Verification of Control Systems using *Circus*

Ana Cavalcanti
Department of Computer Science
University of York
York - England
Ana.Cavalcanti@cs.york.ac.uk

Phil Clayton
Systems Assurance Group, QinetiQ
Malvern, England
p.clayton@eris.qinetiq.com

Abstract

The design of control systems is usually based on diagrammatic definitions of control laws. The independent use of Z and CSP to verify their implementations has been successful, even for very large applications; high levels of automation have been achieved with tools based on a theorem prover called ProofPower. We have extended this approach to integrate the use of Z and CSP using a notation called Circus; as a result, we can handle a larger set of diagrams and prove more properties of the implementation. In this paper, we show how we can reuse the existing tools and experience to provide automation in the context of the new technique. This gives us confidence in its applicability in industry.

Keywords: *Z, CSP, Simulink, refinement.*

1. Introduction

Typically, control systems are designed using control law diagrams, which are graphs of blocks. The connections in the graph represent wires that carry signals, and the blocks represent functions on the values of these signals: they determine the values output through outgoing wires in terms of the values input through incoming wires. A diagram can have a continuous or a discrete time model; we work with discrete-time diagrams, in which signals are sampled at fixed intervals, so that input and output occur in cycles.

Numerical modelling and simulation are the main techniques for validation of control laws. More recent work has proposed the use of logic [6, 12]. We use refinement for the verification of implementations, rather than analysis of diagrams. For that, we use a model of discrete control law diagrams based on first order logic (Z [20]) and a process algebra (CSP [11, 17]). Our objective is to extend an approach to verification of control systems that has been successfully applied in industry. We aim at covering a larger set of diagrams and program properties, but still allow reuse of the expertise on languages and tools already available.

Simulink [1] is practically a standard as a tool to draw and analyse control law diagrams. A translator from discrete-time Simulink diagrams to Z specifications is presented in [3]; it is called ClawZ. Extensive experience with its application for the verification of sequential Ada subprograms (procedures and functions) used in implementations of control systems is already available at QinetiQ [2]. The Z specifications generated by ClawZ are used to formulate refinement conjectures that can be proved using tools based on ProofPower [13]. This is a theorem prover for Z whose powerful tactics allow a high degree of automation.

The Z model of a diagram defined by ClawZ captures the functionality of one cycle; it defines the outputs of the diagram as a function of its inputs. Concurrency is not captured, and, in fact, the computation embedded in the blocks can be carried out concurrently, with order imposed only by the wiring. To verify the architecture of parallel implementations of control law diagrams, QinetiQ employ CSP and model checking.

As an extension of this work, we proposed a *Circus* model for discrete-time Simulink diagrams [8]. *Circus* [19] is a combination of Z, CSP, and the refinement calculus [15]. We defined a translation strategy to convert the output generated by an extended version of ClawZ into a *Circus* specification, using additional graph information about the diagram. In the *Circus* model of a diagram, we capture both its functionality and its concurrent behaviour over any number of cycles.

A calculational strategy to develop concurrent programs based on centralised *Circus* specifications is presented in [9]. In [7], we proposed a different strategy to verify that a *Circus* model of an Ada program refines the *Circus* model of a diagram. In this case, typically,

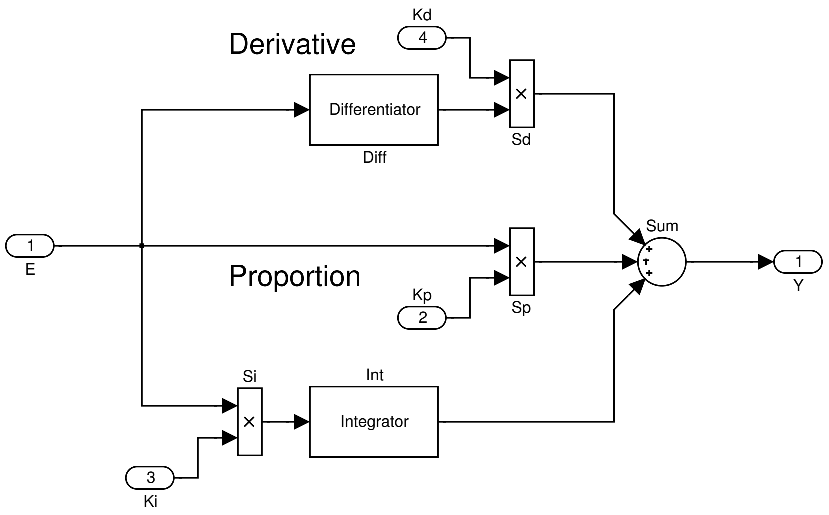

Figure 1. PID (Proportional Integral Derivative) controller

the massive parallelism of the diagram model is reduced to match the architecture of the implementation.

With this technique, independent analyses of the functionality and parallel architecture of an implementation is no longer needed. Yet, in this paper we show that it is possible to reuse the existing technology and expertise on the use of ProofPower and its associated tools in the context of the integrated technique. We provide a strategy based on *Circus* refinement laws to transform the model of the calculations of outputs and of the state updates into appropriate specifications, and show how refinement proof carried out using ProofPower can be used to justify part of the verification entailed by the *Circus* refinement strategy.

In the next section, we give a brief introduction to Simulink diagrams and their *Circus* models. In Section 3, we discuss the features and *Circus* model of typical Ada implementations of control laws. An overview of our refinement strategy is presented in Section 4. In Section 5 we present our approach to reuse of the existing verification technique. Finally, in Section 6, we indicate some future work.

