# From control law diagrams to Ada via

Ana Cavalcanti, Phil Clayton, Colin O'Halloran

Under consideration for publication in Formal Aspects of Computing

# From Control Law Diagrams to Ada via *Circus*

Ana Cavalcanti[1], Phil Clayton[2,3], and Colin O'Halloran[2]

[1] University of York, UK
[2] Systems Assurance Group, QinetiQ Malvern, UK
[3] Veonix, Worcester, UK

**Abstract.** Control engineers make extensive use of diagrammatic notations; control law diagrams are used in industry every day. Techniques and tools for analysis of these diagrams or their models are plentiful, but verification of their implementations is a challenge that has been taken up by few. We are aware only of approaches that rely on automatic code generation, which is not enough assurance for certification, and often not adequate when tailored hardware components are used. Our work is based on *Circus*, a notation that combines Z, CSP, and a refinement calculus, and on industrial tools that produce partial Z and CSP models of discrete-time Simulink diagrams. We present a strategy to translate Simulink diagrams to *Circus*, and a strategy to prove that a parallel Ada implementation refines the *Circus* specification; we rely on a *Circus* semantics for the program. By using a combined notation, we provide a specification that considers both functional and behavioural aspects of a large set of diagrams, and support verification of a large number of implementations. We can handle, for instance, arbitrarily large data types and dynamic scheduling.

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

## 1. Introduction

Control systems can be conveniently specified diagrammatically; in particular, engineers are comfortable with control law diagrams . In the avionics and automotive sectors, at least, the use of Matlab's Simulink [40] for drawing and simulation is standard; it also includes facilities for automatic code generation.

Since safety-critical applications often involve control systems, the validation of control law diagrams has been of great interest: numerical modelling and simulation are the techniques routinely used. Formal analysis, due to the typical complexity and scale of diagrams, is a major challenge; it is not unusual for a diagram to have hundreds of pages. Verification of a diagram's implementation is no simpler.

Existing work is mostly concerned with investigation of properties of the specification or design of a control system [52, 20, 33, 19] described by a diagram. They are valuable contributions, in that they extend

*Correspondence and offprint requests to*: Ana Cavalcanti (`Ana.Cavalcanti@cs.york.ac.uk`)

the restricted static analysis capabilities of tools like Simulink. The work in this paper, on the other hand, provides a complementary facility: proof of correctness of code, as opposed to validation of requirements or designs. More precisely, we present a technique to prove that a (parallel) implementation of a diagram satisfies the functional and behavioural properties that it defines. For that, we define a formal model for discrete-time single-rate Simulink diagrams suitable for reasoning based on refinement, a formal model for Ada programs [5] written in a subset of this language similar to SPARK Ada and with a particular architecture, and a verification strategy based on the application of refinement laws to compare them.

As far as we know, the only effort on formal verification of implementations of control laws is ClawZ [4]. This is a translator from Simulink diagrams to specifications written in a version of Z [56] implemented in the theorem prover ProofPower [34]. The Z specifications are used to define refinement conjectures that connect a diagram and an Ada subprogram (procedure or function); they are proved using tools integrated with ProofPower [1]. We have measured experiments in the context of industrial applications that show a reduction factor between two and a half and four and a half in the human effort required for establishing acceptance when the ClawZ approach and tools are used. There is a cost reduction of 20% in relation to conventional development and verification of safety-critical systems in the area of avionics.

In this paper, we build on ClawZ to specify more complete models of diagrams: we capture their inherent parallelism, as well as functionality. We also establish correctness of the scheduler (as well as the procedures and functions). All this is achieved with the same high level of automation of ClawZ.

The Matlab semantics for Simulink is given implicitly by its simulator. Many works provide a formal semantics of various properties of diagrams; there are results using automata [52], the data-flow language Lustre [11], asynchronous processes [32], Hoare logic [10], and timed formalisms [17], to cite a few. What we provide here is not just yet another formal model for Simulink. Our semantics distinguishes itself in that it is appropriate for refinement-based reasoning, and, therefore, program verification.

The use of code generators is appealing, and they are the basis of several development approaches advocated by works on analysis of Simulink models [35, 27]. When code correctness and certification are an issue, however, the use of code generators does not provide enough assurance; verification of the generator or of the generated code is needed. The frequent updates to generators make the cost of their verification prohibitive. In any case, requirements imposed by the target hardware often mean that complex tailored algorithms need to be used, instead of automatically generated code; experience in the automotive industry, for example, is reported in [47]. In this paper, we pursue a cost-effective approach to code verification.

What we present is a sound and practical approach to prove correctness of implementations of control diagrams. In our technique, the formalisms are hidden from engineers, as the verification strategy is amenable to high levels of automation that ensure practicality. We use a refinement technique based on *Circus* [15], a combination of Z and CSP. With an integrated approach, we significantly extend the class of diagrams that can be modelled, and program properties that can be verified.

We provide a strategy to translate the output of an extended version of ClawZ and a graph model that captures the data-flow of the diagram to a *Circus* specification; Figure 1 summarises our approach. In addition, we present a verification technique for parallel Ada implementations based on the result of the translation. Effectively, the translation defines a *Circus* semantics for discrete-time Simulink diagrams; it is a suitable starting point for reasoning based on refinement.

Using *Circus*, we capture the functionality and concurrency of a diagram, including features related to conditional execution and order of interactions. Moreover, the *Circus* specification can capture the behaviour of the system over any number of cycles. With a *Circus* model, scheduling and the data operations can be verified jointly, and so we can cater for sophisticated dynamic scheduling policies. Since we do not rely on model checking, there are no restrictions on the size of data types.

With *Circus*, separate analyses of programs to cover functionality and scheduling independently are not needed. Our approach to verification is based on a *Circus* model of the Ada program, and a refinement strategy based on *Circus* laws. We establish the correctness of both the sequential subprograms, and the overall parallel behaviour. For the subprograms, we reuse the already well-established verification technique based on ClawZ and ProofPower [13], but we cover all the properties verified using ClawZ and much more.

The technique presented here is specific for Ada programs use a specific architecture commonly used in embedded control systems where time is critical and processing resources are limited. The approach, however, can be adapted to different architectural patterns. In addition, there are no assumptions about the structure of the diagrams, and the verification is entirely compositional.

In practice, many of the changes to the requirements of control systems involve tuning of values of variables; they have no impact on the structure of the diagrams or programs, which tend to be stable. The

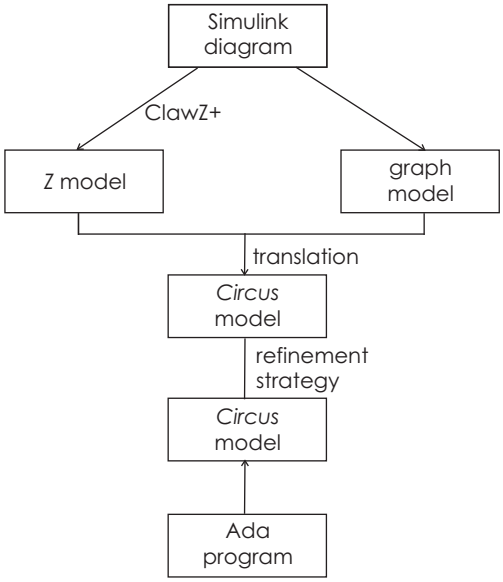


**Fig. 1.** Verification strategy

tactics of refinement and proof are independent of particular values and can be reused directly. Structural changes to diagrams and programs have more of an impact, but since our approach is based on refinement, and so, compositional, the cost of the effort entailed by the change is proportional to its size.

The existing experience with ClawZ improves our confidence in the suitability of the *Circus* semantics. In addition, the availability of tools simplifies the mechanisation of the generation of *Circus* models. We have already implemented a tool that works with ClawZ, and generated models for industrial examples [57] In [14], we presented an initial version of the semantics; here we formalise an improved and extended version. Most importantly, we explain how the semantics can be used to prove programs correctness.

In the next section, we present a brief introduction to Simulink diagrams. In Section 3 we describe ClawZ and *Circus*. Our translation strategy which defines a *Circus* semantics for discrete-time Simulink diagrams is presented in Section 4. Section 5 discusses the *Circus* models for Ada programs. The refinement strategy is presented in Section 6. Finally, in Section 7 we briefly address related work, and in Section 8, we summarise our results, and discuss future work. Appendix A formalises a graph model of diagrams. Appendix B gives *Circus* refinement laws of general interest used in our technique.

## 2. Control law diagrams

In a control law diagram, systems are modelled by directed graphs of blocks connected by wires. Roughly speaking, wires carry signals, and blocks represent functions that determine how outputs are calculated from the inputs. In a continuous-time model, signals vary continuously; in a discrete model, signals are sampled at fixed time intervals, so that input and output take place in cycles. Blocks can be themselves defined by diagrams, and so large diagrams typically have a hierarchical structure.

A simple example of a Simulink diagram is presented in Figure 2; it specifies a PID (Proportional Integral Derivative) controller. This is a simple feedback mechanism that is, however, in widespread use in real control applications. Its main purpose is the correction of an error in some measured value. Typically, the value of the error is obtained using a sensor, and correction is achieved by outputting information used to regulate an actuator. For example, a temperature controller obtains the amount by which it may be too hot or too cold, and indicates how the source of heat should be regulated. This is calculated as the weighted sum of the correction actions indicated by three different methods: proportional, integral, and derivative. The first method produces a correction proportional to the error; the integral value takes the history of errors into account; and finally the derivative correction value considers the rate of change in the error. The controller reads the error and outputs the correction over and over again at predefined intervals.

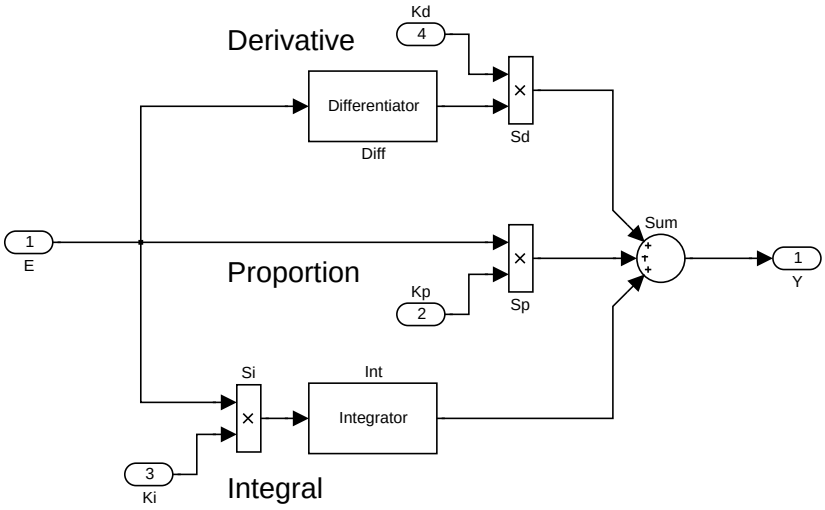


**Fig. 2.** PID (Proportional Integral Derivative) controller

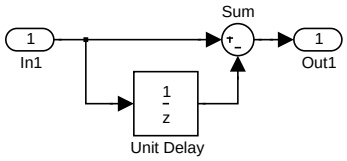


**Fig. 3.** PID Differentiator

In our example, the inputs of the PID controller are the error E, and the weights, Kp, Ki, and Kd, for the proportional, integral, and derivative values. Annotations indicate the branches that calculate the correction according to the Derivative, Proportion, and Integral methods.

Inputs and outputs of the diagram are represented by rounded blocks containing numbers. Each block has a name, and in the case of the inputs and outputs, the blocks are named after them. In our example, we have input blocks E, Kp, Ki, and Kd, and one output block, Y.

Typically, a block takes some input signals and produces some outputs according to a function determined by the kind of block in question. Different block shapes and annotations inside the blocks give a visual indication of their functionality. The circle is a sum block. The block with a $\times$ symbol are product blocks. There are libraries of basic blocks in Simulink, and they can also be user-defined.

In our example, the blocks enclosing names, that is, the blocks named Diff and Int, are subsystems. The names in (the rectangles that represent) the blocks, Differentiator and Integrator, respectively, are just annotations that give an indication of the functionality of the subsystems. They are defined by other diagrams named after the blocks. For example, the diagram Diff is presented in Figure 3.

Blocks can have state. For instance, 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 of a diagram depends on the values of the inputs and of the state in the blocks, if any, 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 are not calculated, and can either be held or reset to an initial value. Any state in blocks within the subsystem is held until the subsystem is about to be executed again, at which point the state can be modified, held, or reset to an initial value.

Merge blocks take a number of inputs and produce one output: the most recently calculated input. Typically, the inputs are connected to conditionally executed subsystems, and in each cycle only one of them produces a calculated output. This is the output produced by the merge block. If none of the inputs are calculated in a cycle, then the merge block repeats its previous output.

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.

## 3. ClawZ and *Circus*

This section describes the notation (*Circus*) and tool (ClawZ) used in our work. In particular, we explain how ClawZ defines a semantics for diagrams; it is a partial semantics, in that it covers only block functionality.

### 3.1. ClawZ

As already mentioned, ClawZ is a tool suite for verification of Ada programs against Simulink diagrams. It provides a self-contained formal account of the functionality of the blocks of a diagram, and of the diagram itself using Z. Verification is based on refinement, with conjectures built in terms of the Z specifications.

The essence of the semantics of a diagram is defined by three elements: the functionality embedded in the Simulink block library, the way in which library blocks are used and connected by wires in the diagram, and the time model. Timing features are not covered in ClawZ and in our work: we assume a discrete-time model, so that the system specified has a cyclic behaviour, in which at each cycle it reads some inputs and produces some outputs. This is the model used in software implementations of control systems.

In specifying the semantics of a diagrammatic notation, the first concern is perhaps the definition of a linear abstract representation. This is already provided by Simulink, which represents diagrams as a list of block specifications in a structured ASCII file, the mdl file. Each specification refers to a basic block in the Simulink library or to a user-defined block specified by another diagram, and records information like inputs and outputs, initial value of the state, if any, and so on. The reference to the block library is an implicit specification of functionality, and the information about inputs and outputs defines a graph structure.

The formal semantics of the basic blocks and, more importantly, of subsystem blocks defined by diagrams (involving basic blocks and, possibly, further subsystem blocks) is defined by ClawZ as Z schemas. It describes how the outputs of the blocks are calculated from its inputs, and perhaps some state information. This gives a more complete view of the diagram functionality than its (linear) Simulink representation. For this reason, it is a convenient starting point to construct a *Circus* model of the diagram.

Formally, ClawZ implements a function which, given a diagram, and the name of one of its blocks, gives a formal characterisation of that block as a set of bindings (records). Typically, this set is defined by a schema. In other words, the Z specification provided by ClawZ does not follow the traditional style in which schemas are used to define the state of the system, and its operations. Instead, the specification uses schemas as one of the fundamental type constructors of Z; blocks (and diagrams) are record types defined by schemas in Z.

An implicit parameter of the semantic function defined by ClawZ, which we call ClawZ itself, is the formalisation of the Simulink block library. It provides a record type definition for each block. For example, the block Sum in Figure 3 is an instance of a sum block that negates its second input; in effect, this is a subtraction. Its model in the ClawZ library is the schema below.

z

$$
\begin{array}{|l}
\hline
Sum\_PM \\
\hline
In1?,\ In2? : \mathbb{R} \\
Out1! : \mathbb{R} \\
\hline
Out1! = In1? -_R In2? \\
\hline
\end{array}
$$

The notation adopted here is the Z dialect of ProofPower, which is very close to the Z standard; we point out the few differences as needed. The Z that precedes the schema above is used by ProofPower to distinguish Z paragraphs from definitions in HOL or SML. The components of the schema are components (fields) of the records in the set that characterises the block. For the inputs, we have components *In1*?, *In2*? and so on, depending on the number of inputs of the block. Similarly, for the outputs, we have components *Out1*!, *Out2*!, and so on. The block in our example has two inputs and one output.

A theory of real numbers for Z is available in ProofPower; above, we declare the components of the schema

to be of type real. The predicate uses a difference operator ($-_R$) for real numbers to define the output. The set defined by *Sum_PM* contains all the bindings with components *In1?*, *In2?*, and *Out1!* whose values are of type real, and are related as described in the predicate.

Some blocks require a parameter upon instantiation. For example, a unit delay takes the initial value of its state. In this case, it is formalised as a (possibly generic) function using an axiomatic description. The generic parameter is the type $X$ of the state, input, and output. The function takes a binding with a single component *X0* and yields a set of bindings that characterises the unit delay block. The type of the bindings in this set is defined by an unnamed horizontal schema $[In1?, initial\_state, state, state', Out1! : X]$ with an empty predicate part. The value of *X0* is used to initialise the state.

z

$$
\begin{array}{|l}
\hline
[X] \\
UnitDelay\_g : [X0 : X] \ \rightarrow \mathbb{P}\ [In1?, initial\_state, state, state', Out1! : X] \\
\hline
\forall\ pars : [X0 : X] \bullet \\
\quad UnitDelay\_g\ pars = [In1?, initial\_state, state, state', Out1! : X \mid \\
\qquad\qquad initial\_state = pars.X0 \ \land\ Out1! = state \ \land\ state' = In1?] \\
\hline
\end{array}
$$

The function is generic because unit delay blocks work on several types of signals: real numbers, vectors, and so on. The value *X0* of the argument record is used to initialise the *intial_state* components of the bindings in the resulting set. In general, the bindings in the set that characterises a block with a state includes, besides input and output components, the three extra components *initial_state*, *state*, and *state'*. They record the value of the state when the system starts its execution, the value of the state at the beginning of the current cycle, and the value of the state at the end of the cycle, respectively.

