mathit{Differentiator} \mathrel{\widehat=} \\
\quad (\mathit{Sum}\{\!|\ a, b, \mathit{Sum\_out}, \mathit{end\_cycle}\ |\!\} \\
\qquad ||\\
\quad \mathit{Product}\{\!|\ c, \mathit{Sum\_out}, \mathit{output}, \mathit{end\_cycle}\ |\!\} \\
\qquad ||\\
\quad \mathit{UnitDelay}\{\!|\ a, b, \mathit{end\_cycle}\ |\!\}) \setminus \{\!|\ \mathit{Sum\_out}, b\ |\!\}
\end{array}
$$

For lack of space, we have to omit the processes *Sum*, *Product*, and *UnitDelay* that model the blocks in Figure 2. This new process refines *pidspec_Differentiator* in Figure 6, given that the channels are renamed properly.

$$
\begin{array}{l}
\quad \mathit{pidspec\_Differentiator} \\
\sqsubseteq \\
\quad \mathit{Differentiator}[a, \mathit{output} := \mathit{FMVPE}, \mathit{Differentiator\_out}]
\end{array}
$$

The renaming is needed because the diagram of a block does not keep the original names of inputs and outputs. The *Circus* refinement calculus can be used to prove this refinement; it is a typical derivation of a distributed implementation from a centralised specification. The state does not require refinement; the major effort is in expressing the recursive main action of *pidspec_Differentiator* as a parallelism. In [15] we tackle a similar problem in an industrial case study.

A refinement relationship should hold every time we translate a diagram and a subsystem corresponding to one of its blocks. The implementation obtained follows the architecture of the diagram, with a process for each of the blocks. As already said, however, this is not the only possible implementation.

Refinement can also be used to reason about diagrams. For example, an action subsystem that takes its input from a block whose output always satisfies the condition of the action subsystem can become a simple subsystem. To prove that, we can calculate the *Circus* model, refine it to simplify the process that defines the action subsystem, and translate it back to a diagram. We can use the same approach to eliminate unnecessary blocks. To make this approach appealing to engineers, however, we need to provide a lot of automation. The algebraic approach of a refinement calculus is, therefore, very appropriate.

8 Conclusions

We have presented a semantics for discrete-time Simulink diagrams using a combination of Z and CSP called *Circus*. Our model captures the functionality of a diagram over any number of cycles, and the inherent parallelism between blocks. Cyclic diagrams involving feedback loops are also covered. There are several combinations of Z with a process algebra [8]; *Circus* is distinctive in its refinement theory. Our semantics opens the possibility of reasoning about control law diagrams using refinement. We discussed some examples, based on a PID controller.

PID controllers are considered in [3], where weakest preconditions are used for reasoning about control systems; the technique can be extended to handle static analysis of programs and concurrency. In [12], Mahony used Isabelle/HOL tools to mechanise an assertion reasoning technique based on predicate transformers for dataflow networks with feedback loops. This is a graphical notation like control law diagrams; however, parallelism needs to be indicated explicitly.

The technique proposed in [4] is a Hoare logic to reason about the frequency response of continuous-time control systems. Continuous systems are also considered in [10], with a focus on timing analysis, as opposed to functionality and concurrency. Our interest is on program verification, rather than system analysis, but extension of our model to include multirate diagrams is in our plans.

We are working on the implementation of CliC, a tool to automate the translation strategy presented here. We are also working on a theorem prover and a model checker for *Circus*, all based on ProofPower. These tools will be a powerful resource in the analysis of control diagrams and their implementation.

In [5], a translation from discrete-time Simulink diagrams to Lustre is presented. It formalises the typing system of Simulink and type-checks diagrams before the translation; it also handles multirate diagrams. The results seem to be complementary to those obtained with ClawZ, which assumes that all signals have type double, and can only cope with single rate diagrams, but with a larger number of block types. Lustre is a functional programming language, and ClawZ aims at supporting verification by refinement of Ada programs.