2. Simulink diagrams and *Circus*

Figure 1 gives an example of a simple Simulink diagram for a PID (Proportional Integral Derivative) controller. The rounded numbered boxes are the inputs, namely, E, Kp, Ki, and Kd, and the output, Y. The circle models a sum. The boxes labelled $\times$ are product blocks. The boxes labelled Differentiator and Integrator are subsystems: blocks defined by other diagrams. For the Integrator, for example, the diagram is that in

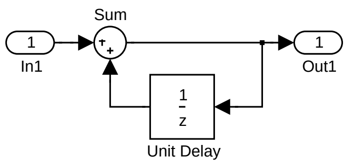

Figure 2. PID Integrator

Figure 2. There, the block labelled 1/z label is a unit delay, which stores the input, and outputs the value stored in the previous cycle: it has a state.

The Z model of a diagram generated by ClawZ uses the notation of ProofPower. It defines blocks as sets of bindings (records), typically specified by schemas. For the block Int, we have the following schema.

```
z
┌ pid_Int ─────────────────────────────
│ In1? : 𝕌; Out1! : 𝕌
│ Sum : pid_Int_Sum;
│ UnitDelay : pid_Int_UnitDelay;
├──────────────────
│ Sum.In1? = In1?;
│ Sum.In2? = UnitDelay.Out1!;
│ Out1! = UnitDelay.In1? = Sum.Out1!
└──────────────────────────────────────
```

The component $In1?$ represents the input of the block, and $Out1!$, its output. The other components represent the blocks in the Integrator diagram: their types are given by the schemas that specify those blocks. The predicate defines the connection of the inputs and outputs, and identifies the input and the output of the Integrator diagram with inputs and outputs of the blocks. The type $\mathbb{U}$ is a universal type in ProofPower. In the case of a simple block, that does not define a subsystem, the only components are inputs and outputs, and the predicate defines how the output can be calculated.

There is a library of blocks in Simulink; correspondingly, there is a library of schemas in ClawZ to define some Simulink library blocks. For example, *pid__Int__Sum* is defined as the ClawZ library schema *Sum_P*2, which models a Sum block with two inputs. For blocks with state, ClawZ includes in the schemas components *state*, *state′*, and *initialstate*; they record the value of the state at the beginning and at the end of the cycle, and at the beginning of the first cycle.

Circus is a language for refinement; it includes specification and programming constructs, which can be freely mixed. For specification, we can use Z and Morgan's specification statements. To model communication and concurrency, we can use CSP. To write code, we can use the language of guarded commands. A *Circus* program is a sequence of paragraphs, just like in Z, but there are also channel and process paragraphs; there is an example in Figure 3.

The blocks, as well as the diagram itself, are defined as processes in *Circus*. In the case of the diagram, the process is a parallel composition of the processes for the blocks;In the case of a block, the process lifts the output of ClawZ to *Circus*. Even if the block is a subsystem, the parallelism in its diagram is ignored. The *Circus* model captures the parallelism only of the top level diagram. For the block Int block, for example, we have the specification in Figure 3; it uses the schema *pid__Int*, which reflects the functionality of the block structure defined in Figure 2, but not its parallelism.

In the complete *Circus* model, we first declare channels to represent the inputs, outputs, and internal wires of the top diagram. We also declare a channel *end_cycle*; it does not have a type, and is used for synchronisation: each block process, after receiving all its inputs and generating all its outputs, synchronises on *end_cycle* before starting the next cycle. After the channel declarations, the *Circus* model includes the ClawZ library, whose definitions are used in the block processes. These come afterwards, before the definition of the diagram, which concludes the specification.

In *Circus*, parallelism is alphabetised. In the definition of the diagram process, the alphabet of the each block process is the set of channels containing the inputs and outputs of the block, and *end_cycle*; like in CSP, the parallel processes synchronise on the intersection of their alphabets. In the *PID* example, *Si* and *Int* are required to synchronise on *end_cycle* and *Si_out*: the output channel of *Si*, which is used as input by *Int*. Since *Si_out* is internal, as it is neither an input nor an output of the diagram, then it is hidden. Because all internal channels are hidden, implementations of the diagram do not need to preserve its block structure; during refinement, we can combine or split blocks.

The specifications of the block processes are similar; Figure 3 presents the process *Int* for the block with the same name. It is a basic process that encapsulates state and exhibits behaviour. The state is defined like in Z, using a schema: *Int_State*; its component is the state of the unit delay block in the Integrator diagram.

The main recursive action at the end defines the behaviour of *Int*. It initialises the state, as defined by the schema *Init*, before iterating to execute the cycles. In *Init*, the variable *pid_Diff_UnitDelay_state′* represents the value of the state component after the initialisation. In the predicate of *Init*, this is defined to be the value of the *initialstate* component of a binding *b* from *pid_Int_UnitDelay*. This is a ClawZ schema also included in *Int*; it models the UnitDelay block of the Integrator using the ClawZ library schema *UnitDelay_g*. It takes as parameter a real number that defines the initial value of the state of the block: *initialstate*. In this case, this value is 0, in accordance with information recorded in the diagram. The notation $0\ e\ 0$ is used to represent 0 as a real number. Besides *pid_Int_UnitDelay*, *Int* includes *pid_Int_Sum* and *pid_Int*; these are the schemas generated by ClawZ for the Integrator.

In each cycle, *Int* calculates and produces its output as defined by *Execute_Int_out*, and in parallel updates the state, as defined by *Int_StateUpdate*. These actions synchronise on *Si_out*, since the input is needed to define the value of the output and of the state.

In a parallelism of actions, to avoid conflict, the disjoint set of variables which each of the parallel actions can modify is fixed. In the parallelism between *Execute_Int_out* and *Int_StateUpdate*, *Execute_Int_out* does not change the state, and *Int_StateUpdate* changes the only state component; this is explicitly indicated by associating them to the empty set of names of variables ($\{\}$) and to the set containing only *pid_Int_UnitDelay_state*. Both actions can access the initial value of *pid_Int_UnitDelay_state*, but only *Int_StateUpdate* can change this value. The same concerns arise for interleaving of actions.