In practical terms, given a diagram, ClawZ produces a Z specification that characterises each of its blocks, as well as the whole diagram. As an example, part of the output of ClawZ for the PID diagram in Figure 2 is presented in Figure 4. The name of a block in the Z specification includes, besides that explicitly indicated in the diagram, the name of the subdiagram in which the block occurs, and the name of the diagram itself. For instance, the Sum block in the subdiagram Diff (Figure 3) is defined by the schema *pid__Diff__Sum*.

This schema, as well as those for the Sd and Sum blocks in the top diagram, that is, *pid__Sd* and *pid__Sum*, are specified directly in terms of library definitions: *Sum_PM* presented above and others. For the *UnitDelay* block, as discussed above, the definition in the Z library is a function; in a particular model it is applied to an appropriate argument to define a set of bindings. In our case, the argument is a binding $X0 \mathrel{\widehat=} 0\ e\ 0$ with a single component *X0* whose value is the real number 0, written *0 e 0* in ProofPower.

The schema *pid* defines the top diagram; it declares the inputs and outputs of the system, and an extra component for each of the blocks at this level, that is, Diff, Int, Sd, Si, Sp, and Sum. The names of the input and output components are still generic, that is, *In1?*, *In2?*, and so on, and *Out1!*. This ensures that the model of a top diagram is similar to that of a subsystem diagram or even of a block; such uniformity is beneficial for reasoning. The types of the block components are the sets of bindings that specify them. The predicate of *pid*, which is omitted for the sake of conciseness, 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.

The definition of Diff is similar to that of the top diagram in *pid*. It is a schema that declares the inputs and outputs, and each of the blocks in the diagram Diff. The predicate, which is similar to that of *pid*, equates, for instance, the inputs of the Sum block to the input of the diagram and the output of the Unit Delay block. It also defines that the output of the diagram is that of Sum. The repeated equality $UnitDelay.In1? = Sum.In1? = In1?$ is not part of the Z standard notation, but is accepted in ProofPower; it is a shorthand for the conjunction of $UnitDelay.In1? = Sum.In1?$ and $Sum.In1? = In1?$. Similarly, the predicate of the *pid* schema is a conjunction of equations that reflect the wiring in Figure 2.

In summary, the inputs of a diagram or of a block are modelled as components *In1?*, *In2?*, and so on; similarly, outputs have conventional names *Out1!*, *Out2!*, and so on. If the block has a state, there are components *state*, *state'*, and *initial_state* to record its value at the beginning and at the end of the cycle, and at the beginning of the first cycle. The other components, if any, represent blocks; for each block in the top diagram or in a subsystem diagram, there is a component. The predicate is a conjunction of equalities that specify how the inputs and outputs are connected.

z

$$pid\_\_Diff\_\_Sum \mathrel{\widehat=} Sum\_PM$$

z

$$pid\_\_Diff\_\_UnitDelay \mathrel{\widehat=} UnitDelay\_g\ (X0 \mathrel{\widehat=} 0\ e\ 0)$$

z

*pid__Diff*

$In1? : \mathbb{U};$
$Sum : pid\_\_Diff\_\_Sum;$
$UnitDelay : pid\_\_Diff\_\_UnitDelay;$
$Out1! : \mathbb{U}$

---

$Out1! = Sum.Out1!;$
$UnitDelay.In1? = Sum.In1? = In1?;$
$Sum.In2? = UnitDelay.Out1!$

...

z

$$pid\_\_Sd \mathrel{\widehat=} Product\_M2$$

...

z

$$pid\_\_Sum \mathrel{\widehat=} Sum\_P3$$

z

*pid*

$In1? : \mathbb{U};\ In2? : \mathbb{U};\ In3? : \mathbb{U};\ In4? : \mathbb{U};$
$Diff : pid\_\_Diff;$
$Int : pid\_\_Int;$
$Sd : pid\_\_Sd;$
$Si : pid\_\_Si;$
$Sp : pid\_\_Sp;$
$Sum : pid\_\_Sum;$
$Out1! : \mathbb{U}$

---

...

**Fig. 4.** ClawZ output for the PID (ProofPower notation)

The ClawZ model of a diagram specifies the functionality of all of its blocks, over one cycle of execution. It does not, however, capture the graph structure of the diagram, and so does not have an explicit record of opportunities for parallelisation. This is addressed by the *Circus* model proposed here.

### 3.2. *Circus*

This is a language for refinement; *Circus* includes specification constructs from Z and Morgan's refinement calculus [41], CSP constructs to model communication and concurrency, and guarded commands, including assignments and conditionals. It is distinctive in that it mixes (Z) data operations and (CSP) constructs for

**channel** *disp*

**channel** $set, add, mult, out : \mathbb{N}$

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

**state** $Register == [\, r : \mathbb{N} \,]$

*Prod*

$\Delta Register$
$x? : \mathbb{N}$

---

$r' = r * x?$

$\bullet\ (\mu\, X \bullet (set?x \rightarrow r := x \Box add?x \rightarrow r := r + x \Box mult?x \rightarrow Prod \Box disp \rightarrow out!r \rightarrow Skip)\,;\, X)$

**end**

**channel** $calc : \mathbb{N}$

**process** $Fact \mathrel{\widehat=}$ **begin**
$FCalc \mathrel{\widehat=} (n : \mathbb{N} \bullet \textbf{if}\ n = 0 \rightarrow Skip \,[\!]\, n \neq 0 \rightarrow mult?n \rightarrow FCalc(n-1)\ \textbf{fi})$
$\bullet\ (\mu\, X \bullet calc?n \rightarrow set!1 \rightarrow FCalc(n)\,;\, disp \rightarrow X)$
**end**

**process** $System \mathrel{\widehat=} (Mem \,[\![\, \{\!|\, set, mult, disp, out \,|\!\}\ |\ \{\!|\, calc, set, mult, disp \,|\!\}\, ]\!]\, Fact) \setminus \{\!|\, set, mult, disp \,|\!\}$

**Fig. 5.** Example of a simple *Circus* specification

communication and parallelism in a flexible way. Events are not attached to state changes: when an event happens, there is no implicitly associated state change. State changes have to be explicitly specified, just like they are in programming languages. (This approach is in contrast with that adopted in other combinations of CSP with a state-based notation [55, 22]). Moreover, refinement can be carried out compositionally.

Like in Z, a *Circus* program is a sequence of paragraphs, but they also include channel and process declarations. Figure 5 gives an example: a factorial calculator that uses a memory register.

Communications are events, just like in CSP. In our example, we first of all declare a few channels. The channel *disp* does not have a type, and so it is used just for synchronisation: to request the memory register to output its value through the channel *out* of type $\mathbb{N}$. The channels *set*, *add*, and *mult* also have type $\mathbb{N}$; they are used to update the memory using the communicated value.

A process encapsulates state and exhibits behaviour. An explicit definition of a process is a sequence of paragraphs; the specifications of *Mem* and *Fact* in Figure 5 are examples. A distinguished paragraph introduces the state schema in the style of Z; in the case of *Mem*, this is the *Register* schema with the single component $r$ of type $\mathbb{N}$, but *Fact* is stateless. Encapsulation means that the state is local; interaction with the process is only via communications through channels.

At the end of an explicit definition, a main action specifies the behaviour of the process. Actions are defined using a combination of Z (state) operations, CSP constructs, and guarded commands.

In *Mem*, the main action is recursive; it repeatedly offers the choice of interaction over any of the channels *set*, *add*, *mult*, and *disp*. Communication over *set* takes an input value $x$, which is assigned to $r$; the input $set?x$ declares $x$ as a local variable whose scope is the assignment or, in more general terms, the action prefixed by the input communication. Similarly, the input prefixing $add?x \rightarrow r := r + x$ declares the local variable $x$ for use in the assignment $r := r + x$. The value communicated over *add* is assigned to $x$ and used to increment the register. For the sake of example, we specify the state update that corresponds to an input over *mult* using a Z schema *Prod*, instead of simply using $r := r * x$. The style of definition of Z state operations is standard, and the input variable $x?$ of *Prod* is linked to the local variable $x$ declared by the input communication. Finally, we observe that interaction on *disp* does not lead to any state operation; instead, it is a request for the output of the value of $r$ through the channel *out*.

In the case of *Fact*, the main action is also recursive: it repeatedly accepts a request to calculate the factorial of a natural number $n$, after which it sets the memory, uses it to calculate the factorial, and requests that the output is displayed. The extra channel *calc* is declared just before the definition of *Fact*.

Typically, a process definition includes several paragraphs to specify actions that are combined in the main action to define the behaviour of the process. In our simple examples, we have, the action *Prod* in the process *Mem*, and the action *FCalc* in the process *Fact*. The latter is a parametrised action with parameter $n$ of type $\mathbb{N}$; it uses the initialised register to calculate the factorial of $n$. A conditional determines if it should terminate immediately, if $n = 0$, or multiply the value of the register by $n$ before recursing, if $n > 0$. The basic action *Skip* terminates immediately without changing the state.

Like actions, processes can also be combined using CSP operators: sequence, choice, parallelism, hiding, and others. Parallelism is alphabetised just like in CSP: we can either define a synchronisation set or the alphabet of the parallel processes or actions. A synchronisation set contains the channels on which the parallel processes (or actions) need to synchronise; communications on all other channels occur independently. If, on the other hand, we use the alphabetised parallel operator, for each parallel process or action, we define an alphabet; in this case, the process (or action) can only communicate on a channel $c$ if it is in its alphabet, and needs to synchronise with all other processes or actions that also have $c$ in their alphabet.

In our example, we define the process *System* as the parallel composition of *Mem* and *Fact*. We use the interface parallel operator, and define the alphabets of *Mem* and *Fact*. Since we leave *add* out of the alphabet of *Mem*, it cannot communicate over this channel, although such communications may be helpful in other uses of *Mem*. Synchronisation is required for the channels in the intersection of the alphabets; in our example, they are *set*, *mult*, and *disp*. These are internal channels used only for communications between the components of the system; therefore, they are hidden in the definition of *System*. In summary, our system takes inputs over *calc*, and produces outputs using *out*; all other channels are hidden, and communications over them are not visible to the environment of *System*.

In the case of a parallelism of actions, there is a concern about conflicting access to state components (and local variables). For that reason, the parallel operators for actions define partitions of the variables in scope. For example, the composition of actions $A_1$ and $A_2$ using the alphabetised parallel operator 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 variables in scope. Both $A_1$ and $A_2$ have access to the initial value of all variables; however, $A_1$ can only modify those named in $ns_1$, and $A_2$ can only modify those in $ns_2$. Figure 5 presents a parallelism between processes, but not between actions. An example of action parallelism is provided in the next section (Figure 10).

A refinement calculus and strategy is available for *Circus* [15]. The strategy aims at calculating concurrent implementations from centralised specifications. In this paper, we provide a few novel refinement laws, which are clearly marked in Appendix B, and a strategy tailored to the verification of Ada implementations with respect to models of diagrams. In this case, we aim at removing the massive parallelism in the models.

## 4. Translation strategy

We formalise the *Circus* model of a diagram as a function $[\![ d ]\!]^C$ that takes the linear representation of a diagram $d$ and provides a *Circus* specification. In this section, we present the definition of this function; the meta-notation that we use to describe the *Circus* specification is based on the Z and *Circus* mathematical and action notations. When there is the possibility of ambiguity, to differentiate the occurrences of symbols of the meta-notation from those of the target *Circus* specification, we use a sans-serif font for the meta-notation.

There are two intermediary models that we extract from $d$ in order to define the *Circus* model. The first is the Z model defined by ClawZ; formally this is the result of applying the $\mathsf{ClawZ}$ function described in the previous section to $d$. In fact, we consider a few extensions to ClawZ to cater for a larger number of blocks. They, however, do not interfere with the structure of the model already described.

The second model of the diagram captures its structure as a graph. It is described in Section 4.1 below, and formalised in Appendix A. Section 4.2 describes the channels used in the *Circus* specification. Modelling of blocks is the subject of Section 4.3. Finally, in Section 4.4 we explain how the models of the blocks are used to define a *Circus* model for the diagram. As detailed in the sequel, in the *Circus* model of a diagram, blocks, as well as the diagram itself, are defined as processes.

For clarity, we present the definition of $[\![ d ]\!]^C$ in an incremental way, with the various paragraphs of the *Circus* specification interspersed with comments and examples. We start the definition below, where we use a **let** clause to name the results of applying the $\mathsf{ClawZ}$ and $\mathsf{DF}$ functions to $d$.

$$[\![ d ]\!]^C \;=\; \mathbf{let}\ \mathsf{clawz} = \mathsf{ClawZ}(d);\ \mathsf{df} = \mathsf{DF}(d) \bullet$$

As already said, the function $\mathsf{ClawZ}$ is that defined by ClawZ. The function $\mathsf{DF}$ defines a graph that captures

$$
\begin{array}{l}
\langle\!| \mathit{spec} == \mathit{PID}, \\
\quad \mathit{inputs} == \{ E, Kp, Ki, Kd \}, \\
\quad \mathit{outputs} == \{ Y \}, \\
\quad \mathit{blocks} == \{ Si \mapsto \langle\!| \mathit{inps} == \langle E, Ki \rangle, \mathit{outs} == \langle Si\_out \rangle, \\
\qquad\qquad \mathit{flows} == \{ \{ Si\_out \} \mapsto \langle\!| \mathit{enabled} == \mathit{always}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{ordered} == \mathit{false}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{rinps} == \{ E, Ki \} |\!\rangle \} |\!\rangle, \\
\qquad Diff \mapsto \langle\!| \mathit{inps} == \langle E \rangle, \mathit{outs} == \langle Diff\_out \rangle, \\
\qquad\qquad \mathit{flows} == \{ \{ Diff\_out \} \mapsto \langle\!| \mathit{enabled} == \mathit{always}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{ordered} == \mathit{false}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{rinps} == \{ E \} |\!\rangle \} |\!\rangle, \\
\qquad \mathit{Sum} \mapsto \langle\!| \mathit{inps} == \langle Sd\_out, Sp\_out, Int\_out \rangle, \mathit{outs} == \langle Y \rangle, \\
\qquad\qquad \mathit{flows} == \{ \{ Y \} \mapsto \langle\!| \mathit{enabled} == \mathit{always}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{ordered} == \mathit{false}, \\
\qquad\qquad\qquad\qquad\qquad \mathit{rinps} == \{ Sd\_out, Sp\_out, Int\_out \} |\!\rangle \} |\!\rangle, \\
\quad \ldots \} |\!\rangle
\end{array}
$$

**Fig. 6.** Graph model for the PID

the data flow in $d$. It is specified in the next section.

### 4.1. Graph model

To provide an accurate *Circus* model of a diagram, we use a graph model that captures its data flow. It is formally specified in Appendix A; here we illustrate the graph structure by means of examples.

The function DF associates a diagram to a record (binding) that registers the diagram name (in a field *spec*), the names of its *inputs* and *outputs*, and a mapping *blocks* that associates each of its blocks, identified by their names, to information about its wiring. The type *Graph* defined in Appendix A defines the set of such records. The range of the mapping *blocks* is specified using the type *BlockWiring*. Part of the record for the PID diagram, that is, DF($d$), where $d$ is the Simulink textual representation of the PID, is in Figure 6. In this case, the name of the diagram is *PID*, the inputs are $E$, $Kp$, $Ki$, and $Kd$, and the output is $Y$. Each block is associated to its wiring information; in Figure 6, we present the wiring for *Si*, *Diff*, and *Sum*.

The inputs and outputs of the diagram are named after its input and output blocks. The internal wires are named after the block that produces it as an output, using suffix *_out*, if there is only one output, or *_out*1, *_out*2 and so on, if there is more than one output. For clarity of the model, however, when the output of a block is connected to an output port, we name the channel after the output. In our example, the output of the Sum block, for instance, is named $Y$, after the output of the diagram, rather than *Sum_out*.

The wiring of a block is defined by a binding that records its inputs (*inps*), outputs (*outs*), and the dependencies between them, that is, the *flows* of execution. To explain the need to model flows of execution, we first consider the diagram in Figure 7 (a). It has two inputs I1 and I2, and three outputs O1, O2, and O3. The subsystem block SS is defined by the diagram in Figure 7 (b). If we considered only the diagram in Figure 7 (a), we could say that SS takes two inputs and produces two outputs. Inspection of Figure 7 (b), however, reveals that O2 can be provided only once both inputs are available, but O1 can be determined from just I1. So, a model that defines that SS can output O1 only once both I1 and I2 are input is too restrictive. The graph model, therefore, needs to record that SS has two (independent) flows of execution: one that calculates O1 and another that calculates O2. For O2, both inputs are required, but not for O1.

In principle, each output determines a potentially independent flow of execution that calculates it, but a group of outputs may all be part of a single flow. Typically, the calculations involved in the definition of the value of each output are different, but they may, for example, depend on exactly the same inputs. Therefore, the flows are recorded as a function from sets of outputs to a binding (of type *Flow* as defined in Appendix A) that records information about the flow of execution that determines these outputs.

Relevant information about a flow determines its required inputs (*rinps*). For the block SS in Figure 7 (a), for instance, the required inputs for the flow $\{ O2 \}$ is $\{ I1, I2 \}$, but for $\{ O1 \}$, it is $\{ I1 \}$.