Additional experience with refinement of *Circus* models for control law diagrams will lead to a suite of refinement laws that are adequate to this domain of application. For example, powerful laws should be available to prove the refinement of *pidspec_Differentiator* discussed in the previous section. The proposal, proof, and tool support for the application of these laws is in our agenda of work.

A Simulink model can include a stateflow block, which is defined by a diagram that has local data and includes finite state machines, flow-diagram notations, and state-transition diagrams. The finite state machine reacts to events triggered in the Simulink model; the reactions lead to state changes that affect the behavior of the Simulink model. Stateflow diagrams are studied in [18, 17]. We will investigate the use of *Circus* to model stateflow diagrams; it seems promising as *Circus* can cope with both the data and reactive aspects of the problem. Ultimately, we want to cover the whole of the Simulink notation in a uniform framework for program verification based on *Circus*.

Acknowledgements

This work is funded by the Royal Society. We discussed it with Mark Adams, Alfred Smith, Ian Toyn, Karen Stephenson, Gaius Wilson, and Jim Woodcock. We are also grateful to anonymous referees for useful suggestions.

References

1. *The MathWorks. Simulink*. `http: //www.mathworks.com/products/simulink`.
2. R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *ICFEM 2000*, pages 169 – 176. IEEE Press, 2000.
3. J. Blow and A. Galloway. Generalised Substitution Language and Differentials. In *ZB 2002*, volume 2272 of *LNCS*, pages 396 – 415. Springer-Verlag, 2002.
4. R. J. Boulton, R. Hardy, and U. Martin. 6th International Workshop on Hybrid Systems. In *A Hoare-Logic for Single-Input Single-Output Continuous-Time Control Systems*, volume 2623 of *LNCS*, pages 113 – 125. Springer-Verlag, 2003.
5. P. Caspi, A. Curic, A. Maignan, C. Sofronis, and S. Tripakis. Translating Discrete-Time Simulink to Lustre. In *EMSOFT 2003*, volume 2855 of *LNCS*, pages 84 – 99. Springer-Verlag, 2003.
6. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
7. A. L. C. Cavalcanti and J. C. P. Woodcock. ZRC—A Refinement Calculus for Z. *Formal Aspects of Computing*, 10(3):267—289, 1999.
8. C. Fischer. How to Combine Z with a Process Algebra. In *ZUM'98*. Springer-Verlag, 1998.
9. C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.
10. M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, and R. Ernst. Embedded System Design using the SPI Workbench. In *3rd International Forum on Design Languages*, 2000.
11. D. J. King, R. D. Arthan, and I. C. L. Winnersh. Development of Practical Verification Tools. *ICL Systems Journal*, 11(1), 1996.
12. B. Mahony. Workshop on Formalising Continuous Mathematics. In *The DOVE Approach to the Design of Complex Dynamic Processes*, pages 167 – 187, 2002.
13. C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.
14. C. O'Halloran and A. Smith. Verification of Picture Generated Code. In *ASE 1999*, pages 127 – 136. IEEE Press, 1999.
15. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Refining Industrial Scale Systems in Circus. In *CPA 2004*, pages 281–309. IOS Press, September 2004.
16. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.
17. C. Spencer. Model Checking for Stateflow Diagram with Floating Point Variables and Complex Expressions. Master's thesis, Carnegie Mellon University, 2002.
18. A. Tiwari. Formal Semantics and Analysis Methods for Simulink Stateflow Models. Technical report, SRI International, 2002. `http://www.csl.sri.com/~tiwari/-stateflow.html`.
19. J. C. P. Woodcock and A. L. C. Cavalcanti. The Semantics of *Circus*. In *ZB 2002*, volume 2272 of *LNCS*, pages 184—203. Springer-Verlag, 2002.
20. J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refinement, and Proof*. Prentice-Hall, 1996.