Since *pid_Int* is not an operation over the state of *Int*, we use it to define a state operation: *Calculate_Int*. There, we use Δ*Int_State* to declare *pid_Int_UnitDelay_state* and *pid_Int_UnitDelay_state′* to represent the value of the state component before and after the operation. We also declare the input and out-

$$
\begin{array}{l}
\textbf{process}\ Int \mathrel{\widehat=} \textbf{begin}\\
\quad \textbf{state}\ Int\_State \mathrel{\widehat=} [pid\_Int\_UnitDelay\_state : \mathbb{U}]\\
\quad pid\_Int\_Sum \mathrel{\widehat=} Sum\_P2\\
\quad pid\_Int\_UnitDelay \mathrel{\widehat=} UnitDelay\_g(X0 \mathrel{\widehat=} 0\ e\ 0)
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{pid\_Int}\\
In1? : \mathbb{U};\ Out1! : \mathbb{U}\\
Sum : pid\_Int\_Sum;\ UnitDelay : pid\_Int\_UnitDelay\\
\hline
Sum.In1? = In1? \land Sum.In2? = UnitDelay.Out1! \land UnitDelay.In1? = Sum.Out1!\\
Out1! = UnitDelay.In1?\\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Init}\\
Int\_State'\\
\hline
\exists\, b : pid\_Int\_UnitDelay \bullet pid\_Int\_UnitDelay\_state' = pid\_Int\_UnitDelay.initialstate\\
\hline
\end{array}
$$

$$
\begin{array}{|l}
\hline
\textit{Calculate\_Int}\\
\Delta Int\_State\\
In1?, Out1! : \mathbb{U}\\
\hline
\exists\, b : pid\_Int \bullet b.In1? = In1? \land b.UnitDelay.state = pid\_Int\_UnitDelay\_state \land\\
\qquad\qquad b.UnitDelay.state' = pid\_Int\_UnitDelay\_state' \land b.Out1! = Out1!\\
\hline
\end{array}
$$

$$
\begin{array}{l}
\quad Calculate\_Int\_out \mathrel{\widehat=} Calculate\_Int \setminus (pid\_Int\_UnitDelay\_state') \land \Xi Int\_State\\
\quad Execute\_Int\_out \mathrel{\widehat=} \textbf{var}\ In1 : \mathbb{U} \bullet Si\_out?x \to In1 := x;\ \textbf{var}\ Out1 : \mathbb{U} \bullet Calculate\_Int\_out;\ Int\_out!Out1 \to Skip\\
\quad Calculate\_Int\_state \mathrel{\widehat=} Calculate\_Int \setminus (Out1!)\\
\quad Int\_StateUpdate \mathrel{\widehat=} \textbf{var}\ In1 : \mathbb{U} \bullet Si\_out?x \to In1 := x;\ Calculate\_Int\_state\\
\\
\quad \bullet\ (Init;\ \mu X \bullet (Execute\_Int\_out \mathbin{[\![} \{\,\} \mid \{\!|E|\!\} \mid \{pid\_Int\_UnitDelay\_state\} \mathbin{]\!]} Int\_StateUpdate);\ end\_cycle \to X)\\
\\
\textbf{end}
\end{array}
$$

Figure 3. ***Circus*** **process for the block** Int

put variables; the use of the decorations ? and ! in their names is a Z convention. In the predicate, $In1$ and $pid\_Int\_UnitDelay\_state$ are used to identify a binding b of pid_Int. This binding defines the values of $pid\_Int\_UnitDelay\_state'$ and $Out1!$.

In the specification of $Execute\_Int\_out$, we need an operation that only calculates the output $Out1$, but does not affect the state. We use $Calculate\_Int\_out$, which is specified by hiding $pid\_Int\_UnitDelay\_state'$ in the definition of $Calculate\_Int$, and conjoining the result with the schema $\Xi Diff\_State$ that specifies that $pid\_Int\_UnitDelay\_state$ is not modified. The action $Calculate\_Int\_state$ is used in $Int\_StateUpdate$. It is defined by hiding $Out1!$ in the definition of $Calculate\_Diff$, so that it defines a value for $pid\_Int\_UnitDelay\_state'$, but does not have an output.

This is a simple example. More of the complexity of real diagrams is considered in [8].

3. *Circus* model for Ada implementations

The implementation of a control law diagram is usually composed of subprograms that implement sequentially the functionality of a (group) of blocks, and schedulers for these subprograms. Typically, the cycle is split in time frames and the schedulers allocate the execution of the subprograms to these frames.

In Figure 4 we present the package (module) architecture of an Ada implementation of the PID. The packages `Exec_0`, `Exec_1`, `Exec_2`, and `Exec_3` are the main programs that are executed in parallel. In each of them, there is an initialisation of variables followed by a loop whose iterations last for the duration of a time frame, and schedule some subprograms.

Each main program is associated with a scheduler: `F_Sch`, `Task_1`, `Task_2`, or `Task_3`. The subprograms that implement the blocks are in the package `PID`, which is implemented using another package `Discrete`. The program `Exec_0` does not use `PID` because it only maintains timing information: its schedule, `F_Sch` keeps a frame counter. The other schedulers actually allocate the subprograms of `PID` to frames.

The structure of packages of the Ada program is not preserved in its *Circus* model because the module construct of *Circus*, process, represents an independent flow of execution. In the *Circus* model of the PID implementation, for example, we have a process for each of its main programs: $Exec_0$, $Exec_1$, $Exec_2$, and $Exec_3$.