To cater for action and enabled subsystems, we also need to record whether or not a flow of execution is always enabled. As an example, we consider the diagram in Figure 8. A basic If block takes the input In1; if

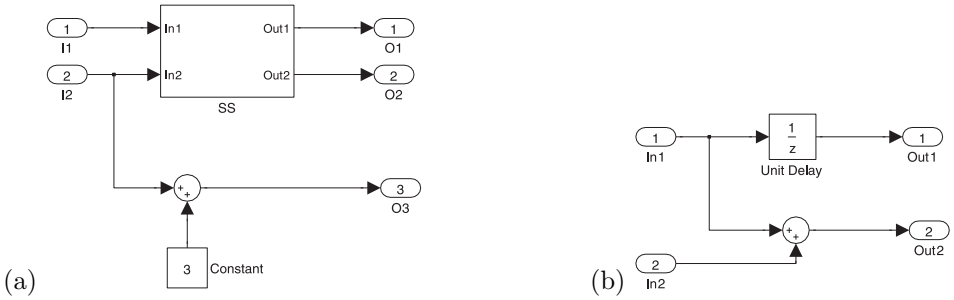


**Fig. 7.** Independent flows of execution

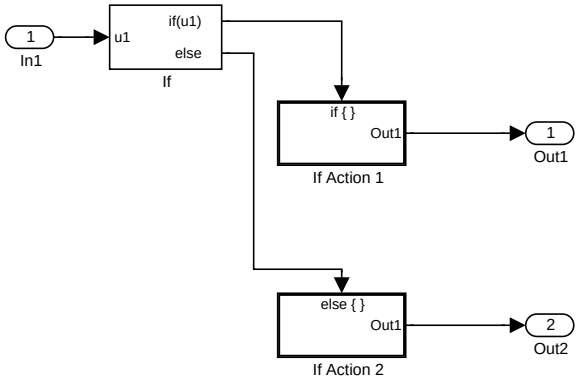


**Fig. 8.** If Action Subsystems

it is greater than 0, then the first output is *true*, which is represented by 1 in Simulink, otherwise the second output is 1. These outputs are connected to the action ports of two action subsystems.

The output of an action subsystem depends on whether the value provided in its action port is 1 or not, that is, on whether the subsystem is enabled or not. The graph model for such a subsystem block, therefore, needs to record, for each of the flows defined by its outputs, the name of the action port. For If Action 1, we have a single flow $\{Out1\}$, and its enabling port is just $If\_out1$. (As explained in Appendix A, formally, this is recorded in the field *enabled* of the record of type *Flow* that models the flow as $esigs(\{If\_out1\})$.)

Finally, we need to record whether the output of a flow of execution depends on the order of its required inputs. This is necessary to cater for merge blocks, which take a number of inputs and output the latest calculated one. The characterisation of a merge block with two inputs is as follows.

$$\begin{array}{l} \langle\!| \, inps == \langle\, In1, In2 \,\rangle, \\ \quad outs == \langle\, Out1 \,\rangle, \\ \quad flows == \{\, \{\, Out1 \,\} \mapsto \langle\!| \, enabled == always, ordered == true, rinps == \{\, In1, In2 \,\} \,|\!\rangle \,\} \,|\!\rangle \end{array}$$

Intuitively, a merge block combines its inputs into a single output whose value is equal to the most recently computed, that is, updated, input. Even inputs that are not updated need to be provided (communicated), before the output is available. So, above the value of *rinps* for the single flow $\{\, Out1 \,\}$ includes both inputs.

In our example, as indicated in Figure 6, the blocks are very simple: they have one flow, which is always enabled, and whose output does not depend on the input order. Blocks like Diff represent a diagram, but from the point of view of the PID, it is just a block; its internal communications are abstracted away.

In the previous examples involving subsystem blocks, the information about their flows can be extracted by an analysis of the structure of the diagrams that define them. Even basic blocks, however, can have interesting flows of execution. For example, the unit delay block can produce outputs before it receives (all) the inputs. To construct the graph model of a diagram, we, therefore, need a library that records information about the basic blocks that compose diagrams, just like in ClawZ.

We consider, for instance, the diagram in Figure 9, which defines the Int block of the PID diagram (see

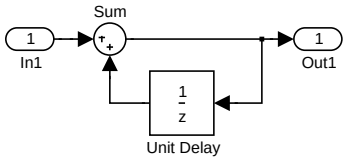


**Fig. 9.** PID Integrator

Figure 2). In constructing the model of this diagram, we need the information that the output of Unit Delay is available before its input is received. The input is the output of a Sum block that takes the output of Unit Delay itself as input. A model for the diagram that requires all inputs of all blocks to be provided before their outputs are produced would, therefore, incorrectly allow for a deadlock.

Since Unit Delay is a basic block, however, to determine the immediate availability of its output, we need to resort to recorded information about such blocks. Its characterisation is as follows; the input is named $In1$, and the output $Out1$. As in ClawZ, this is just a convention; when blocks in diagrams are considered, the proper names of the inputs and outputs have to be determined in accordance with the wiring.

$$\begin{array}{l} \langle\!| \mathit{inps} == \langle\, \mathit{In1} \,\rangle, \\ \quad \mathit{outs} == \langle\, \mathit{Out1} \,\rangle, \\ \quad \mathit{flows} == \{\, \{\, \mathit{Out1} \,\} \mapsto \langle\!| \mathit{enabled} == \mathit{always}, \mathit{ordered} == \mathit{false}, \mathit{rinps} == \varnothing \,|\!\rangle \,\} \,|\!\rangle \end{array}$$

Its only input is not required by its only flow $\{\, \mathit{Out1} \,\}$, so the value of its *rinps* field is the empty set.

It is the graph model of a diagram that identifies, for instance, the channels declared and used in its *Circus* model. This is described in detail in the next section.

## 4.2. Channels

The *Circus* specification of a diagram first declares all signals as channels; for that, the information in the fields inputs, outputs, and each of the fields outs in the blocks fields of df is used.

**channel** df.inputs, df.outputs, $\{$ B : Block $\bullet$ df.blocks(B).outs $\}$ : $\mathbb{U}$

Even though df.inputs is a set of signals, we use it above to denote a list of the signals in this set; the same comment applies to df.outputs and to the set of sequences df.blocks(B).outs of signals: one for each block B of the diagram. The type Block contains the valid block names; Appendix A gives the formalisation of the df model. All these signals are declared as channels of type $\mathbb{U}$.

We also declare a synchronisation channel *end_cycle*; after taking all its inputs and producing all its outputs, each process representing a block of a diagram waits to synchronise on *end_cycle* before proceeding to the next cycle. In this way, the behaviour of all block processes are kept in phase.

**channel** *end_cycle*

In this paper, we only consider single-rate diagrams; for multi-rate diagrams, we will explore the timed version of *Circus* named *Circus Time* [49].

The *Circus* specification corresponding to the PID, for example, starts as follows.

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

Next, the *Circus* specification includes the ClawZ library, which is used in clawz. There is then a process for each block, and at the end, the definition of the diagram; the are defined in the following sections.

## 4.3. The blocks

The model of a block is a single centralised process defined explicitly, independently of whether the block is simple, like Sd, or a subsystem, like Diff. This process lifts the clawz model, which is based on type definitions,

to *Circus* actions. For each block B in dom df.blocks, we define a *Circus* process also called B.

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

We consider a block whose flows are always enabled and do not depend on the order of the inputs.

The state of B includes a component for each component named *state* used in the definition of B in clawz.

**state** B_*State*

clawz(B)__$h$(def1)_*state*, ..., clawz(B)__h(defn)_*state* : $\mathbb{U}$

To determine the names defi used in the definition of the *state* components of B_*State*, we consider the signature of the ClawZ definition clawz(B). As already said, this is a set of bindings. Its signature, therefore, is the power set of a schema type, or more plainly, of a record type defined by listing the record fields and respective (maximal) types. For example, the schema *pid__Diff* (in Figure 4) characterises the PID Diff block. Its signature is the powerset of the schema type defined below.

$$[\, In1?, Out1! : \mathbb{A};\ Sum : [\, In1, In2?, Out1! : \mathbb{A}\,];\ UnitDelay : [\, In1?, Out1!, initialstate, state, state' : \mathbb{A}\,]\,]$$

The type $\mathbb{A}$ is the given set of values used in number systems. Because the above signature has a component *UnitDelay* whose type is a schema with a component called *state* (as defined by *UnitDelay_g* in the ClawZ library) the process *Diff* (see Figure 10) has a state component *pid__Diff__UnitDelay_state*.

We specify a function *stateN*, which, given a schema type $S$, defines the set of sequences of component names that can be used to select a (sub)component of $S$ whose type is itself a schema with a component named *state*. For the schema type above, *stateN* identifies the set containing the sequence $\langle$ *UnitDelay*$\rangle$.

We define *stateN*($S$) in terms of a function *stateT*($s$, $T$), which applies to Z (maximal) types $T$, rather than just schema types $S$. The Z types include given sets, power sets, cartesian products, and schemas. The first parameter $s$ of *stateT* is a sequence of component names. Formally, *state*($S$) is defined as *stateT*($\langle\,\rangle$, $S$), and intuitively, $s$ is the sequence of names that can be used to select a component of $S$ that has type $T$.

We provide an inductive definition for *stateT*($s$, $T$) based on the structure of types $T$ in Z.

**Definition 4.1.**

$$stateT(s, TN) = stateT(s, \mathbb{P}\, T) = stateT(s, T_1 \times T_2) = \varnothing$$
$$stateT(s, [\, i \bullet c_i : T_i\,]) = \{\, s\,\}, \quad \text{if } \exists\, i \bullet c_i = state$$
$$stateT(s, [\, i \bullet c_i : T_i\,]) = \bigcup i \bullet stateT(s \frown \langle c_i\rangle, T_i), \quad \text{if } \neg\ \exists\, i \bullet c_i = state$$

We use $TN$ to stand for a type name, that of a given set, and $T$, $T_1$, $T_2$, and $T_i$ to stand for arbitrary type descriptions. For given sets, power sets, and cartesian products, *stateT* gives the empty set of selector sequences: given sets have no components, and, in a model of a block, we do not have other blocks arranged in a power set or a cartesian product. What we do have is blocks directly inside other blocks. In our example, for instance, *pid__Diff* is the model of a block that includes as components models of other blocks. In the case of the component *Sum*, it is a block without state, but in the case of *UnitDelay*, we do have a state. Correspondingly, in the definition of *stateT*, we consider schema types $[\, i \bullet c_i : T_i]$ which include components $c_i$ of type $T_i$. If any of the names $c_i$ is *state*, then the sequence $s$ is identified as a selector for a schema type with a *state* component. Otherwise, we consider each of the components $c_i$ individually. For each of them, we consider the result of recursively applying *stateT* to the sequence obtained by appending $c_i$ to $s$, and to the type $T_i$ of $c_i$. The result is the distributed union of the sets of selector sequences so obtained.

The sequences of names in *stateN*(clawz(B)) are exactly the sequences defi used above to construct the names of the state components of the process B as specified above. The name $h$(defi) used in the declaration of the state schema B_*State* is the __-separated list of the names in defi. The simple definition of the syntactic function $h$ is omitted. For our example, the name of clawz(B) is *pid__Diff* and, since the result of applying *state* to its signature is just the singleton sequence $\langle$ *UnitDelay*$\rangle$, the name of the state component is *pid__Diff__UnitDelay*. We observe that such names always identify a definition of the model of a block.

After the state declaration, we include clawz(B). In our example, the schema *pid__Diff*, as well as the schemas *pid__Diff__Sum* and *pid__Diff__UnitDelay* used in the specification of *pid__Diff*, are included. They were originally presented in Figure 4, as part of the ClawZ output.

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

**state** $Diff\_State == [\, pid\_\_Diff\_\_UnitDelay\_state : \mathbb{U} \,]$

$pid\_\_Diff\_\_Sum == Sum\_PM$

$pid\_\_Diff\_\_UnitDelay \mathrel{\widehat=} UnitDelay\_g(X0 \mathrel{\widehat=} 0\ e\ 0)$

*pid__Diff*

$In1? : \mathbb{U}$
$Sum : pid\_\_Diff\_\_Sum$
$UnitDelay : pid\_\_Diff\_\_UnitDelay$
$Out1! : \mathbb{U}$

---

$Out1! = Sum.Out1!$
$UnitDelay.In1? = Sum.In1?$
$Sum.In1? = In1?$
$Sum.In2? = UnitDelay.Out1!$

*Init*

$Diff\_State'$

---

$\exists\, b : pid\_\_Diff\_\_UnitDelay \bullet pid\_\_Diff\_\_UnitDelay\_state' = b.initial\_state$

*Calculate_Diff*

$\Delta Diff\_State$
$In1?, Out1! : \mathbb{U}$

---

$\exists\, b : pid\_\_Diff \bullet$
$\quad b.In1? = In1? \land b.UnitDelay.state = pid\_\_Diff\_\_UnitDelay\_state \land$
$\quad b.UnitDelay.state' = pid\_\_Diff\_\_UnitDelay\_state' \land b.Out1! = Out1!$

$Calculate\_Diff\_out == Calculate\_Diff \setminus (pid\_\_Diff\_\_UnitDelay\_state') \land \Xi Diff\_State$

$$Execute\_Diff\_out \mathrel{\widehat=} \left( \begin{array}{l} \textbf{var}\ In1 : \mathbb{U} \bullet \\ \quad \left( \begin{array}{l} E?x \rightarrow In1 := x; \\ \textbf{var}\ Out1 : \mathbb{U} \bullet Calculate\_Diff\_out;\ Diff\_out!Out1 \rightarrow Skip \end{array} \right) \end{array} \right)$$

$Flows \mathrel{\widehat=} Execute\_Diff\_out$

$Calculate\_Diff\_State \mathrel{\widehat=} Calculate\_Diff \setminus (Out1!)$

$StateUpdate \mathrel{\widehat=} \textbf{var}\ In1 : \mathbb{U} \bullet E?x \rightarrow In1 := x;\ Calculate\_Diff\_State$

$\bullet\ Init;$
$\quad \mu\, X \bullet (Flows \llbracket \{\,\} \mid \{\!|\ E\ |\!\} \mid \{pid\_\_Diff\_\_UnitDelay\_state\} \rrbracket StateUpdate);\ end\_cycle \rightarrow X$

**end**

**Fig. 10.** *Circus* process for the block Diff

The initialisation of the state is based on the clawz(B) specification.

*Init*

$B\_State'$

---

$\exists\, b : clawz(B)\_\_h(defi) \bullet clawz(B)\_\_h(defi)\_state' = b.initial\_state$

A state component $clawz(B)\_\_h(defi)\_state$, corresponding to the component named *state* of the bindings in

the set $\mathsf{clawz(B)\_\_}h(\mathsf{defi})$, is initialised with the value of the component *initial_state* of the bindings in this set. In *Init*, we identify a binding $b$ of type $\mathsf{clawz(B)\_\_}h(\mathsf{defi})$; its value for *initial_state* defines the initial value of $\mathsf{clawz(B)\_\_defi\_}state$. For instance, if $\mathsf{clawz(B)\_\_}h(\mathsf{defi})$ is a type that models a unit delay block, like in our example, $\mathsf{clawz(B)\_\_}h(\mathsf{defi})$ is a set whose bindings all have the same value for *initial_state*: that specified in the linear representation of the diagram. In Figure 10, the definition of the *Init* schema is a very direct instantiation of its characterisation above; $\mathsf{clawz(B)\_\_}h(\mathsf{defi})$ is $pid\_\_Diff\_\_UnitDelay$.

The definition clawz(B) specifies the state changes resulting from the execution of B as well as its outputs, but clawz(B) is not an operation over the state B_*State*: it is a type. We define a schema *Calculate*_B that lifts clawz(B) to a data operation on B_*State*. It includes the input and output variables *In*1?, *In*2?, *Out*1!, *Out*2!, and so on, of clawz(B), following the standard Z style of specifying data operations. 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 *.state* components of $b$. The new value of the state and the outputs are defined by $b$.

*Calculate*_B

$\Delta$B_*State*

Ini?, Outj! : $\mathbb{U}$

---

$\exists\, b : \mathsf{clawz(B)} \bullet$
$b.\mathsf{Ini?} = \mathsf{Ini?} \land b.d(\mathsf{defi}).state = \mathsf{clawz(B)\_\_}h(\mathsf{defi})\_state \land$
$b.d(\mathsf{defi}).state' = \mathsf{clawz(B)\_\_}h(\mathsf{defi})\_state' \land b.\mathsf{Outj!} = \mathsf{Outj!}$

If B_*State* has a component $\mathsf{clawz(B)\_\_}h(\mathsf{defi})\_state$, it is because clawz(B) has a component that can be selected using defi with *state* and *state*′ components. This justifies the references above to $b.d(\mathsf{defi}).state$ and $b.d(\mathsf{defi}).state'$. The result of $d(\mathsf{defi})$ is the .-separated list of the names in defi.

If clawz(B) does not involve any such component defi, then the set of bindings specified by *Calculate*_B is actually the same as that specified by clawz(B). In this case, *Calculate*_B has only the Ini? and Outj! components of clawz(B), and its predicate is reduced to $\exists\, b : \mathsf{clawz(B)} \bullet b.\mathsf{Ini?} = \mathsf{Ini?} \land b.\mathsf{Outj!} = \mathsf{Outj!}$. It is simple to prove that, for every binding $c$ of type $[\,\mathsf{Ini?}, \mathsf{Outj!} : \mathbb{U}\,]$, we have that $c \in \mathsf{clawz(B)}$ if, and only if, $c \in$ *Calculate*_B. The argument can proceed as follows.