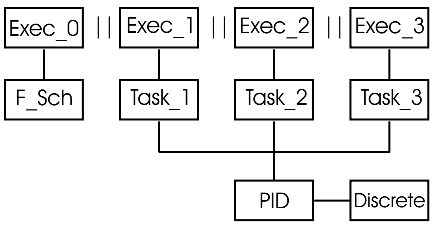

Figure 4. Architecture of the Ada implementation

In each of them, inputs and outputs are communicated via the channels identified in the model of the diagram. Also, new internal channels are created to communicate shared variables; specifically, we need channels *Dsh* and *Ish* to communicate two shared variables *D* and *I*. Finally, the scheduling is modelled using synchronisation on *end_cycle* and on an extra channel *frame*, even though in the Ada program the scheduling uses variables to record time periods and a delay command.

Except for the need to match inputs and outputs to channels, and to identify shared and time variables, most of the *Circus* model can be calculated from the program, but we leave this as future work. Identifying the correspondence between the program variables and the wires of the diagram, and consequently, the channels of the *Circus* model is an activity already undertaken as part of the current verification process.

In Figure 5 we present $Exec_0$. Its state component *cur_f* corresponds to the Ada variable that records the number of the current frame. Its type, *FrameIndex*, contains only the numbers 1 and 2. Since *cur_f* is shared, it is output through the new channel *frame*. The structure of the main action is determined by the number of frames, and the scheduling: after the initialisation, in each frame, its number is output, then a conditional uses that information to invoke the actions to be executed. (In $Exec_0$, no action is scheduled.) In the end of the second frame, there is a synchronisation on *end_cycle*. Finally, the frame number is updated.

The processes $Exec_2$ and $Exec_3$ implement the groups of blocks Si and Int, and Diff and Sd (Figure 1). The process $Exec_2$ is in Figure 6. Its state components are the variables that are used in `Exec_2`; and the actions correspond to the procedures used in `Exec_2`. The frame scheduling is factored out in a procedure *Step*. In contrast with *Exec_0*, the number of the current frame is input, rather than output. Before the procedure *Calc_Integral* is called, its inputs are received in interleaving. Since the variable *I* is shared, its value is communicated at the beginning of the second frame.

The process $Exec_1$ is similar to $Exec_2$ and $Exec_3$.

```
process $Exec_0 \mathrel{\widehat=}$ begin
    state $[cur\_f : FrameIndex]$
    $Init\_F \mathrel{\widehat=} cur\_f := 1$
    $Next\_F \mathrel{\widehat=} cur\_f := (cur\_f \bmod 2) + 1$
    $\bullet\ Init;\ \mu\ X \bullet frame!cur\_f \rightarrow$
                    if$(cur\_f = 1) \rightarrow Skip$
                    $[\!]\ (cur\_f = 2) \rightarrow end\_cycle \rightarrow Skip$
                    fi;
                    $Next\_F;\ X$
end
```

Figure 5. ***Circus*** **model of** `Exec_0`

The main difference is that its *Step* procedure inputs rather than outputs the values of the shared variables; also, it produces the output at the end of the cycle.

The model of the complete program is the parallel composition of $Exec_0$, $Exec_1$, $Exec_2$, and $Exec_3$, synchronising on their common channels; the new channels *frame*, *Dsh*, and *Ish* are hidden. The structure of this rather small example is representative of real control systems in its approach to scheduling and sharing.

4. Refinement strategy

A Simulink diagram has a hierarchical structure. We can assume that, at the top level, there is always a diagram with a single block that takes all the inputs and produces all the outputs. For all interesting examples, this is a single subsystem block that is specified by another diagram. Like for the PID, this may include further subsystem blocks, and, in this case, yet another level of diagrams is used. There is no limit to the number of diagrams that can be used to specify a control law; and they can be organised across several levels.

The *Circus* model of a top level diagram is a single process that lifts the ClawZ output. For the verification of a sequential implementation, this is a suitable model. For the verification of a parallel implementation, however, the *Circus* model of the diagram in the second level is likely to be more adequate. In fact, the best starting point is a model of a diagram in which all blocks that are implemented by different subprograms are explicitly drawn, rather than hidden in subsystem blocks. Typically, to draw this diagram, we need to rearrange the hierarchy of diagrams.

In the case of the PID, the model of the diagram in Figure 1 is adequate to verify the parallel implementation presented in Section 4. Since the implementations of Diff and Int are sequential, there is no need to expand their diagrams. If we did so, the *Circus* model of the diagram would include extra processes and channels corresponding to the blocks and wires in those diagrams. No doubt, this model would be equivalent to that of the smaller diagram in Figure 1, since the ex-

$$
\begin{array}{l}
\textbf{process } \mathit{Exec}_2 \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state } [\mathit{Error}, \mathit{Ki}, I : \mathbb{U}] \\
\quad \mathit{Init\_Integral} \mathrel{\widehat=} I := 0.0 \\
\quad \mathit{Integ} \mathrel{\widehat=} (\textbf{val } \mathit{Input}, K : \mathbb{U};\ \textbf{res } \mathit{Output} : \mathbb{U} \bullet \mathit{Output} := \mathit{Output} + K \times \mathit{Input}) \\
\quad \mathit{Calc\_Integral} \mathrel{\widehat=} \mathit{Integ}(\mathit{Error}, \mathit{Ki}, I) \\
\quad \mathit{Step} \mathrel{\widehat=} \mathit{frame}?f \to \textbf{if}(f = 1) \to ((E?x \to \mathit{Error} := x) \lbrack\!\lbrack \{\mathit{Error}\} \mid \{\mathit{Ki}\} \rbrack\!\rbrack (\mathit{Ki}?x \to \mathit{Kd} := x));\ \mathit{Calc\_Integral} \\
\qquad\qquad\qquad\qquad [\!] \ (f = 2) \to \mathit{Ish}!I \to \mathit{end\_cycle} \to \mathit{Skip} \\
\qquad\qquad\qquad\qquad \textbf{fi} \\
\quad \bullet \ \mathit{Init\_Integral};\ \mu\ X \bullet \mathit{Step};\ X \\
\textbf{end}
\end{array}
$$

Figure 6. ***Circus*** **model of** `Exec_2`

tra channels would be internal. The verification based on it, however, would require more effort to remove the parallelism that is not present in the implementation.

Our verification strategy comprises four phases that progressively collapse the parallelism of the diagram model to match the architecture of the implementation.

- **P1** Write the main action of each block process as a a recursion that iteratively takes the inputs in interleaving, calculates the outputs and updates the state, communicates the outputs in interleaving, and synchronise on *end_cycle*.
- **P2** Collapse the parallelism between the processes of the blocks that are implemented by a single subprogram, and write the main action of the resulting processes in the form described in phase **P1**.
- **P3** For each of the processes created in phase **P2**, introduce the action that specifies the corresponding subprogram, and prove that the calculations can be refined by a call to that action.
- **P4** Collapse the parallelism between the processes that are scheduled in the same task.

Each phase of the strategy can be accomplished applying *Circus* refinement laws. In [7] we explain the procedure. Here, we explain how we can use ProofPower and its associated tools to carry out phase **P3**.

For the PID example, in phase **P1**, the main action of *Int* is written as follows.

$$
\begin{array}{l}
\mathit{Init};\ \mu X \bullet \mathit{Si\_out}?x \to \mathit{In}1 := x; \\
\qquad\qquad \mathit{Calculate\_Int\_out};\ \mathit{Calculate\_Int\_state}; \\
\qquad\qquad \mathit{Int\_out}!\mathit{Out}1 \to \mathit{Skip};\ \mathit{end\_cycle} \to X
\end{array}
$$

The variables *In*1 and *Out*1 are now state components. Since there is only one input and one output, no interleaving is needed.

In phase **P2**, we identify the subprograms that implement blocks. For each that implements more than one, we create a single process by collapsing the parallelism between the processes for the blocks. In `Exec_2`, for example, we identify the procedure `Calc_Integral` which implements Si and Int. We create a process *SiInt* by collapsing *Si* and *Int*, and refine its main action to be written as follows.

$$
\begin{array}{l}
\mathit{Init}; \\
\mu\ X \bullet (E?x \to \mathit{pid\_Si\_In}1 := x) \mathbin{|||} (\mathit{Ki}?x \to \mathit{pid\_Si\_In}2 := x); \\
\qquad \mathit{pid\_Si};\ \mathit{pid\_Int\_In}1 := \mathit{pid\_Si\_Out}1; \\
\qquad \mathit{Calculate\_Int\_out};\ \mathit{Calculate\_Int\_state}; \\
\qquad \mathit{Sd\_out}!\mathit{pid\_Sd\_out} \to \mathit{end\_cycle} \to X
\end{array}
$$

Inputs are taken from *E* and *Ki* in interleaving; for conciseness, we omit the name sets. Afterwards, the calculations of both *Si* and *Int* are performed, and the output of *Sd* is produced. The output of *Si* is the input of *Int*; since this is an internal communication, it is removed.

In phase **P3**, we refine the main action of each process that corresponds to a subprogram: we introduce the declaration and a call to the action that models the subprogram. For that, we match the variables of the program to those of the model. Since the model variables correspond to wires and block states, this means matching the program variables to diagram components.

In the case of *SiInt*, we introduce the definition of *Calc_Integral* as the following call.

$$\mathit{Integ}(\mathit{pid\_Si\_In}1, \mathit{pid\_Si\_In}2, \mathit{pid\_Int\_UnitDelay\_state})$$

Actually, first we introduce *Integ* so that it can be used in *Calc_Integral*. Since *Integ* does not refer to program variables, we define it just as shown in Figure 6.

What we need to prove is the following refinement.

$$
\begin{array}{l}
\quad \left(\begin{array}{l} \mathit{pid\_Si};\ \mathit{pid\_Int\_In}1 := \mathit{pid\_Si\_Out}1; \\ \mathit{Calculate\_Int\_out};\ \mathit{Calculate\_Int\_state} \end{array} \right) \\
\sqsubseteq \\
\quad \mathit{Calc\_Integral}
\end{array} \tag{1}
$$

In the next section, we show how we carry out this proof using ProofPowerZ. With this result, we can write the main action of *SiInt* as follows.

$$
\begin{array}{l}
\mathit{Init}; \\
\mu\ X \bullet (E?x \to \mathit{pid\_Si\_In}1 := x) \mathbin{|||} (\mathit{Ki}?x \to \mathit{pid\_Si\_In}2 := x); \\
\qquad \mathit{Calc\_Integral}; \\
\qquad \mathit{Sd\_out}!\mathit{pid\_Sd\_out} \to \mathit{end\_cycle} \to X
\end{array}
$$

At this point, we can simplify *SiInt* by removing the actions that are no longer used: the ClawZ schemas.

In phase **P4**, we group the subprogram processes in task processes, and rewrite their main actions to use the frame definition.

5. Using ProofPower

Using ProofPower, we can prove that a specification statement that specifies the functionality of a (group of) blocks in terms of the program variables can be refined by Ada commands. The specification statement is constructed using the output of ClawZ for the group of blocks and identifying how the variables of the program correspond to components of the diagram. The Ada commands are those in the body of the subprogram that implements the group of blocks.