$c \in$ *Calculate*_B
$\Leftrightarrow \exists\, b : \mathsf{clawz(B)} \bullet b.\mathsf{Ini?} = c.\mathsf{Ini?} \land b.\mathsf{Outj!} = c.\mathsf{Outj!}$ [definition of *Calculate*_B]
$\Leftrightarrow \exists\, b : \mathsf{clawz(B)} \bullet b = c$ [property of bindings]
$\Leftrightarrow c \in \mathsf{clawz(B)}$ [one-point rule]

In our example, the schema *Calculate_Diff* lifts *pid__Diff* to an operation over *Diff_State*. For that, we establish a correspondence between the *state* and *state*′ components of the bindings in the component *UnitDelay* of *pid__Diff* and the state components *pid__Diff__UnitDelay_state* and *pid__Diff__UnitDelay_state*′.

Each flow in a block calculates some of the outputs Outj!. For each flow identified by a set f of signals in the domain of df.blocks(B).flows we define an action *Execute*_$\mathcal{N}_\mathsf{f}$, where $\mathcal{N}_\mathsf{f}$ is a unique name determined by the set f. It can be, for instance, formed by a list of the elements in f; that is a unique name, since the flows of a block produce disjoint outputs. In our example, as shown in Figure 6, the block Diff has a single flow that calculates the value output through the channel *Diff_out*. We, therefore, define an action *Execute_Diff_out*.

An *Execute*_$\mathcal{N}_\mathsf{f}$ action uses a schema *Calculate*_$\mathcal{N}_\mathsf{f}$ that defines the values of the outputs in f. It is specified in terms of *Calculate*_B using the schema calculus: we hide the final value of the state, any inputs that are not required and outputs that are not produced, and conjoin the result with $\Xi$ B_*State* so that the state is not modified. The schema $\Xi$ B_*State* specifies that the values of the state components are preserved.

$$\mathit{Calculate}\_\mathcal{N}_\mathsf{f} \mathrel{\widehat=} (\mathit{Calculate}\_\mathsf{B} \setminus (\alpha\mathsf{B}\_\mathit{State}', \mathsf{nrinps}, \mathsf{npouts})) \land \Xi\mathsf{B}\_\mathit{State}$$

**where** $\mathsf{nrinps} = \{\, \mathsf{Ini?} \mid \mathsf{df.blocks(B).inps(i)} \notin \mathsf{df.blocks(B).flows(f).rinps} \,\}$
$\mathsf{npouts} = \{\, \mathsf{Outj!} \mid \mathsf{df.blocks(B).outs(j)} \notin \mathsf{f} \,\}$

We use $\alpha S$ to denote a list of the components of a schema $S$. The set nrinps contains the names Ini? of the components that represent the inputs of B, as defined by df, that are not required for the flow f. As a slight abuse of notation, we refer to this set in a hiding, where a list of its elements is required. The same comment applies to npouts, which contains the names Outj! of the outputs of B that are not produced by f.

In our example, we have defined the schema *Calculate_Diff_out*, which calculates the value of the output *Diff_out* of the block, but does not change *pid__Diff__UnitDelay_state*. All inputs are required and the single output of the block is produced, so only the state component is hidden.

The action $Execute\_\mathcal{N}_{\mathsf{f}}$ takes the required inputs, and then calculates and produces the outputs of f.

$$Execute\_\mathcal{N}_{\mathsf{f}} \mathrel{\widehat=} \left( \begin{array}{l} \textbf{var } \mathsf{rinps} : \mathbb{U} \bullet \\ \quad \left( \begin{array}{l} |||(\mathsf{inp}, \mathsf{Ini}) : \mathsf{crinps} \bullet (\mathsf{inp}?x \rightarrow \mathsf{Ini} := x, \{ \mathsf{Ini} \}); \\ \quad \left( \begin{array}{l} \textbf{var } \mathsf{pouts} : \mathbb{U} \bullet \\ \quad Calculate\_\mathcal{N}_{\mathsf{f}};\ |||(\mathsf{out}, \mathsf{Outj}) : \mathsf{cpouts} \bullet (\mathsf{out}!\mathsf{Outj} \rightarrow Skip, \{\}) \end{array} \right) \end{array} \right) \end{array} \right)$$

$$\begin{array}{l} \textbf{where } \mathsf{rinps} = \{ \mathsf{Ini} \mid \mathsf{df.blocks(B).inps(i)} \in \mathsf{df.blocks(B).flows(f).rinps} \} \\ \qquad \mathsf{pouts} = \{ \mathsf{Outj} \mid \mathsf{df.blocks(B).outs(j)} \in \mathsf{f} \} \\ \qquad \mathsf{crinps} = \{ (\mathsf{inp}, \mathsf{Ini}) \mid \mathsf{inp} \in \mathsf{df.blocks(B).flows(f).rinps} \wedge \mathsf{df.blocks(B).inps(i)} = \mathsf{inp} \} \\ \qquad \mathsf{cpouts} = \{ (\mathsf{out}, \mathsf{Outj}) \mid \mathsf{out} \in \mathsf{f} \wedge \mathsf{df.blocks(B).outs(j)} = \mathsf{out} \} \end{array}$$

First, $Execute\_\mathcal{N}_{\mathsf{f}}$ declares variables Ini to record the values of the required inputs: those in the set characterised by rinps. Namely, we declare Ini when the i-th input is required by f. Once again we refer to a set, in this case rinps, to denote a list of its elements, in this case in the variable declaration. Similarly, to calculate the outputs, $Execute\_\mathcal{N}_{\mathsf{f}}$ declares the variables in the set pouts; it contains the name Outj whenever the j-th output is produced by f. In *Execute_Diff_out*, there is one input variable $In1$, and one output variable $Out1$.

The inputs of a block can be received in interleaving, that is, in an arbitrary order, through each of the channels inp corresponding to an input required by f. The set crinps contains the pairs (inp, Ini) where inp is a channel that corresponds to a required input of f, and i, used to form the name Ini, is the position of that input of B. In $Execute\_\mathcal{N}_{\mathsf{f}}$, actions $\mathsf{inp}?x \rightarrow \mathsf{Ini} := x$ that take an input $x$ through the channel inp and assign it to the local variable Ini are interleaved. This is formalised as an iterated interleaving over all pairs (inp, Ini) in crinps. The name Ini is used to define the name partition of the interleaved action, as required by the interleaving operator for actions to enforce absence of conflict in the access to state components and local variables (see Section 3.2). We use the pair $(\mathsf{inp}?x \rightarrow \mathsf{Ini} := x, \{ \mathsf{Ini} \})$ to describe that each interleaved action $\mathsf{inp}?x \rightarrow \mathsf{Ini} := x$ is associated with the partition { Ini }.

Similarly, outputs are sent in interleaving through the channels out in f. The value output through such a channel out is that in Outj, where j is the position of the corresponding output in B. The value of Outj is defined by the schema $Calculate\_\mathcal{N}_{\mathsf{f}}$. The pairs (out, Outj) are the elements of the set cpouts. The interleaved actions do not change any state components or local variables, so their name partitions are empty. In our example, there is only one input and one output, so in *Execute_Diff_out* the interleaving is reduced to a single prefixing. The required input is $E$; as the only input, its position is 1, so the corresponding variable in rinps is $In1$. The only output is *Diff_out*, with corresponding variable $Out1$.

After the specification of all the actions $Execute\_\mathcal{N}_{\mathsf{f}}$, an action *Flows* combines them in parallel.

$$Flows \mathrel{\widehat=} \ \| \mathsf{f} : \mathrm{dom}\, \mathsf{df.blocks(B).flows} \bullet (Execute\_\mathcal{N}_{\mathsf{f}}, \mathsf{df.blocks(B).flows(f).rinps}, \{\})$$

The alphabets of each of the parallel actions $Execute\_\mathcal{N}_f$ are the required inputs of f. This means that any inputs that are required by more than one flow are shared by synchronisation. There are no shared outputs. The flows do not change any of the state components, so each of the parallel actions $Execute\_\mathcal{N}_f$ are associated to the empty set { } of variable names in the parallelism. Above, we describe each of a parallel actions as a triple, containing the action, and its associated alphabet and name set. Since in Diff there is only one flow, in Figure 10, the parallelism in the action *Flows* is reduced to the action *Execute_Diff_out*.

The schema *Calculate_B* is also used to define a schema *Calculate_B_State* as specified below; it defines the new value of the state after the execution of the block B.

$$Calculate\_\mathsf{B}\_State \mathrel{\widehat=} Calculate\_\mathsf{B} \setminus (\mathsf{Outj})$$

$$\textbf{where } \mathsf{j} \in 1 \,..\, \#\mathsf{df.blocks(B).outs}$$

In *Calculate_B_State* all output variables Outj of *Calculate_B* are hidden. An example is presented in Figure 10: the action *Calculate_Diff_State*, which is defined in terms of *Calculate_Diff* by hiding $Out1!$.

The action *StateUpdate* that updates the state takes all the inputs in df.blocks(B).inps in interleaving.

Like in $Execute\_\mathcal{N}_f$, appropriate variables are declared to record inputs, but all inputs are required.

$$StateUpdate \mathrel{\widehat=} \left( \begin{array}{l} \mathbf{var}\ \mathsf{Ini} : \mathbb{U} \bullet \\ \quad \left( \begin{array}{l} |||(\mathsf{inp}, \mathsf{Ini}) : \mathsf{cinps} \bullet (\mathsf{inp}?x \rightarrow \mathsf{Ini} := x, \{\, \mathsf{Ini} \,\}); \\ Calculate\_\mathsf{B}\_State; \end{array} \right) \end{array} \right)$$

**where** $\mathsf{cinps} = \{\, (\mathsf{inp}, \mathsf{Ini}) \mid \mathsf{inp} = \mathsf{df.blocks}(\mathsf{B}).\mathsf{inps}(\mathsf{i}) \,\}$

In our example, we declare an action *StateUpdate* which takes the only input through $E$ and executes the action specified by *Calculate_Diff_State* to update the state.

As explained previously, the main action at the end of the process definition specifies its behaviour. For B, it is as shown below. It 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 a 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 variable names; on the other hand, *StateUpdate* is associated with the set $\alpha\mathsf{B}\_State$ including all state components. The synchronisation set rInps contains all the inputs required by at least one flow of B. This is because, when an input is received, it needs to be made available to the flows that require it and to the action that updates the state, and so they all synchronise to receive the shared input.

$$\mathsf{rInps} \mathrel{\widehat=} \bigcup \{\!|\, \mathsf{f} : \mathrm{dom}\, \mathsf{df.blocks}(\mathsf{B}).\mathsf{flows} \bullet \mathsf{df.blocks}(\mathsf{B}).\mathsf{flows}(\mathsf{f}).\mathsf{rinps} \,|\!\}$$

As already observed, not all inputs are necessarily required by a flow. Therefore, if we took the range of df.blocks(B).inps as the synchronisation set, we would be too restrictive. If a block has no state, the recursion in the main action only executes *Flows* followed by the synchronisation on *end_cycle*.

## 4.4. The diagram

As already indicated, our *Circus* model abstracts from specific timing aspects of a Simulink diagram; it ignores, for instance, definitions of sampling periods and step sizes that determine the length of the cycle size. Instead, we use synchronisation (on the channel *end_cycle*) to make sure that the calculations embedded in the blocks are kept in step and, therefore, take the correct inputs and specify the expected outputs. In this context, the time-based block diagram semantics is reduced to that of a data flow chart [40].

Accordingly, to define the semantics of a Simulink diagram, we use basically the CSP standard approach to modelling networks of components [30]. As explained above, each box (block) is modelled as a process, and each line (wire) is modelled as a channel. To give the semantics of the network (diagram), we therefore use the parallel composition of the block processes, with the synchronisation sets defined by the channels in their interface. (In CSP terminology, these block diagrams are called connection diagrams.)

The synchronisation required by the parallelism in the model of a network of processes determines the possible flows of execution for the diagram. A connection is modelled by a synchronisation on the same channel. In the case of our model, the channels are those that represent inputs and outputs of the diagram, and those named after the outputting block with the *_out* suffix.

For the PID, for example, synchronisation ensures that the input taken through the channel $E$ is shared by the processes *Diff*, *Sp*, and *Si*. On the other hand, since these processes do not synchronise with each other on any other of their data channels, their subsequent execution is independent. Each block process recurses to proceed with the next cycle of calculations; to make sure that they all finish (and start) a new cycle together, we require that they synchronise on *end_cycle*.

Finally, we hide all channels that represent internal wires, rather than inputs and outputs. From the point of view of the user of the control system modelled by the diagram, data flowing in these wires is invisible. In fact, they are just a modelling device used to specify the system as a control law diagram. By regarding them as internal channels, we are providing a specification that encapsulates the structure of blocks. In practical terms, this means that, in an implementation, we do not need to have a separate process for each block; refinement can lead to combination and splitting of blocks.

Precisely, the *Circus* model for the whole diagram is a process called df.spec defined as the parallel execution of all the block processes as specified below.

$$\textbf{process}\ \mathsf{df.spec} \mathrel{\widehat=} (\ \|\ \mathsf{B} : \mathrm{dom}\, \mathsf{df.blocks} \bullet (\mathsf{B}, \alpha\mathsf{B})) \setminus (\ \mathsf{Signal} \setminus (\mathsf{df.inputs} \cup \mathsf{df.outputs}))$$

$$\textbf{where}\ \alpha\mathsf{B} = \mathrm{ran}\, \mathsf{df.blocks}(\mathsf{B}).\mathsf{inps} \cup \mathrm{ran}\, \mathsf{df.blocks}(\mathsf{B}).\mathsf{outs} \cup \{\!|\mathit{end\_cycle}|\!\}$$

The alphabet $\alpha\mathsf{B}$ of each block B includes its inputs and outputs, and *end_cycle*. Signal is the set of all channels that correspond to wires in the diagram: all except *end_cycle*. Therefore, the set defined above as Signal \ (df.inputs ∪ df.outputs) includes all channels that represent neither an input nor an output of the diagram: they correspond to the wires that connect blocks.

For the *PID* diagram, the *Circus* model is the process defined below.

$$\textbf{process}\ \mathit{PID} \mathrel{\widehat=} \left(\begin{array}{l} \mathit{Si}\ \{\!|\ \mathit{E}, \mathit{Ki}, \mathit{Si\_out}, \mathit{end\_cycle}\ |\!\} \\ \quad \| \\ \mathit{Diff}\ \{\!|\ \mathit{E}, \mathit{Diff\_out}, \mathit{end\_cycle}\ |\!\} \\ \quad \| \\ \mathit{Int}\ \{\!|\ \mathit{Si\_out}, \mathit{Int\_out}, \mathit{end\_cycle}\ |\!\} \\ \quad \| \\ \mathit{Sd}\ \{\!|\ \mathit{Kd}, \mathit{Diff\_out}, \mathit{Sd\_out}, \mathit{end\_cycle}\ |\!\} \\ \quad \| \\ \mathit{Sp}\ \{\!|\ \mathit{E}, \mathit{Kp}, \mathit{Sp\_out}, \mathit{end\_cycle}\ |\!\} \\ \quad \| \\ \mathit{Sum}\ \{\!|\ \mathit{Sd\_out}, \mathit{Sp\_out}, \mathit{Int\_out}, \mathit{Y}, \mathit{end\_cycle}\ |\!\} \end{array}\right) \setminus \{\!|\ \mathit{Si\_out}, \mathit{Diff\_out}, \mathit{Int\_out}, \mathit{Sd\_out}, \mathit{Sp\_out}\ |\!\}$$

As hinted above, the processes *Si*, *Diff* and *Sp*, for example, are required to synchronise on the input channel *E* that they share, and on *end_cycle*. This is exactly the intersection of their alphabets. Similarly, the internal channel *Diff_out* is in the alphabet of both *Diff* and *Sd*; so, these processes are required to synchronise on *Diff_out* and *end_cycle*. All the *_out* channels are hidden.

By modelling the wiring via channels, and allowing the definition that an output can be produced (that is, communicated) before an input is received, we can cope with feedback loops. More specifically, in such a case, the input and output are modelled by parallel actions. History is kept in the state.

As said before, typically a diagram is hierarchical, in the sense that some blocks may be defined by other diagrams. In general, at the top level we have a diagram with a single block: it takes all the inputs and produces all the outputs of the system. If we use this diagram to generate a *Circus* model, we obtain a single process encapsulating the ClawZ output. This is the most adequate model for the verification of a sequential implementation: we basically use the current ClawZ technique [13]. On the other hand, if we have a parallel implementation as a target, we should work with a *Circus* model of the diagram of the top level block.

In our example, we use the parallel *Circus* model presented in Section 4 for the diagram in Figure 2, because we have a parallel implementation as a target. Since the implementation of the Diff block, for example, is sequential, we do not need to use the alternative parallel model that would be generated by the translation of its diagram in Figure 3. In this parallel model of Diff, there would be, for instance, a channel *Unit_Delay_out* corresponding to the communication between the Unit Delay and the Sum blocks in Figure 3. This parallel model of Diff would be architecturally more elaborate than its sequential implementation. Since *Unit_Delay_out* is internal, this parallel model would be equivalent to the sequential model provided in Figure 10 for Diff, but the latter is more adequate for our verification.

As already indicated, our simple example does not illustrate parallel flows in blocks, but parallelism does show up in the diagram model, reflecting the fact that the three correction actions can be calculated independently. The PID model is appropriate in both size and complexity to illustrate the main concepts and strategies involved in our verification technique. In the next section, we present an implementation of our PID, before discussing how we can prove that such implementation is correct.