For *Integ*, the following conjecture is provable.

$$\Delta Output : \left[\begin{array}{l} \exists\, Integ\_System \bullet \\ \quad Si.In1? = Input \land Si.In2? = K \land \\ \quad Int.UnitDelay.state = Output_0 \land \\ \quad Int.UnitDelay.state' = Output \land \\ \quad Int.Out1! = Output \end{array}\right] \qquad (2)$$

$$\sqsubseteq$$

$$Output := Output + K \times Input$$

In the specification, $\Delta Output$ defines that the parameter *Output* can be modified. The predicate is a quantification over a schema *Integ_System*.

$$\begin{array}{|l} \multicolumn{1}{l}{Integ\_System} \\ \hline Int : pid\_Int;\ Si : pid\_Si \\ \hline Int.In1? = si.Out1! \\ \hline \end{array}$$

This schema is generated by ClawZ if we indicate that *Si* and *Int* form an artificial subsystem: a group of blocks that define a subsystem, even though they do not correspond to a subsystem block. In the predicate of the specification statement in (2), the inputs, output, and state components of the artificial subsystem are identified with parameters of *Integ*. Like in the refinement calculus, $Output_0$ refers to the initial value of *Output*. The refinement conjecture states that such a system can be implemented by the body of *Integ*. Our aim is to use this result to prove the refinement in (1). For that, we carry out the following steps.

1. **Match the procedure parameters to model variables.** Implicitly, we are matching wires and block states in the diagram to procedure parameters.

   Int the example of the *Integ* procedure, *Input* corresponds to *pid_Si_In*1, *K* to *pid_Si_In*2, and *Output* to *pid_Int_UnitDelay_state* and *pid_Int_Out*1.

2. **Introduce the parameter declarations, taking the corresponding model variables as arguments.** In *Circus*, the treatment of parametrised procedures is based on Back's parametrised commands [4], which have been incorporated in a refinement calculus for Z [10]. In this context, we can refine any program to an instantiation of a parametrised command as long as proper renamings are carried. For, the specification in (1), which from now on we name *SiIntC* for conciseness, we proceed as follows.

$$\begin{array}{l} (\mathbf{val}\ Input, K : \mathbb{U};\ \mathbf{vres}\ Output : \mathbb{U} \bullet \\ \qquad SiIntC \left[\begin{array}{l} Input/pid\_Si\_In1?, \\ K/pid\_Si\_In2?, \\ Output/pid\_Int\_UnitDelay\_state, \\ Output'/pid\_Int\_UnitDelay\_state', \end{array}\right] \\ )(pid\_Si\_In1?, pid\_Si\_In2?, pid\_Int\_UnitDelay\_state) \end{array}$$

   The renamings rewrite *SiIntC* in terms of the parameters, but the instantiation of arguments means that the program acts on the model variables, just as *SiIntC* does. We observe that *Output* is only matched to *pid_Int_UnitDelay_state*. Later, we have to record the extra relationship to *pid_Int_Out*1.

3. **Remove state components that are not matched to procedure parameters from the state**. A state component is a variable that is in scope in the main action of a process; we can remove its declaration from the state and introduce a corresponding variable block in the main action.

   In our example, we need to remove *pid_Si_Out*1, *pid_Int_In*1, and *pid_Int_Out*1 from the state. The first two components record the value communicated internally by *Si* to *Int*. This communication is eliminated in a sequential implementation. The last component matches *Output*, but since *Output* matches two variables, one is left out for now.

4. **Reduce the scope of the variable block to include only the calculations and updates**. This can be achieved with standard variable block laws. In our example, the body of the block is reduced to $SiIntC[Input/pid\_Si\_In1?, \ldots]$.

5. **Reduce the calculations and updates to a schema**. This requires specific laws to flatten sequences. One of the them is presented below.

   Law 5.1 (schema-assign-seq)

$$\begin{array}{l} [\Delta S \mid p];\ x := e \\ = \\ [\Delta S \mid \exists\, x_0 \bullet p[x_0/x'] \land x' = e[x_0/x]'] \end{array}$$

   provided $x, x' \in \alpha S$

   It flattens a sequence of a schema that defines an operation over a state *S* that includes a variable *x*, and an assignment to x. The value defined for x by the original schema is given the name x_0 and hidden in the predicate of the resulting schema.

In this step, we use this law and others like this in the following way.

(a) **Join *_out* and *_state* schemas back together.** In our specification, we have the sequence *Calculate_Int_out*; *Calculate_Int_state*. The sequence operator is not the schema calculus operator, but the command constructor. By removing it, we get the schema *Calculate_Int* (Figure 3), which lifts *pid_Int*.

(b) **Eliminate assignments.** For that, we use the law above, or another similar law that applies when the assignment comes first in the sequence. The assignments correspond to passages of values from a block to another through the internal wires. The equality that arises from the assignments in the resulting schema should be kept: they should not, for instance, be used to eliminate quantifiers using the one-point rule. In our example, we keep the equality $pid\_Int\_In1' = pid\_Si\_Out1'$.

(c) **Flatten remaining sequences.** The left sequences involve only schemas. The ClawZ schemas should be incorporated, but not expanded. In our example, we do not expand the schema *pid_Si*: the ClawZ model of *Si*.

The final result of this step for our example is the schema shown below.

$$
\begin{array}{|l}
\textit{Step5} \\
\hline
pid\_Int\_In1, pid\_Int\_In1', \\
pid\_Int\_UnitDelay\_state, \\
pid\_Int\_UnitDelay\_state' \\
pid\_Si\_In1?, pid\_Si\_In2?, \\
pid\_Si\_Out1!, pid\_Int\_Out1! : \mathbb{U} \\
\hline
pid\_Si \land pid\_Int\_In1' = pid\_Si\_Out1' \\
\left(\begin{array}{l}
\exists b : pid\_Int \bullet \\
\quad b.In1? = pid\_Int\_In1' \\
\quad b.UnitDelay.state = \\
\qquad pid\_Int\_UnitDelay\_state \\
\quad b.UnitDelay.state' = \\
\qquad pid\_Int\_UnitDelay\_state' \\
\quad b.Out1! = pid\_Int\_Out1!
\end{array}\right) \\
\hline
\end{array}
$$

It acts on a state that includes all the variables in scope in the main action. We do not decorate *pid_Int_In1* with a ? because its final value is changed, so it is not an input variable in this schema.

6. **Remove renamings and variable block.** This can be achieved by turning them into an existential quantification: for the renamings, we use the one-point-rule, and for variable blocks, we use the law below.

Law 5.2 (var-schema-hiding)

$$(\mathbf{var}\ x : T \bullet SExp) = SExp \setminus (x, x')$$

A schema hiding is an existential quantification. For our example, the result is sketched below.

$$
\begin{array}{|l}
\textit{Step6} \\
\hline
Output, Output', Input, K : \mathbb{U} \\
\hline
\exists pid\_Int\_In1, pid\_Int\_In1', pid\_Si\_Out1!, \ldots \bullet \\
\quad pid\_Si\_In1? = Input \land pid\_Si\_In2? = K \land \\
\quad \ldots \land \\
\quad \ldots\text{predicate of } \textit{Step5}\ldots \\
\hline
\end{array}
$$

7. **Rename the dashed *In* variables to use the ? decoration.** This is possible because the undashed variables are not free in the quantified predicate, so we can eliminate their quantification, and change the name of the dashed variables. In the example, the quantification becomes $\exists pid\_Int\_In1?, pid\_Si\_Out1!, \ldots : \mathbb{U} \bullet \ldots$ We cannot rename pid_Int_In1' without eliminating *pid_Int_In1* beforehand because, in *Circus*, *pid_Int_In1* and *pid_Int_In1?* are the same variable. There are no input and output variables, only state components and local variables; the ? and ! decorations are used for compatibility with Z.

8. **Express quantification using simplified ClawZ schemas.** We introduce schemas that contain just the declaration part of the ClawZ schemas for the blocks. For *pid_Int*, we declare the schema below.

$$
\begin{array}{|l}
\textit{pid\_IntD} \\
\hline
In1?, Out1! : \mathbb{U} \\
Sum : pid\_\_Int\_\_Sum \\
UnitDelay : pid\_\_Int\_\_UnitDelay \\
\hline
\end{array}
$$

We also declare a schema *pid_SiD*.

The quantification can be expressed in terms of these schemas: there is a direct correspondence between the quantified variables and their components. After all, it was these schemas that were used to construct the model in the first place. For example, the component *In1?* of *pid_IntD* corresponds to *pid_Int_In1?*.

Part of the rewritten predicate is shown below.

$$
\begin{array}{l}
\exists\, Int : pid\_IntD;\ Si : pid\_SiD \bullet \\
\ Si.In1? = Input \land Si.In2? = K \land \\
\ \ldots \land \\
\ pid\_Si[Si.In1?/pid\_Si\_In1?, \ldots] \land \\
\ Int.In1? = Si.Out1! \land \\
\left(\begin{array}{l}
\exists\, b : pid\_Int \bullet \\
\quad b.In1? = Int.In1? \\
\quad b.UnitDelay.state = Int.UnitDelay.state \\
\quad b.UnitDelay.state' = Int.UnitDelay.state' \\
\quad b.Out1! = pid.Int.Out1!
\end{array}\right)
\end{array}
$$

9. **Check that the quantified variables are elements of the proper ClawZ schema.** In the example, we need to check that Int and Si are bindings of pid_Int and pid_Si, and not simply pid_IntD and pid_SiD as declared. We need to consider two cases.

   (a) **The block has a state.** In this case, an existential quantification over a binding of the ClawZ schema defines the variable. This is due to the way in which state is handled in the *Circus* model using the state in the ClawZ schemas. In our example, Int gives an illustration.

   Since the blocks are deterministic, the quantified predicate allows us to conclude that the binding is equal to the quantified variable: $b = Int$ in the example. With that, the one-point rule can be used to eliminate the existential quantification and conclude that $Int \in pid\_Int$.

   (b) **The block does not have a state.** In this case, instead of an existential quantification, a renaming of the original schema defines the quantified variable: Si and pid_Si illustrate the situation. The renaming explicitly states that the quantified variable satisfies the property of the ClawZ schema. A more concise way of expressing this is to say that the variable belongs to the type defined by the ClawZ schema. In the example, we can write $Si \in pid\_Si$.

   As a result, we can simplify the predicate of the schema, by giving stronger declarations for the variables. In our example, the result is as follows.

$$
\begin{array}{l}
\exists\, Int : pid\_Int;\ Si : pid\_Si \bullet \\
\ Si.In1? = Input \land Si.In2? = K \land \\
\ Int.UnitDelay.state = Output \land \\
\ Int.UnitDelay.state' = Output' \land \\
\ Int.In1? = Si.Out1!
\end{array}
$$

10. **Transform schema into a specification statement.** This can be accomplished with a law of the Z refinement calculus; they are all valid in *Circus*. The result is almost exactly the specification statement used in the current verification technique (see (2)), but the components and the predicate of the artificial subsystem schema are explicitly included.

11. **Declare the artificial subsystem schema and use it in the specification.** This is a simple application of predicate calculus to change the quantification.