## 5. Ada programs and their *Circus* models

The only realistic design for a system like the PID is a sequential implementation, because this is a very simple and small control system. In this case, to prove its correctness, we do not need *Circus*: the current technique

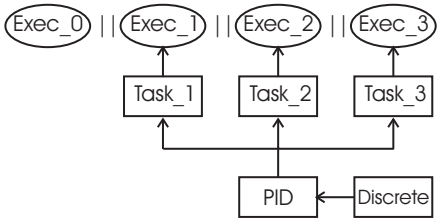


**Fig. 11.** Architecture of the Ada implementation

based on ClawZ is enough. To illustrate the application of our refinement technique, however, we consider a parallel implementation, whose architecture is representative of those commonly used in embedded control systems where time is critical and processing resources are limited. The use of more powerful microprocessors reduces the need for concurrency for performance reasons; however, fault-tolerant architectures still require concurrent master/slave implementations. The growing requirement for multiple linked control systems (such as a flight, engine, and fuel control systems) means that overall system control still requires concurrent implementations as that presented in the sequel for our simple PID example.

Typically, the cycle of the diagram is broken down into time frames, and schedulers determine the subprograms that are executed in each time frame. For our PID example, we have an Ada implementation in which the cycle is broken into two frames. In complex applications, the use of frames is slightly more complicated than this, with the need for major and minor time frames, but using a single kind of time frame is enough to illustrate the principles of our verification technique.

Our implementation comprises four main programs `Exec_0`, `Exec_1`, `Exec_2`, and `Exec_3`, which execute concurrently. The notion of main program is not part of the Ada model of concurrency. In fact, the Ada multi-threading facilities are not used in the implementations that we consider: the main programs are Ada procedures that are executed by different processors. Figure 11 presents the architecture: we use ellipses to distinguish the procedures that correspond to the main programs; the double bars indicate that they run in parallel. Each of them initialises a few variables, and loops; the body of the loop executes for the duration of a time frame, and schedules part of its functionality.

We present in Figure 12 the code for the procedure `Exec_3`. It uses an Ada package `Timing`, which declares constants that characterise the frame, and variables like `Start_Time` and `End_Time`, which are used to define the time to start and to finish the computations of a frame. Another package, `Task_3`, implements the scheduling for `Exec_3`. After executing the initialisation procedure of `Task_3`, that is, `Task_3.Init`, the procedure `Exec_3` loops: at the start of each frame, it carries out the scheduled tasks, as defined in the procedure `Task_3.Step`, and waits until the end of the frame to proceed.

The package `Task_3` is also presented in Figure 12. It uses another package `F_Sch` that declares a frame counter `Cur_F`. It also uses a package `PID`, which implements the functionality of the blocks. The initialisation procedure of `Task_3`, named `Init`, initialises the state of the Diff block using the procedure `Init_Derivative` of the package `PID`, which we present in Figure 13. In the procedure `Step`, `Task_3` schedules `Calc_Derivative`, also a procedure of the `PID` package, in the first frame of every cycle; it carries out the calculations of the Diff and Sd blocks. The implementation of `PID` uses one further package, `Discrete` which provides procedures of general interest to calculate differentials and integrals.

The main programs `Exec_1`, `Exec_2`, and `Exec_3` are all associated with a frame scheduler: `Task_1`, `Task_2`, or `Task_3`. They are depicted in Figure 11, where we use squares to indicate that they are Ada packages; they are connected to the procedures that use them. The procedure `Exec_0` only maintains timing information: it updates, for example, `Start_Time`, `End_Time`, and `Cur_F`. Synchrony between the main programs is maintained by the use of `delay until` commands, which all rely on the values of the shared variables `Start_Time` and `End_Time` to determine the right time to start and end a frame.

In practice, `Exec_0` corresponds to an ASIC timer that regulates the execution of time frames. The procedure `Exec_1` implements the blocks Sp and Sum; `Exec_2` implements the blocks Si and Int; finally, `Exec_3`, as already discussed, implements the blocks Diff and Sd.

To summarise, our verification strategy is for Ada implementations whose architecture can be characterised by : (1) the number of frames in which the cycle is broken; (2) the number of `Exec` procedures that define parallel processes; (3) the set of procedures that implement the functionality of a group of blocks;

```
with Timing; use Timing; with Task_3;

procedure Exec_3 is begin
  Task_3.Init;
  loop
    delay until Start_Time;
    Task_3.Step;
    delay until End_Time;
  end loop;
end Exec_3;
```

```
with F_Sch; use F_Sch; with PID;

package body Task_3 is

  procedure Init is
  begin
    PID.Init_Derivative;
  end Init;

  procedure Step is
  begin
    if Cur_F = 1
    then
      PID.Calc_Derivative;
    else  -- Cur_F = 2
      null;
    end if;
  end Step;
end Task_3;
```

**Fig. 12.** Ada code of `Exec_3` and `Task_3`

and (4) the allocation of these procedures to frames defined by each of the `Task` packages. This architectural pattern, in our experience, is characteristic of applications developed in military avionics.

At the moment, ClawZ can verify the correctness of only the procedures that implement block functionality. Our strategy covers their coordinated use in the way just explained. As a side effect, it ensures that any assumptions taken as preconditions for the verification of a procedure are discharged. This is achieved with the same level of automation of ClawZ, which has already proved to be acceptable in an industrial setting.

To prove that an implementation of a diagram is correct, we use the *Circus* model of the diagram constructed as discussed in the previous section, a *Circus* model of the Ada program, and an algebraic refinement strategy. Most of the model of the program can be calculated automatically using a *Circus* semantics for (a subset of) Ada, that is, a semantics that characterises Ada programs using *Circus* specifications. The only hurdle is that, as explained above, scheduling is based on shared variables that record time periods (like `Start_Time` and `End_Time`) and on a `delay` command. This can be handled directly by *Circus Time* [50, 48], the timed extension of *Circus*, but here we use synchronisation on *end_cycle* and on an extra channel *frame*. Therefore, we do not need the variables `Start_Time` and `End_Time` used in the program.

The *Circus* model of the program contains a process for each `Exec` procedure; the structure of packages is not preserved in these processes. In our example, we have four processes that define the parallel programs. Inputs and outputs are communicated through the channels defined in the model of the diagram. Moreover, shared variables in the program have their values communicated through internal channels: we declare an extra channel for each shared variable. In our example, we have two shared variables `D` and `I` that are declared in the specification of `PID` (see Figure 13); so we declare two extra channels, *Dsh* and *Ish*, that are used to communicate the values of `D` and `I` that are shared by the task packages.

The model for the `Exec` procedure that represents the timer is determined by the number of frames of each cycle. In our example, this is the process that models `Exec_0`; it is shown in Figure 14. The process $Exec_0$ keeps track of the number *cur_f* of the current frame, which corresponds to the Ada variable `Cur_F`. In every frame, $Exec_0$ outputs this number through the channel *frame*, and at the end of the second frame synchronises on *end_cycle*. This captures the interpretation of the timing variables in terms of the channels *frame* and *end_cycle*. The type *FrameIndex* is used (in the program and in its model) to number the frames.

The models for the other `Exec` procedures are similar, but they take into account the allocation of procedures to frames, and the sharing of variables. The state components are the variables that are used directly or indirectly by the `Exec` procedure. The actions are in direct correspondence with the procedures that it allocates. The main action defines the behaviour of the process as defined in the `Exec` procedure itself. We present the model of `Exec_3` in Figure 15; it is derived by flattening `Exec_3`, `Task_3`, `PID`, and

```
with Discrete;
with External_Inputs; use External_Inputs;
with External_Outputs; use External_Outputs;

package body PID is

  Diff_Mem : Float;

  procedure Init_Integral is
  begin
    I := 0.0;
  end Init_Integral;

  procedure Init_Derivative is
  begin
    Diff_Mem := 0.0;
  end Init_Derivative;

  procedure Calc_Proportion is
  begin
    P := Kp * Error;
  end Calc_Proportion;

  procedure Calc_Integral is
  begin
    Discrete.Integ (
      Input  => Error,
      K      => Ki,
      Output => I);
  end Calc_Integral;

  procedure Calc_Derivative is
  begin
    Discrete.Diff (
      Input  => Error,
      K      => Kd,
      Mem    => Diff_Mem,
      Output => D);
  end Calc_Derivative;

  procedure Calc_Output is
  begin
    Position := P + I + D;
  end Calc_Output;

end PID;
```

**Fig. 13.** Ada code of `PID`

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

**Fig. 14.** *Circus* model of `Exec_0`

`Discrete`. Jointly, they declare and use variables `Error`, `Kd`, `Diff_Mem`, and `D`. They also define procedures `Init_Derivative`, `Diff`, `Calc_Derivative`, and `Step`. As shown in Figure 12, the `Exec_3` procedure, after the initialisation, iterates indefinitely executing the procedure `Step`.

Like the procedure `Step`, the action *Step* captures the functionality of a time frame. It finds out the index of the current frame using the channel *frame*. In the first frame of a cycle, the derivative is calculated using the procedure `Calc_Derivative`. In the model, the relevant inputs are taken in interleaving before the corresponding action *Calc_Derivative* is called. This is based on a correspondence between the program variables and the wires of the diagram: in our example, between `Error` and $E$, and `Kd` and $Kd$.

As further discussed in the next section, our technique requires an analysis of the diagram and the program to establish not only how program variables correspond to wires, but also how procedures correspond to blocks. This activity is part of the ClawZ verification process, and therefore, we have evidence that it is acceptable in practice. Controlled experiments inside QinetiQ have indicated a reduction factor between two and a half and four and a half in the cost of establishing acceptance using ClawZ. Overall, the reduction

$$
\begin{array}{l}
\textbf{process}\ \mathit{Exec}_3 \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ [\mathit{Error}, \mathit{Kd}, \mathit{Diff\_Mem}, D : \mathbb{U}] \\
\quad \mathit{Init\_Derivative} \mathrel{\widehat=} \mathit{Diff\_Mem} := 0.0 \\
\quad \mathit{Diff} \mathrel{\widehat=} \left( \begin{array}{l} \textbf{val}\ \mathit{Input}, K : \mathbb{U};\ \textbf{vres}\ \mathit{Mem} : \mathbb{U};\ \textbf{res}\ \mathit{Output} : \mathbb{U} \bullet \\ \quad \mathit{Output} := K * (\mathit{Input} - \mathit{Mem})\,;\, \mathit{Mem} := \mathit{Input} \end{array} \right) \\
\quad \mathit{Calc\_Derivative} \mathrel{\widehat=} \mathit{Diff}\,(\mathit{Error}, \mathit{Kd}, \mathit{Diff\_Mem}, D) \\
\quad \mathit{Step} \mathrel{\widehat=} \left( \begin{array}{l} \mathit{frame}?f \rightarrow \\ \quad \left( \begin{array}{l} \textbf{if}(f = 1) \rightarrow \\ \quad \left( \begin{array}{l} ((E?x \rightarrow \mathit{Error} := x) \,[\![\, \{\, \mathit{Error} \,\} \mid \{\, \mathit{Kd} \,\} \,]\!]\, (\mathit{Kd}?x \rightarrow \mathit{Kd} := x))\,; \\ \mathit{Calc\_Derivative} \end{array} \right) \\ [\!]\ (f = 2) \rightarrow \mathit{Dsh}!D \rightarrow \mathit{end\_cycle} \rightarrow \mathit{Skip} \\ \textbf{fi} \end{array} \right) \end{array} \right) \\
\quad \bullet\ \mathit{Init\_Derivative}\,;\, (\mu\, X \bullet \mathit{Step}\,;\, X) \\
\textbf{end}
\end{array}
$$

**Fig. 15.** *Circus* model of `Exec_3`

when compared to conventional verification of safety-critical avionics systems is of 20%. Moreover, the correspondence between the wires of the diagram and the channels of its *Circus* model is direct.

In *Step*, the value $x$ taken from the channel $E$ is stored in the state component *Error*, and the value taken from *Kd* is stored in the state component of the same name. As mentioned above, the inputs are taken in interleaving. The interleaved action that takes input from $E$ has write access to *Error*, and the action that takes input from *Kd* has write access to the variable *Kd*.

Since $D$ is a shared variable that is calculated by $\mathit{Exec}_3$, its value is communicated in the second frame. For that, the internal channel *Dsh* is used. This value is read by the process $\mathit{Exec}_1$ as shown below. The modelling of the use of shared variables as communications is very simple: we identify the frames that write and the frames that read the variables. If we identify a frame in which a variable is both written and read, we have already identified a potential problem in the program, and there is no need to proceed with the verification: the potential racing has to be eliminated. Otherwise, we insert the required read and write communications over the internal channel that represents the variable.

To further illustrate the technique, we consider the model of `Exec_1`, in which the *Step* procedure collects the values of the shared variables, and produces an output at the end of the cycle.

$$
\mathit{Step} \mathrel{\widehat=} \mathit{frame}?f \rightarrow \left( \begin{array}{l} \textbf{if}(f = 1) \rightarrow \\ \quad \left( \begin{array}{l} ((E?x \rightarrow \mathit{Error} := x) \,[\![\, \{\, \mathit{Error} \,\} \mid \{\, \mathit{Kp} \,\} \,]\!]\, (\mathit{Kp}?x \rightarrow \mathit{Kp} := x)); \\ \mathit{Calc\_Proportion} \end{array} \right) \\ [\!]\ (f = 2) \rightarrow \left( \begin{array}{l} ((\mathit{Ish}?x \rightarrow I := x) \,[\![\, \{\, I \,\} \mid \{\, D \,\} \,]\!]\, (\mathit{Dsh}?x \rightarrow D := x)); \\ \mathit{Calc\_Output}; \\ Y!\mathit{Position} \rightarrow \mathit{end\_cycle} \rightarrow \mathit{Skip} \end{array} \right) \\ \textbf{fi} \end{array} \right)
$$

The actions *Calc_Proportion* and *Calc_Output* correspond to procedures of the same name that implement the functionality of the blocks `Sp` and `Sum`; they are in the package `PID`. The program variables $I$ and $D$ become state components of $\mathit{Exec}_1$. They are shared variables in the program, but are set here using the values communicated via the internal channels *Ish* and *Dsh*.

The model of the complete Ada program is given by the parallel composition of the processes that model the `Exec` procedures. The alphabet of the processes are defined by all the channels that they use; *frame* and

the channels representing shared variables are hidden. In our example, we have the process *AdaPID* below.

$$\textbf{process}\ AdaPID \mathrel{\widehat=} \left( \begin{array}{l} Exec_0 \ \{\!|\ frame, end\_cycle\ |\!\} \\ \quad || \\ Exec_1 \ \{\!|\ frame, E, Kp, Ish, Dsh, Y, end\_cycle\ |\!\} \\ \quad || \\ Exec_2 \ \{\!|\ frame, E, Ki, Ish, end\_cycle\ |\!\} \\ \quad || \\ Exec_3 \ \{\!|\ frame, E, Kd, Dsh, end\_cycle\ |\!\} \end{array} \right) \setminus \{\!|\ frame, Dsh, Ish\ |\!\}$$

In the design of the Ada programs that we consider here (see Figure 11), there is no explicit use of the concurrency facilities of Ada. Concurrency is achieved using mechanisms external to the language. Basically, there is a main program for each processor used in the system implementation. As shown above, the *Circus* model captures the parallel execution of these programs.

Our example program is representative of real applications, in particular in its treatment of cycles, scheduling, and sharing. With the *Circus* model of the PID diagram presented in the previous section, and the *Circus* model of the Ada program just described, we are now in a position to establish correctness.

## 6. Refinement strategy

The existing refinement strategy for *Circus* [15] is concerned with the development of concurrent implementations from centralised specifications; it is based on algebraic laws of refinement in the style of [41], for example. The scenario in the verification of implementations of control law diagrams is different. The diagrams present massive opportunities for parallelism, and our model is a parallel composition of blocks. Implementations, on the other hand, usually provide sequential algorithms to implement groups of blocks.

Many of the existing refinement laws of *Circus* are still useful, but we need extra laws. In addition, since the specification (model of the diagram) is highly structured, we can provide guidance, and therefore automation, in the application of the laws, if we have a particular implementation architecture in mind, and can identify the correspondence between components of that architecture and the diagram. In this paper, we consider the program architecture identified in the previous section.

We present a refinement strategy for proving that a *Circus* model of a diagram, *PID* in our example, is refined by the *Circus* model of a parallel Ada implementation, *AdaPID* in our example. The strategy prescribes the application of a number of *Circus* refinement laws. The semantics of *Circus* [46] is based on Hoare and He's unifying theories of programming. This model and its mechanisation in ProofPower [45] are the basis for the proof of soundness of the laws. Soundness of our strategy follows from the soundness of the individual refinement laws used. They are presented in Appendix B, with the novel laws marked.

In the refinement strategy, we have three aims: (1) collapse the parallelism of the specification to match the architecture of the implementation;(2) prove the correctness of the implementation of the functionality of the blocks; and (3) follow a uniform approach that can be automated by tactics of refinement expressed using a tactic language like that presented in [43]. The strategy comprises the following four phases.

**NB Normalise blocks** For each block, refine the corresponding *Circus* process in the diagram model to write its main action in a normal form: a recursion that iteratively executes an action that captures the behaviour of a cycle as an interleaving of inputs, followed by output calculations and state update, followed by an interleaving of outputs, and synchronisation on *end_cycle*. The successful completion of this phase confirms that the blocks can be implemented sequentially; only syntactic checks are required.