12. **Record missing correspondences between parameters and model variables.** If any of the parameters represented more than one diagram component (or equivalently, model variable), only one of the correspondences was used. We can now strengthen the postcondition of the specification statement to record the others.

    In the example, we did not record the correspondence between $Output$ and pid_Int_Out1. If we strengthen the postcondition of the specification statement with the conjunct $Int.Out1! = Output$, we record the relationship.

13. **Use ProofPower.** The specification statement is now exactly that constructed using the current approach. So, the refinement can be proved using ProofPower; it justifies the replacement of the specification statement with the procedure body.

14. **Introduce the procedure call.** This is now a direct application of the copy rule.

Except for the actual use of ProofPower, and the need to match program variables to diagram components, all the steps of this procedure can be automated.

6. Conclusions

We have presented an approach to reuse existing tools and expertise in the verification of control systems in the context of a technique that covers both the sequential subprograms and the schedulers of an implementation. It is based on a notation that integrates Z and CSP, and, as such, can formalise the functionality and the parallelism in diagrams and their implementations. We use a single model of the diagram, and a unified verification technique.

Our approach is a procedure for application of refinement laws; using a tactic language [16], we can automate it. All the steps of this procedure are justified by refinement laws. If the application of any of them fails, there is an error in the program, or in our interpretation of how the procedures and variables correspond to components of the diagram.

The majority of the proof obligations generated are related to the implementation of the functionality of the blocks. We have shown how these can be discharged using the tools and proof tactics already available. Experience shows that they can discharge 95% to 98% of the proof obligations automatically.

The needed information concerning the relationship between wires of the diagram and variables of the program and of the model, and between blocks and procedures, is also required by the existing verification technique. Therefore, the feasibility of producing this information is already confirmed for real examples.

In [5], a weakest precondition semantics is used to validate Simulink diagrams; a PID controller is also considered as an example. Assertion reasoning is also used in [18] to analyse Stateflow diagrams. The work in [14] considers dataflow networks with feedback looks, which are similar to control diagrams, although parallelism has to be explicitly indicated. The reasoning technique is also based on predicate transformers.

Our example is simple, and further case studies are in our plans for future work. In any case, the PID verification illustrates the issues involved in the use of ProofPower and its associated tools. They are applied in the third phase of our refinement strategy to processes generated in the second phase by collapsing two or more block processes. Our approach handles sequences of assignments and schemas that define the functionality of a group of blocks. The processes generated in the second phase of our strategy are always of this form.

Automation is essential for the success of a verification technique. Diagrams can span over hundreds of pages, and it is not feasible to handle them without tool support. We plan to automate the rest of our strategy using a refinement tool and tactics based on ProofPower.

We are also working on tools for *Circus*. Of particular relevance to verification of control systems is the *Circus* model checker. It follows the approach adopted by FDR for CSP, but integrates the use of theorem proving to handle proof obligations generated by the data part of the *Circus* programs. An important piece of future work is to investigate its application to prove the refinement conjectures generated by our strategy. Currently, the use of FDR is limited by the size of the models of the control diagrams and of the schedulers. The use of *Circus* guides the combined use of traditional techniques of model checking and theorem proving and is likely to lead to a high level of automation.

Acknowledgements

This work is funded by the Royal Society. We discussed it with Colin O'Halloran, and Jim Woodcock.

References

[1] *The MathWorks Site. Simulink Reference Manual.* `www.mathworks.com/products/simulink`.

[2] M. M. Adams and P. B. Clayton. Cost-Effective Formal Verification for Control Systems. In *ICFEM 2005*, volume 3785 of *LNCS*, pages 465 – 479. Springer, 2005.

[3] R. Arthan, P. Caseley, C. O'Halloran, and A. Smith. ClawZ: Control laws in Z. In *ICFEM 2000*, pages 169 – 176. IEEE Press, 2000.

[4] R. J. R. Back. Procedural Abstraction in the Refinement Calculus. Technical report, Department of Computer Science, Åbo, Finland, 1987. Ser. A No. 55.

[5] J. Blow and A. Galloway. Generalised Substitution Language and Differentials. In *ZB 2002*, volume 2272 of *LNCS*, pages 396 – 415. Springer-Verlag, 2002.

[6] R. J. Boulton, R. Hardy, and U. Martin. A Hoare-Logic for Single-Input Single-Output Continuous-Time Control Systems. In *HIS 2003*, volume 2623 of *LNCS*, pages 113 – 125. Springer-Verlag, 2003.

[7] A. L. C. Cavalcanti and P. Clayton. From Control Law Diagrams to Ada via *Circus*. In *FM 2006*, 2006. Submitted.

[8] A. L. C. Cavalcanti, P. Clayton, and C. O'Halloran. Control Law Diagrams in *Circus*. In *FM 2005*, volume 3582 of *LNCS*, pages 253 – 268. Springer-Verlag, 2005.

[9] 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.

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

[11] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[12] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, and R. Ernst. Embedded System Design using the SPI Workbench. In *FDL 2000*, 2000.

[13] D. J. King, R. D. Arthan, and I. C. L. Winnersh. Development of Practical Verification Tools. *ICL Systems Journal*, 11(1), 1996.

[14] B. Mahony. First International Workshop on Formalising Continuous Mathematics. In *The DOVE Approach to the Design of Complex Dynamic Processes*, pages 167 – 187, 2002.

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

[16] M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. ArcAngel: A Tactic Language for Refinement. *Formal Aspects of Computing*, 15(1):28 – 47, 2003.

[17] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[18] I. Toyn and A. Galloway. Proving Properties of Stateflow Models using ISO Standard Z and CADiZ. In *ZB 2005*, volume 3455 of *LNCS*, pages 104 – 123. Springer-Verlag, 2005.

[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*. Prentice-Hall, 1996.