**BJ Blocks join** Collapse the parallelism between the processes of the blocks that are implemented by a single procedure in the Ada program, and then between the processes that represent procedures that are handled by a single scheduler. The success of this phase confirms that the architecture of the implementation is appropriate, in the sense that it groups blocks and procedures that can be implemented sequentially. Again, only syntactic checks are raised by the law applications.

**Pr Procedures** For each of the processes created in phase **BJ**, introduce the action in the model of the program that specifies the corresponding procedure, and prove that the calculations of the outputs and

the state updates can be refined by a call to that action. This requires proof of a number of verification conditions, which can be discharged using the existing ClawZ tools (with a very high level of automation).

**Sc Scheduler** Refine the process that corresponds to the system to get the main programs. Success guarantees that the scheduling of the procedures is correct; only syntactic checks are required.

In the following sections, we discuss refinement strategies for each of these phases.

As mentioned before, the application of our strategy requires the identification of the correspondence between the architectures of the diagram and of the implementation. Namely, for each Ada procedure, we identify the blocks that it implements. We also establish the correspondence between the wires and state information in the diagram with the program variables. The identification of these correspondences is already part of the ClawZ technique; this requirement does not impair scalability. Finally, we determine the number of frames used in the implementation, for each main program, we identify the procedures that it schedules, and, for each procedure that implements block functionality, the number of the frame to which it is allocated; it is trivial to retrieve this information from the model of the program, or from the program itself.

## 6.1. Phase NB: normalise blocks

To normalise the model of a block we (a) remove the parallelism between the actions that model the flows of execution and the state update, and (b) promote the local variables of the main action to state components. This is only possible if all the flows require all the inputs. If not, then there is at least one flow that may produce its outputs before all the inputs arrive; for these, a sequential implementation that waits for all the inputs is not correct: a parallel implementation that decouples the production of (some) outputs from the arrival of all inputs is required. If the implementation under verification implements the block sequentially, the failure of this phase of the refinement strategy indicates that problem.

If, on the other hand, we have a parallel implementation for the block functionality, then the centralised model of the block is an inadequate starting point for the application of our refinement strategy. In this case, if the architecture of the implementation is related to that of the diagram of the block, then, as said before, we should use the model of this diagram for verification. If not, the existing *Circus* refinement strategy can be used. In our experience, it is almost always possible to relate the architecture of an implementation to a diagram, in the sense that we can map procedures and main programs to the blocks that they implement and schedule. As highlighted above, our strategy explores this relationship.

Precisely, in this phase, we tackle blocks whose flows are combined as in Figure 17, Configuration (4). (In particular, in the main action of the block processes, the state update is combined in this way with the flows.) In these cases, the refinement steps in Figure 16 succeed, when applied to the main actions of the processes that model the blocks: all but the one that models the diagram. Each step is supported by refinement laws listed in Appendix B; we discuss here just the novel and specific laws.

We use the main action of the process *Diff* (Figure 10) reproduced below to illustrate the refinement steps.

$$
\begin{array}{l}
Init; \\
\left(\begin{array}{l}
\mu\, X \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{var } In1 : \mathbb{U} \bullet \\
\quad \left(\begin{array}{l} E?x \rightarrow In1 := x; \\ \textbf{var } Out1 : \mathbb{U} \bullet Calculate\_Diff\_out;\ Diff\_out!Out1 \rightarrow Skip \end{array}\right)
\end{array}\right) \\
\quad [\![ \{\} \mid \{\!|E|\!\} \mid \{pid\_\_Diff\_\_UnitDelay\_state\} ]\!] \\
\left(\begin{array}{l}
\textbf{var } In1 : \mathbb{U} \bullet E?x \rightarrow In1 := x;\ Calculate\_Diff\_State \\
end\_cycle \rightarrow X
\end{array}\right)
\end{array}\right);
\end{array}\right)
\end{array}
$$

For clarity, we apply a copy-rule to eliminate all references to action names: Law (copy-rule-action). After that, we apply the steps of refinement as explained below.

1. **Synchronise inputs.** Since all flows require all inputs, as does the state update, all parallel actions in the body of the recursion declare local variables to hold each of the input values, and take all of them in interleaving. (In our example, an interleaving is not needed because we have a single input.) In this step, we extract from the parallelism the variable declarations and the interleaving using a version of

Refine (the main action) of all block processes as follows.

1. **Synchronise inputs.** Apply laws like Law (var-int-par-join) to extract from the parallelism the variable declarations and the interleaving.
2. **Expand the scope of the output variables.** Apply Laws (var-exp-par), (var-exp-seq) and (join-blocks) to expand the blocks that declare the Outj variables, and join the resulting blocks.
3. **Isolate the input processing.** Exhaustively apply Law (par-seq-step) to remove the parallelism.
4. **Introduce interleaving of outputs.** Apply Law (par-inter), and then exhaustively Law (inter-unused-name).
5. **Simplify the interleaving.** Apply Law (inter-unit).
6. **Extend the scope of the variable declarations to the outer level.** Apply Laws (var-exp-seq) and (var-exp-rec) exhaustively.
7. **Turn the input and output variables into state components.** Apply Law (main-var-state) (to the complete basic process).

**Fig. 16.** Refinement strategy: phase **NB**

Law (var-int-par-join) below, which considers in detail the case of two inputs.

**Law[var-int-par-join]**

$$\left(\begin{array}{l}(\mathbf{var}\ x_1 : T_1;\ x_2 : T_2 \bullet (c?x \rightarrow x_1 := x \ [\![\{x_1\} \mid \{x_2\}]\!]\ d?y \rightarrow x_2 := y);\ A_1) \\ \quad [\![ns_1 \mid \{c, d\} \cup cs \mid ns_2]\!] \\ (\mathbf{var}\ x_1 : T_1;\ x_2 : T_2 \bullet (c?x \rightarrow x_1 := x \ [\![\{x_1\} \mid \{x_2\}]\!]\ d?y \rightarrow x_2 := y);\ A_2)\end{array}\right)$$
$$=$$
$$\left(\begin{array}{l}\mathbf{var}\ x_1 : T_1;\ x_2 : T_2 \bullet \\ \quad (c?x \rightarrow x_1 := x \ [\![\{x_1\} \mid \{x_2\}]\!]\ d?y \rightarrow x_2 := y);\ (A_1 \ [\![ ns_1 \mid \{c, d\} \cup cs \mid ns_2 ]\!]\ A_2)\end{array}\right)$$

**provided** $\{x_1, x_2\} \cap (ns_1 \cup ns_2) = \varnothing$

This law emphasis that if a variable is not in the name set of a parallel action, then any use that it makes of that variable has only a local effect. If, as above, we have two parallel actions that declare local variables $x_1$ and $x_2$, then we can, instead, declare these variables before the parallelism, as long as $x_1$ and $x_2$ are not included in the name sets of the parallel actions. This is guaranteed by the proviso of the law. In this case, just as before, both parallel actions have access to the initial values of the variables, and any changes that they make are not visible. Moreover, Law (var-int-par-join) establishes that, since the parallel actions initialise $x_1$ and $x_2$ in the same way, then this initialisation also can be extracted from the parallelism. In general, such extraction can change the value of $x_1$ and $x_2$ beyond the parallelism, when in fact its changes to these variables are, as already said, originally visible only locally. Since in this case, however, the scope of $x_1$ and $x_2$ finishes right after the parallelism, this is not a concern.

For *Diff*, we have the following result after applying a simplified version of Law (var-int-par-join).

$$\begin{array}{l}Init; \\ \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\mathbf{var}\ In1 : \mathbb{U} \bullet \\ \quad E?x \rightarrow In1 := x; \\ \quad \left(\begin{array}{l}\mathbf{var}\ Out1 : \mathbb{U} \bullet Calculate\_Diff\_out;\ Diff\_out!Out1 \rightarrow Skip \\ \quad [\![\{\,\} \mid \{\!|E|\!\} \mid \{pid\_\_Diff\_\_UnitDelay\_state\}]\!] \\ Calculate\_Diff\_State\end{array}\right)\end{array}\right)\end{array}\right); \\ end\_cycle \rightarrow X\end{array}\right)\end{array}\right)\end{array}$$

We do not have an interleaving of inputs, but extract from the parallelism the declarations of $In1$ and the corresponding initialisations using the value input through $E$.

2. **Expand the scope of the output variables.** Since there are no repeated declarations of output variables, because each output is handled by a single flow, we can expand the scope of the blocks that

introduce the Outj variables, and join the resulting nested blocks. This can be achieved by applying Laws (var-exp-par), (var-exp-seq) and (join-blocks). For *Diff*, the result is as follows.

$$Init;\ \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\textbf{var}\ In1, Out1 : \mathbb{U} \bullet \\ \left(\begin{array}{l} E?x \rightarrow In1 := x; \\ \left(\begin{array}{l} Calculate\_Diff\_out;\ Diff\_out!Out1 \rightarrow Skip \\ \quad [\![ \{Out1\} \mid \{\!|E|\!\} \mid \{pid\_\_Diff\_\_UnitDelay\_state\} ]\!] \end{array}\right) \\ Calculate\_Diff\_State \end{array}\right)\end{array}\right)\end{array}\right) \\ end\_cycle \rightarrow X\end{array}\right);\end{array}\right)$$

3. **Isolate the input processing.** To remove the remaining parallelism, the schemas that process the inputs to define the values of the outputs and the state updates are extracted by the repeated application of Law (par-seq-step). For our example, we obtain the result below.

$$\begin{array}{l} Init; \\ \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\textbf{var}\ In1, Out1 : \mathbb{U} \bullet \\ \left(\begin{array}{l} E?x \rightarrow In1 := x; \\ Calculate\_Diff\_out;\ Calculate\_Diff\_State; \\ \left(\begin{array}{l} Diff\_out!Out1 \rightarrow Skip \\ \quad [\![ \{Out1\} \mid \{\!|E|\!\} \mid \{pid\_\_Diff\_\_UnitDelay\_state\} ]\!] \end{array}\right) \end{array}\right)\end{array}\right)\end{array}\right); \\ end\_cycle \rightarrow X\end{array}\right)\end{array}\right)\end{array}$$

Since *Calculate_Diff_out* does not change the state, it cannot interfere with *Calculate_Diff_State*, which does use the state components; therefore, the proviso of Law (par-seq-step) is satisfied.

4. **Introduce interleaving of outputs.** In the remaining parallelism, none of the inputs are taken, but the channels are in the synchronisation set are exactly the input channels. We can, therefore, turn the parallelism into an interleaving, using Law (par-inter). We can also use Law (inter-unused-name) to empty the name sets of the resulting interleaving, since there are no update operations left.

$$Init;\ \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\left(\begin{array}{l}\textbf{var}\ In1, Out1 : \mathbb{U} \bullet \\ \left(\begin{array}{l} E?x \rightarrow In1 := x; \\ Calculate\_Diff\_out;\ Calculate\_Diff\_State; \\ (Diff\_out!Out1 \rightarrow Skip \ [\![ \{\} \mid \{\} ]\!]\ Skip) \end{array}\right)\end{array}\right); \\ end\_cycle \rightarrow X\end{array}\right)\end{array}\right)$$

The output on *Diff_out* is now in interleaving, rather than in parallel, with *Skip*, and therefore, we do not need to indicate a synchronisation set, in this case $\{\!|E|\!\}$, anymore.

5. **Simplify the interleaving.** Due to the processing of the state, that does not produce any output, one of the interleaved actions is always *Skip*. We remove it using a unit law for interleaving: Law (inter-unit). In our example, this removes the interleaving altogether, because Diff has only one output; in general, we are left with an interleaving of the outputs (of the block).

$$Init;\ \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\left(\begin{array}{l}\textbf{var}\ In1, Out1 : \mathbb{U} \bullet \\ \left(\begin{array}{l} E?x \rightarrow In1 := x; \\ Calculate\_Diff\_out;\ Calculate\_Diff\_State; \\ Diff\_out!Out1 \rightarrow Skip \end{array}\right)\end{array}\right); \\ end\_cycle \rightarrow X\end{array}\right)\end{array}\right)$$

6. **Extend the scope of the variable declarations to the outer level.** For that, we use standard variable block laws: Laws (var-exp-seq) and (var-exp-rec).

$$\left(\begin{array}{l}\textbf{var}\ In1, Out1 : \mathbb{U} \bullet \\ \left(\begin{array}{l} Init; \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l} E?x \rightarrow In1 := x; \\ Calculate\_Diff\_out;\ Calculate\_Diff\_State; \\ Diff\_out!Out1 \rightarrow Skip; \\ end\_cycle \rightarrow X \end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right)$$

7. **Turn the input and output variables into state components.** This is possible because they are

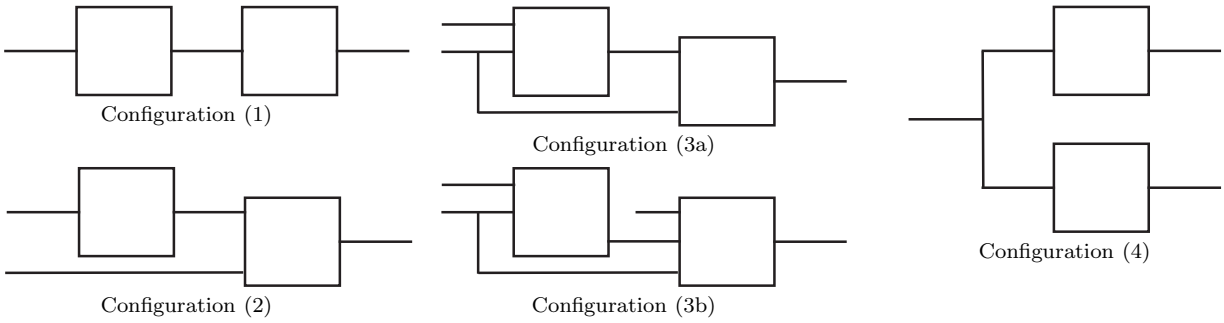


**Fig. 17.** Block configurations

| Procedure | Blocks | Scheduler | Frame |
|---|---|---|---|
| Calc_Proportion | Sp | Exec_1 | 1 |
| Calc_Output | Sum | Exec_1 | 2 |
| Calc_Integral | Si, Int | Exec_2 | 1 |
| Calc_Derivative | Diff, Sd | Exec_3 | 1 |

**Table 1.** PID: correspondence between procedures of the Ada program and blocks of the diagram

local to the whole main action. Law (main-var-state), which justifies this step, applies to a complete basic process, rather than to actions. For our example, the resulting main action is as follows.

$$Init; \left( \mu X \bullet \left( \begin{array}{l} E?x \rightarrow In1 := x; \\ Calculate\_Diff\_out;\ Calculate\_Diff\_State; \\ Diff\_out!Out1 \rightarrow Skip; \\ end\_cycle \rightarrow X \end{array} \right) \right)$$

The variables $In1$, and $Out1$ are now state components of $Diff$.

In our example, the processes corresponding to the blocks Si, Sd, Sp, and Sum do not have a state, and have only one flow of execution. We, therefore, after Step (1), proceed to Step (7), because there are no parallel actions in their main action to be handled. For the process $Int$, which corresponds to the remaining block Int, the verification is very similar to that illustrated above for $Diff$.

## 6.2. Phase BJ: blocks join

In this phase, we need information about the Ada procedures that implement block functionality, namely, the blocks that they implement, and about the procedures handled by each scheduler. For our example, investigating the program Exec_3, we identify Calc_Derivative, the procedure that implements the functionality of the blocks Diff and Sd. The other procedures in Exec_3 do not implement blocks: the procedure Init_Derivative is a state initialisation, Diff is used in Calc_Derivative, and Step is part of the scheduler in Exec_3. In considering the program Exec_2, we also find a Calc_Integral procedure which implements the blocks Si and Int. Finally, the main program Exec_1 has procedures Calc_Proportion, which implements the block Sp, and Calc_Output, which implements the block Sum. Table 1 gives a summary of the kind of information about the procedures that needs to be collected for our example.

This refinement phase tackles, first, each of the procedures that implement more than one block. For each of them, we consider the processes that model the blocks that they implement: we remove, in the process that defines the diagram, the parallelism between these processes. As a result, we create a single process for each procedure. For that, we consider two blocks at a time, and proceed as shown below, and summarised in Figure 18. Afterwards, with the collection of processes now in correspondence with the procedures of the implementation, we proceed in much the same way to group the processes that correspond to procedures scheduled by a single task (main program). At the end, we have a process for each of the schedulers.

**For each set of processes that model blocks implemented by the same procedure**, apply exhaustively the refinement steps below to each pair of processes (until a single process is left).

1. **Create a single process.** Apply definition of process parallelism [46].
2. **Extract initialisations.** Apply Law (par-seq-step).
3. **Extract the synchronisation on** *end_cycle*. Apply Law (rec-sync).

*Remove parallelism.* The refinement laws to be used next depend on the arrangement of the blocks represented by the pair of processes under consideration. Figure 17 gives all configurations for which this step can succeed, since the overall behaviour can be sequentialised.

**For Configurations (1) to (3)**, proceed as follows.

4. **Evaluate the synchronisation entailed by the internal communications.** For that, we apply specific laws that depends on the number of inputs and outputs of the blocks. For example, for Configuration (2), we have laws like Law (par-out-inp-inter-exchange).
5. **Remove internal communications.** For each internal communication, apply the version of Law (hid-join) for processes, followed by Law (hid-par-dist) to localise the hiding of the internal channel around the process created in Step (1) above. Apply the definition of process hiding [46], and, then exhaustively apply distribution laws of action hiding to localise the hiding around the prefixing with the internal communication. To conclude, apply Law (hid-step).
6. **Sequentialise assignments.** Apply Law (inter-seq).
7. **Introduce interleaving of inputs.**
   (a) **Keep just the inputs in the interleaving** Apply Law (inter-seq-extract-snd) exhaustively.
   (b) **Leave in the name sets only the input variables** Apply Law (inter-unused name).
   (c) **Flatten the interleaving**. Apply Law (inter-assoc)).

**For Configuration (4)**, proceed as follows.

4. Apply a version of Law (var-int-par-join), for the right number of inputs.
5. Carry out Steps (3) and (4) in Figure 16.

**For each set of processes that model procedures scheduled by the same scheduler**, apply exhaustively to each pair of processes the refinement steps above, except only for Step (7) (in the case of Configurations (1) to (3)). Instead, Step (7) below should be applied at the end.

7. **Introduce input of shared variables**. For each channel corresponding to a shared variable, apply the version of Law (hid-join) for processes an Law (hid-par-dist) to localise its hiding around the process created in Step (1), apply the definition of process hiding [46]. Apply a version of Law (inter-split) for the right number of inputs, and use again Laws (hid-join) and (hid-par-dist), and the definition of process hiding, to move the hiding back out.

**Fig. 18.** Refinement strategy: phase **BJ**

To illustrate the steps of this phase, we consider the `Calc_Derivative` procedure, that is, we join the processes *Diff* and *Sd*, which model Diff and Sd. In our example, we also need to tackle the procedure `Calc_Integral`, and we proceed in a similar way. (Later, we consider the procedures `Calc_Proportion` and `Calc_Output`, or equivalently, the blocks Sp and Sum, because they are both scheduled by `Exec_1`.)

1. **Create a single process.** This is achieved using the definition of process parallelism [46]. It describes $P_1 \llbracket\ cs\ \rrbracket P_2$ as a basic process whose state includes all the components of $P_1$ and $P_2$ and whose main action is the parallel composition of the main actions $A_1$ of $P_1$ and $A_2$ of $P_2$. If there are clashes in the names of the state components (or any other definitions) of $P_1$ and $P_2$, they are resolved by renaming. The name sets associated to $A_1$ and $A_2$ in the parallelism are the state components of $P_1$ and $P_2$. For

`Calc_Derivative`, we create a process *DiffSd*; its main action is as follows.

$$
\left(\begin{array}{l}
\left(\begin{array}{l} Init; \left(\mu\, X \bullet \left(\begin{array}{l} E?x \rightarrow pid\_\_Diff\_In1 := x; \\ Calculate\_Diff\_out;\ Calculate\_Diff\_State; \\ Diff\_out!pid\_\_Diff\_Out1 \rightarrow Skip; \\ end\_cycle \rightarrow X \end{array}\right)\right)\end{array}\right) \\
\quad [\![ \{ pid\_\_Diff\_\_UnitDelay\_state, pid\_\_Diff\_In1, pid\_\_Diff\_Out1 \} \mid \\
\qquad \{\!| Diff\_out, end\_cycle |\!\} \mid \\
\quad \{ pid\_\_Sd\_In1, pid\_\_Sd\_In2, pid\_\_Sd\_Out1 \} ]\!] \\
\left(\begin{array}{l} \mu\, X \bullet \left(\begin{array}{l} \left(\begin{array}{l} Kd?x \rightarrow pid\_\_Sd\_In1 := x \\ \quad [\![ \{ pid\_\_Sd\_In1 \} \mid \{ pid\_\_Sd\_In2 \} ]\!] \end{array}\right); \\ Diff\_out?x \rightarrow pid\_\_Sd\_In2 := x \\ pid\_\_Sd; \\ Sd\_out!pid\_\_Sd\_Out1 \rightarrow Skip; \\ end\_cycle \rightarrow X \end{array}\right)\end{array}\right)
\end{array}\right)
$$

The Ini and Outj variables in the state are renamed when the processes are joined to avoid clashes. They are prefixed with the name of the diagram and of the process, and since these are unique, the new names of the variables are also unique. The parallelism requires synchronisation on the intersection of the alphabets of the original processes: in our example, the channels *Diff_out* and *end_cycle*. The parallel actions have write access to the state components of the corresponding original processes.

2. **Extract initialisations.** The initialisations are not implemented in parallel. They are carried out before the scheduling of the procedures that implement block functionality starts, or, in other words, before the program enters the cyclic behaviour defined by the diagram. Therefore, in this step, we remove the initialisations from the parallelism using Law (par-seq-step).

3. **Extract the synchronisation on** *end_cycle*. For that, we use the fixed-point Law (rec-sync).

**Law[rec-sync]**

$$
\begin{array}{l}
(\mu\, X \bullet A_1;\ c \rightarrow X) \,[\![\, ns_1 \mid \{\!|\, c \,|\!\} \cup cs \mid ns_2 \,]\!]\, (\mu\, X \bullet A_2;\ c \rightarrow X) \\
= \\
\mu\, X \bullet (A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2);\ c \rightarrow X
\end{array}
$$

**provided** $c \notin usedC(A_1, A_2);\ wrtV(A_1) \cap usedV(A_2) = \varnothing;$ *and* $wrtV(A_2) \cap usedV(A_1) = \varnothing.$

The first proviso ensures that in the parallelism of recursive actions, the channel $c$ is only used at the end of the bodies $A_1;\ c \rightarrow X$ and $A_2;\ c \rightarrow X$ of each recursion. The set $usedC(A)$ contains the channels used by the action, or list of actions, $A$. The synchronisation on $c$ ensures that the recursions proceed in lock-step. This law states that we can establish the lock-step by considering a single recursive action in which $A_1$ and $A_2$ are executed in parallel in each iteration. There is, however, a concern about the use of data. As an example, we consider the case in which $A_1$ uses a variable $x$ that is modified by $A_2$. Since the recursions never finish, the parallelism of the recursions never finishes. Therefore, $A_1$ never has access to the modified value of $x$; before the parallelism finishes, $A_1$ only has access to the initial value of $x$ (or to any modifications that $A_1$ makes itself). On the other hand, in the context of the single recursion, in each iteration, $A_1$ would take the value of $x$ resulting from the execution of $A_2$ in the previous iteration. In this case, the parallelism starts and finishes at each iteration. The same concern applies to $A_2$ in relation to $A_1$. The second and third provisos, however, establish that the variables that are possibly modified by $A_1$ are not used by $A_2$, and vice-versa. We use $wrtV(A)$ to refer to the set of variables whose values can potentially be changed by the action (or list of actions) $A$, and $usedV(A)$ to refer to the set of variables that are used by $A$. Formal definitions of all the syntactic functions used here can be found in [42].

Proceeding with our example, after this step, we get the following parallel main action.

$$Init;\ \left(\mu\, X \bullet \left(\begin{array}{l} \left(\begin{array}{l} \left(\begin{array}{l} E?x \rightarrow pid\_\_Diff\_In1 := x; \\ Calculate\_Diff\_out;\ Calculate\_Diff\_State; \end{array}\right) \\ Diff\_out!pid\_\_Diff\_Out1 \rightarrow Skip \end{array}\right) \\ \quad [\![ \{ pid\_\_Diff\_\_UnitDelay\_state, pid\_\_Diff\_In1, pid\_\_Diff\_Out1 \} \mid \\ \qquad \{\!| Diff\_out |\!\} \mid \\ \quad \{ pid\_\_Sd\_In1, pid\_\_Sd\_In2, pid\_\_Sd\_Out1 \} ]\!] \\ \left(\begin{array}{l} \left(\begin{array}{l} Kd?x \rightarrow pid\_\_Sd\_In1 := x \\ \quad [\![ \{ pid\_\_Sd\_In1 \} \mid \{ pid\_\_Sd\_In2 \} ]\!] \\ Diff\_out?x \rightarrow pid\_\_Sd\_In2 := x \end{array}\right); \\ pid\_\_Sd; \\ Sd\_out!pid\_\_Sd\_Out1 \rightarrow Skip \end{array}\right) \\ end\_cycle \rightarrow X \end{array}\right)\right);$$

The parallelism of recursions becomes a recursive parallel action, with the synchronisation on *end_cycle* outside the parallelism, which no longer requires synchronisation on this channel.

*Remove the parallelism* The steps required to achieve this objective depend on the way in which the blocks are arranged. Also, collapsing parallelism is not always possible: we combine blocks connected in sequence. (As said before, if we have more than two blocks to combine, we collapse two at a time.)

The configurations presented in Figure 17 cover all the cases. In the first three, the final output, that is, the output of the second block, depends on all outputs of the first block. The communications of the outputs of the first block to the second one are internal, and can be eliminated. Configuration (4) involves no internal channels and, therefore, the removal of the parallelism is simpler: we extract the common interleaving of inputs using a variation of Law (var-int-par-join), and then we proceed as in Steps (3) and (4) of phase **NB**. Proceeding with our example, we observe that the blocks Diff and Sd are connected according to Configuration (2) of Figure 17, so we carry out the steps illustrated below.

4. **Evaluate the synchronisation entailed by the internal communications.** Highly specialised, but similar, laws justify this step. For each configuration, we have one law, and variations that take into account the different number of inputs and outputs. The law used in this step for our example is presented below. It is useful when the first block has one output, represented by the communication $c_1$, which requires synchronisation with one of the two interleaved inputs of the second block.

**Law[par-out-inp-inter-exchange]**

$$\begin{array}{l} (A_1;\ c_1 \rightarrow Skip) \,[\![\ ns_1 \mid \{\!|\, c_1 \,|\!\} \mid ns_2 \,]\!]\, ((c_1 \rightarrow A_2 \,[\![\, ns_3 \mid ns_4 \,]\!]\, c_2 \rightarrow A_3);\ A_4) \\ = \\ (A_1;\ c_1 \rightarrow A_2 \,[\![ns_1 \cup ns_3 \mid ns_4]\!]\, c_2 \rightarrow A_3);\ A_4 \\ \textbf{provided}\ \ c_1 \neq c_2;\ c_1 \notin usedC(A_1, A_2, A_3, A_4); \\ \qquad\qquad ns_3 \cup ns_4 \subseteq ns_2;\ wrtV(A_1) \subseteq ns_1;\ wrtV(A_2) \subseteq ns_3;\ \textit{and}\ wrtV(A_4) \subseteq ns_2. \end{array}$$

The provisos guarantee that the only use of $c_1$ is that explicitly indicated. In this case, an application of Law (par-out-inp-inter-exchange) evaluates the synchronisation: the communication is joined with the processing of the communicated value in $A_2$, and the parallelism is removed. All that remains is an interleaving (of inputs); since $A_4$ is not concerned with the communications in question, it is kept out of the interleaving. The provisos guarantee that the removal of the parallelism does not make changes that used to be local to become global. For example, if $ns_3$ and $ns_4$ are contained in $ns_2$, then the changes to the variables of $ns_3$ and $ns_4$ that are possibly carried out by $A_2$ and $A_3$ were not masked by the removed parallelism, and are not affected. Similarly, it is required that the changes that can be carried by $A_1$, $A_2$, and $A_4$ were not masked. For $A_3$, the name set $ns_4$ is used in both the original parallelism and in the remaining interleaving, so we do not need to impose any further restrictions.

Straightforward generalisations of Law (par-out-inp-inter-exchange) handle cases in which there are several interleaved output communications, instead of just $c_1$, as long as all these outputs are matched to an input in the parallel action. Several extra inputs to the second block, instead of just $c_2$, can also be easily handled. Finally, the simplification of Law (par-out-inp-inter-exchange) for the case in which there is no extra input $c_2$ to the second block is also trivial.

In our example, the internal communication is through $Diff\_out$; the result is as follows.

$$
\begin{array}{l}
Init; \\
\left(\begin{array}{l}
\mu\, X \bullet \left(\begin{array}{l}
\left(\begin{array}{l}
\left(\begin{array}{l}
E?x \rightarrow pid\_\_Diff\_In1 := x; \\
Calculate\_Diff\_out;\ Calculate\_Diff\_State;
\end{array}\right) \\
Diff\_out!pid\_\_Diff\_Out1 \rightarrow pid\_\_Sd\_In2 := pid\_\_Diff\_Out1
\end{array}\right) \\
\quad [\![ \{ pid\_\_Diff\_\_UnitDelay\_state, pid\_\_Diff\_In1, pid\_\_Diff\_Out1, pid\_\_Sd\_In2 \} \mid \\
\qquad \{ pid\_\_Sd\_In1 \} ]\!] \\
Kd?x \rightarrow pid\_\_Sd\_In1 := x
\end{array}\right) ; \\
pid\_\_Sd; \\
Sd\_out!pid\_\_Sd\_Out1 \rightarrow Skip; \\
end\_cycle \rightarrow X
\end{array}\right)
\end{array}
$$

The evaluation of the communication defines the input value; in our example, the input value $x$ used in $pid\_\_Sd\_In2 := x$ is determined to be the output value $pid\_\_Diff\_Out1$.

5. **Remove internal communications.** As mentioned, the communication over $Diff\_out$ is internal to this process. This channel was used for communication between the processes *Diff* and *Sd*, but these have now been collapsed into the single process *DiffSd*, and $Diff\_out$ is only used inside this process.

   For each such internal communication arising from the evaluation in the previous step, we basically use the hiding distribution laws to localise the hiding of the internal channel around the prefixing with the communication, in preparation to eliminate it. The hiding is originally in the process that defines the diagram model (see Section 4.4). It is, first of all, localised around the process created in this phase; in our example, *DiffSd*. For that, we use the process version of Law (hid-join) to isolate the hiding of the internal channel, and (a version for the right number of parallel processes of) Law (hid-par-dist). Afterwards, the hiding can be moved to the main action using the definition of hiding for processes: the resulting process is obtained by applying the hiding to the main action. Finally, the hiding can be pushed in towards the prefixing with the internal communication using distribution laws of hiding (for actions). To conclude, we apply Law (hid-step) to remove the communication.

   For our example, as a result of all this, the communication over $Diff\_out$ is removed, and only the assignment in the original prefixing stays. It captures the communication between the blocks Diff and Sd. Since the inputs and outputs of both blocks are now modelled as components of the state of the new process *DiffSd*, there is no longer a need for a communication.

6. **Sequentialise assignments.** If there were several internal communications, Step (5) leaves us with an interleaving of assignments. We transform the interleaving into a sequence of assignments using Law (inter-seq). In *DiffSd*, there is only one internal communication.

7. **Introduce interleaving of inputs.** As illustrated by our example, we are left with an interleaving that may include more than just the inputs. The calculation of outputs, state updates, and the assignments from Steps (5) and (6) may be in the interleaving. We need to simplify it as follows.

   (a) **Keep just the inputs in the interleaving**, by removing all calculations, updates, and assignments using Law (inter-seq-extract-snd) exhaustively. This leaves just prefixings of assignments to input variables in the interleaving. For our example, the result is below.

   $$
   \begin{array}{l}
   Init; \\
   \left(\begin{array}{l}
   \mu\, X \bullet \left(\begin{array}{l}
   (E?x \rightarrow pid\_\_Diff\_In1 := x \ [\![ \ldots ]\!]\ Kd?x \rightarrow pid\_\_Sd\_In1 := x); \\
   Calculate\_Diff\_out;\ Calculate\_Diff\_State; \\
   pid\_\_Sd\_In2 := pid\_\_Diff\_Out1;\ pid\_\_Sd; \\
   Sd\_out!pid\_\_Sd\_Out1 \rightarrow Skip; end\_cycle \rightarrow X
   \end{array}\right)
   \end{array}\right)
   \end{array}
   $$

   For conciseness, we omit the name sets in the interleaving; they do not change.

   (b) **Leave in the name sets only the input variables**; Law (inter-unused name) can be used for that. In our example, we remove $pid\_\_Diff\_\_UnitDelay\_state$, $pid\_\_Diff\_Out1$, and $pid\_\_Sd\_In2$ from the first name set. The second name set already contains only the right input variable.

   (c) **Flatten the interleaving** using associativity (Law (inter-assoc)). For our example, this is not needed because we have just two inputs. For *DiffSd*, the resulting main action is shown in Figure 19. The communication over $Diff\_out$ has become internal, and so it has been replaced with an assignment.

$$Init;\ \left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\left(\begin{array}{l}(E?x \rightarrow pid\_\_Diff\_In1 := x)\\ \quad [\![\{\, pid\_\_Diff\_In1 \,\} \mid \{\, pid\_\_Sd\_In1 \,\}]\!]\\ (Kd?x \rightarrow pid\_\_Sd\_In1 := x)\end{array}\right);\\ Calculate\_Diff\_out;\ Calculate\_Diff\_State;\\ pid\_\_Sd\_In2 := pid\_\_Diff\_Out1;\ pid\_\_Sd;\\ Sd\_out!pid\_\_Sd\_Out1 \rightarrow Skip\,;\, end\_cycle \rightarrow X\end{array}\right)\end{array}\right)$$

**Fig. 19.** Main action of the process *DiffSd* - end of phase **BJ**

Inputs are taken in interleaving from $E$ and $Kd$, the calculations of *Diff* and *Sd* are performed, and the output of *Sd* is produced, before a synchronisation on *end_cycle*.

As already said, the *DiffSd* process so obtained corresponds to the Ada procedure `Calc_Derivative`. We also join the processes *Si* and *Int* to produce a process *SiInt* that corresponds to the procedure `Calc_Integral`. The process *Sp* models the block Sp and already corresponds to the procedure `Calc_Proportion`. Similarly, *Sum* corresponds to the procedure `Calc_Output`. So, all processes correspond to procedures.

Now, we need to consider the schedulers. The parallelism between the processes that model procedures handled by the same scheduler also needs to be collapsed. We proceed much in the same way as above for the removal of parallelism between the processes that model blocks implemented by the same procedure. The idea is that a set of blocks implemented by a single procedure can be seen as a virtual block (now that it is modelled by a single process). Figure 20 uses dashed boxes to indicate the virtual blocks of the PID; some of the virtual blocks are actual blocks, namely, those implemented by a procedure on their own.

As mentioned before, `Exec_1` schedules two procedures: `Calc_Proportion` and `Calc_Output` (which implement the blocks Sp and Sum). Therefore, in this phase, we also join the processes *Sp* and *Sum*, following the same steps above, to produce a process *SpSum*. The corresponding (virtual) blocks are connected according to Configuration (2) in Figure 17, so the refinement steps are really similar, but we do not apply Step (7) in Figure 18. At the end of Step (6), for *SpSum*, we obtain the main action below.

$$\left(\begin{array}{l}\mu\, X \bullet \left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}\left(\begin{array}{l}((E?x \rightarrow pid\_\_Sp\_In1 := x) \;[\![\ldots]\!]\; (Kp?x \rightarrow pid\_\_Sp\_In2 := x))\,;\\ Calculate\_Sp\_out\,;\, pid\_\_Sum\_In2 := pid\_\_Sp\_Out1\end{array}\right)\\ \qquad \{pid\_\_Sp\_In1, pid\_\_Sp\_In2, pid\_\_Sp\_Out1, pid\_\_Sum\_In2\}\\ \qquad [\!|\\ (Sd\_out?x \rightarrow pid\_\_Sum\_In1 := x)\ \{pid\_\_Sum\_In1\}\\ \qquad [\!|\\ (Int\_out?x \rightarrow pid\_\_Sum\_In3 := x)\ \{pid\_\_Sum\_In3\}\end{array}\right);\\ Calculate\_Sum\_out;\\ Sum\_out!pid\_\_Sum\_Out1 \rightarrow Skip\end{array}\right);\\ end\_cycle \rightarrow X\end{array}\right)\end{array}\right)$$

What we have as a result of the refinement is that the interleaving of inputs of the resulting process includes also calculations of outputs and state updates. In our example, the calculation of the output of the block Sp is inside the interleaving. The calculation of the output of Sum, on the other hand, is after the interleaving. We do not join these calculations, which is the objective of Step (7) in Figure 18, because, even though they are scheduled in sequence (by the same scheduler), they are implemented by different procedures. Instead, we handle the input of values of the shared variables that needs to be joined with the processing of these inputs. For the case of the process *SpSum*, we proceed as follows.

7. **Group input of shared variables**. As explained in Section 5, in the model of the Ada program, the shared variables are output after they are calculated, and input when needed. We have, therefore, one internal channel for each shared variable. In the diagram model, these channels are already present: the shared variables correspond to wires in the diagram, which are modelled by channels. For our example, we have the shared variables `I` and `D`, which correspond to the channels *Int_out* and *Sd_out* (see Figure 22).

   At this stage, the outputs through these channels already take place after the calculations of the values of the shared variables. In Figure 19, for example, we have the output through *Sd_out* after the sequence of data operations carried out in the process *DiffSd* to calculate the output of the block Sd (or more precisely, of the virtual block including Diff and Sd).

   What we need to do in this step is to collect the corresponding inputs before the calculations that

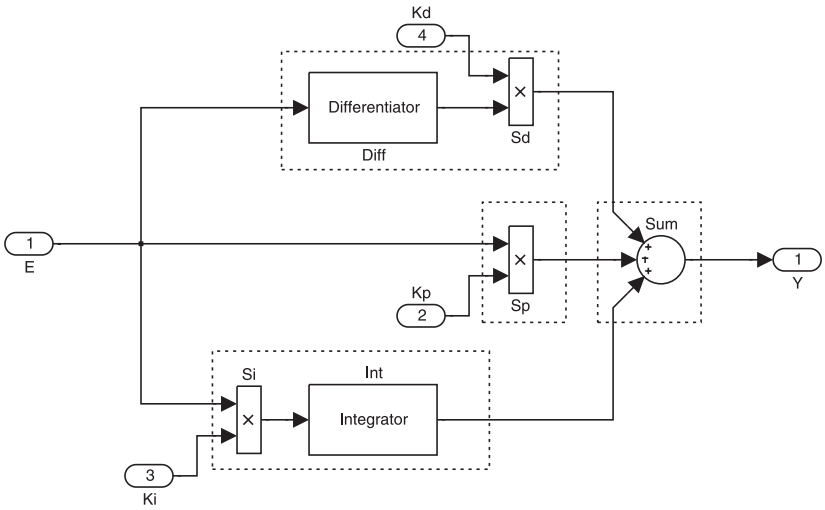


**Fig. 20.** PID controller: virtual blocks that correspond to procedures

use the shared variables. In the main action of *SpSum* shown above, for example, we observe that the input of the shared variables (through $Int\_out$ and $Sd\_out$) are interleaved with those of the system inputs (through $E$ and $Kp$). Since, however, channels corresponding to shared variables are internal, we can split the interleaving to input the value of the shared variables afterwards separately. This is achieved with a version for two inputs of Law (inter-split) below.

**Law[inter-split]**

$$((A_1;\ c \rightarrow A_2) \,[\![\ ns_1 \mid \{\!|\, c \,|\!\} \cup cs \mid ns_2 \,]\!]\, ((A_3 \,[\![\, ns_3 \mid ns_4 \,]\!]\, c \rightarrow A_4);\ A_5)) \setminus \{\!|\, c \,|\!\}$$
$$=$$
$$((A_1;\ c \rightarrow A_2) \,[\![\ ns_1 \mid \{\!|\, c \,|\!\} \cup cs \mid ns_2 \,]\!]\, (A_3 \,;(c \rightarrow A_4);\ A_5)) \setminus \{\!|\, c \,|\!\}$$

**provided** $c \notin usedC(A_1, A_3, A_5);\ usedC(A_2, A_4) = \varnothing;$
$wrtV(A_3) \subseteq ns_3, wrtV(A_4) \subseteq ns_4, usedV(A_4) \cap wrtV(A_3) = \varnothing.$

In this law, we have one internal communication $c$ that occurs only where explicitly shown, as guaranteed by the first two provisos. In the first parallelism, the second parallel action engages in $c$ in interleaving with another action $A_3$. This law states that we can extract $c$ (and its associated action $A_4$) from the interleaving. The point is that the communication $c$ is internal, and we are free to choose the order in which it takes place, as long as that does not block other actions.

Potentially, sequentialising $c \rightarrow A_4$ can hold up interaction between $A_2$ and $A_3$, which is no longer possible, but the second proviso guarantees that $A_2$ is just a data operation. It does not use any channels, and so it does not interact with $A_3$. Another potential problem is the fact that, without the interleaving, $A_4$ can take place only when $A_3$ is finished. Again, the second proviso guarantees that $A_4$ is just a data operation, so that this does not matter: the moment in which $A_4$ takes place is not visible.

We also guarantee that the elimination of the interleaving does not make changes that are originally local to $A_3$ and $A_4$ to become global. In other words, every variable changed by $A_3$ is in the name set $ns_3$, so that none of these changes are masked by the interleaving. Similarly, we require that all variables changed by $A_4$ are in $ns_4$. Finally, without the interleaving, $A_4$ has access to the final value of the variables changed by $A_3$. The last proviso guarantees, however, that $A_4$ does not use any of these variables.

For our example, as already explained, *SpSum* takes the inputs through $Int\_out$ and $Sd\_out$ in interleaving with the inputs through $E$ and $Kp$. The channels $Int\_out$ and $Sd\_out$, however, are internal. So, we can apply a version of Law (inter-split) with two internal communications (in interleaving) to get the result shown in Figure 21. In fact, strictly speaking, to take advantage of Law (inter-split), we need to localise the hiding of the channels $Sd\_out$ and $Int\_out$ in the main action of *SpSum*. As explained in

$$\left(\begin{array}{l} \mu\, X \bullet \left(\begin{array}{l} \left(\begin{array}{l} ((E?x \rightarrow pid\_\_Sp\_In1 := x) \,[\![ \ldots ]\!]\, (Kp?x \rightarrow pid\_\_Sp\_In2 := x))\,; \\ Calculate\_Sp\_out\,; pid\_\_Sum\_In2 := pid\_\_Sp\_Out1 \end{array}\right)\,; \\ \left(\begin{array}{l} ((Sd\_out?x \rightarrow pid\_\_Sum\_In1 := x) \,[\![ \ldots ]\!]\, (Int\_out?x \rightarrow pid\_\_Sum\_In3 := x))\,; \\ Calculate\_Sum\_out; \\ Sum\_out!pid\_\_Sum\_Out1 \rightarrow Skip \end{array}\right)\,; \\ end\_cycle \rightarrow X \end{array}\right) \end{array}\right)$$

**Fig. 21.** Main action of the process *SpSum* - end of phase **BJ**

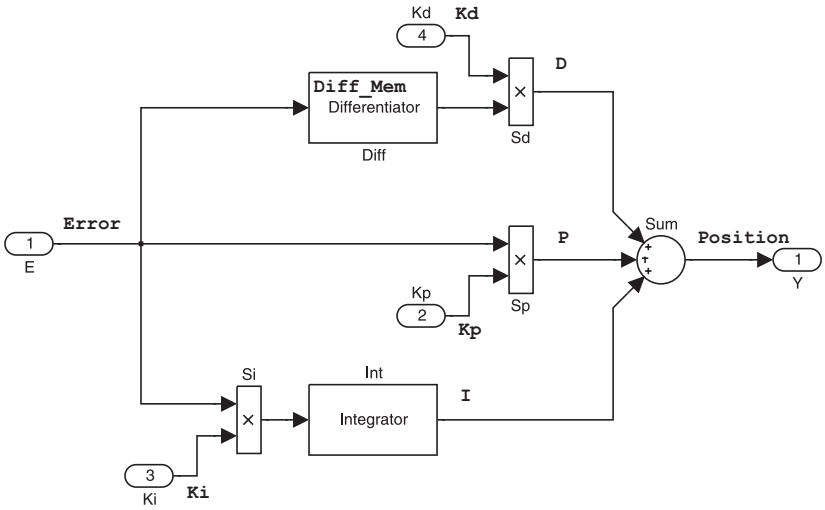


**Fig. 22.** PID controller: correspondence with program variables

Step (5) of Figure 18, we can do this by applying versions of Laws (hid-join) and (hid-par-dist) (for the right number of channels) and the definition of process hiding. After applying Law (inter-split), however, we move the hiding back out, using the same laws (in the opposite direction).

To conclude, at the end of this phase, the process *PID* is as follows.

$$\textbf{process}\ PID \mathrel{\widehat=} \left(\begin{array}{l} SiInt \ \{\!|\, E, Ki, Int\_out, end\_cycle \,|\!\}\ || \\ DiffSd \ \{\!|\, E, Kd, Sd\_out, end\_cycle \,|\!\}\ || \\ SpSum \ \{\!|\, Sd\_out, E, Kp, Int\_out, Y, end\_cycle \,|\!\} \end{array}\right) \setminus \{\!|\, Int\_out, Sd\_out \,|\!\}$$

The only internal channels remaining are *Int_out* and *Sd_out*.

## 6.3. Phase Pr: procedure introduction

The phases **NB**, **BJ**, and **Sc** verify the (parallel) architecture of the implementation. This phase, on the other hand, focusses on the functionality of the procedures. We refine all basic processes (produced in the previous phase) with the objective of using the action models of the Ada procedures to carry out the calculations of outputs and state updates, instead of the schemas of the diagram model.

We use the information about how wires and state in the diagram are matched to the program variables. In our example, we have that the program variables `Diff_Mem`, `I`, `P`, `Kp`, `Error`, `Ki`, `D`, and `Position`, for instance, which are used in the main program `Exec_3`, correspond to the wires of the diagram as shown in Figure 22. In particular, we observe that the input E is called `Error`, the output Y is called `Position`, and `Diff_Mem` corresponds to the state component of Diff, which is the state of its unit delay block.

As explained in detail in Section 4, the variables of the *Circus* model of the diagram correspond to wires and

For each process, we follow the steps below, for each sequence of data operations (schemas and assignments) in its main action.

1. **Introduce a definition for the Ada procedure(s) that implement(s) the operations**. Apply to the process Law (action-intr), using the definition obtained from the program model, and replacing the program variables with the corresponding model variables.
2. **Establish that the data operations are refined by a call to the new action**. This is achieved using ClawZ (as described in [13]).
3. **Remove unused actions**. Apply Law (action-intr) from right to left.

**Fig. 23.** Refinement strategy: phase **Pr**

state components of the diagram. Therefore, with the information that relates the diagram to the program, we get a correspondence also between the variables of the *Circus* model of the diagram and the program variables. In our example, for the process *DiffSd*, we have the following correspondence: `Error` corresponds to $pid\_\_Diff\_In1$, `Kd` to $pid\_\_Sd\_In1$, `Diff_Mem` to $pid\_\_Diff\_\_UnitDelay\_state$, and `D` to $pid\_\_Sd\_Out1$. This correspondence, however, does depend on the particular process being refined. For example, while in *DiffSd* the variable `Error` corresponds to $pid\_\_Diff\_In1$, in the process *SpSum*, it corresponds to $pid\_\_Sp\_In1$.

Moreover, we observe that a variable corresponding to an internal wire may correspond to two model variables in the same process. This occurs if the originally parallel processes that model the blocks connected by the wire have been collapsed. In our example, the program variable `P`, for instance, corresponds to the model variables $pid\_\_Sp\_Out1$ and $pid\_\_Sum\_In2$ of the process *SpSum*.

For processes that correspond to procedures, this issue is handled by the ClawZ toolset (using a tool based on symbolic execution of specifications [1]). For processes that correspond to schedulers, we always associate the program variable with the input variable of the model. In our example, since *SpSum* corresponds to the scheduler in `Exec_1`, we associate `P` with $pid\_\_Sum\_In2$. This reflects the fact that, when joining processes like *Sp* and *Sum* in the previous refinement phase, we reduce their communication to an assignment to the input variable. It is that input variable that is used in the program.

Figure 23 describes the refinement steps to be carried in this phase. We need to consider the main action of all the processes, and, for each them, identify the contiguous sequences of data operations, that is, schemas and assignments. These are the specifications of the procedures that implement block functionality or perform initialisation operations. For the processes that correspond to procedures, we find one or two groups of data operations: the initialisation, and the procedure specification. For example, in the main action of *DiffSd* (see Figure 19), we have the operation *Init* on its own, which is an initialisation operation implemented by the procedure `Init_Derivative`, and the sequence below, which specifies the procedure `Calc_Derivative`.

$$Calculate\_Diff\_out;\ Calculate\_Diff\_State;\ pid\_\_Sd\_In2 := pid\_\_Diff\_Out1;\ pid\_\_Sd$$

For a process that corresponds to a scheduler, we find a group of data operations for each of the procedures that it schedules. The order in which we find them is determined by the order in which the procedures are scheduled in the program. For the *SpSum* process (see Figure 21), for example, we find the group of data operations $Calculate\_Sp\_out;\ pid\_\_Sum\_In2 := pid\_\_Sp\_Out1$ corresponding to the (specification of the) procedure `Calc_Proportion`, and $Calculate\_Sum\_out$, corresponding to `Calc_Output`.

Proceeding with our example, we consider the process *DiffSd* and, more precisely, its specification of `Calc_Derivative` as shown above to explain and illustrate the refinement steps in Figure 23.

1. **Introduce a definition for the Ada procedure(s) that implement(s) the operations**, in terms of the model variables. In the process *DiffSd*, we introduce an action that corresponds to the procedure `Calc_Derivative`, and for that we need to introduce an action corresponding to the procedure `Diff` as well. Since `Diff` does not refer to program variables, its definition is just the action *Diff* in $Exec_3$ (see Figure 15), which is part of the model of the Ada program that we are verifying. For `Calc_Derivative`, the definition is the action call below, which differs from that in $Exec_3$ for *Calc_Derivative* in that it uses the model variables, instead of the program variables `Error`, `Kd`, `Diff_Mem`, and `D`.

$$Diff\,(pid\_\_Diff\_In1, pid\_\_Sd\_In1, pid\_\_Diff\_\_UnitDelay\_state, pid\_\_Sd\_Out1)$$

   The introduction of a new action into a process is trivial, and is justified by Law (action-intr).

$$
\begin{array}{l}
\textbf{process}\ DiffSd \mathrel{\widehat=} \textbf{begin} \\
\quad Diff\_State == [\, pid\_\_Diff\_\_UnitDelay\_state : \mathbb{U};\ pid\_\_Diff\_In1 : \mathbb{U} \,] \\
\quad Sd\_State == [\, pid\_\_Sd\_In1, pid\_\_Sd\_Out1 : \mathbb{U} \,] \\
\quad \textbf{state}\ DiffSd\_State == Diff\_State \land Sd\_State \\
\quad Init\_Derivative \mathrel{\widehat=} pid\_\_Diff\_\_UnitDelay\_state := 0\ e\ 0 \\
\quad Diff \mathrel{\widehat=} \left( \begin{array}{l} \textbf{val}\ Input, K : \mathbb{U};\ \textbf{vres}\ Mem : \mathbb{U};\ \textbf{res}\ Output : \mathbb{U} \bullet \\ \quad Output := K \times (Input - Mem)\,;\, Mem := Input \end{array} \right) \\
\quad Calc\_Derivative \mathrel{\widehat=} Diff\,(pid\_\_Diff\_In1, pid\_\_Sd\_In1, pid\_\_Diff\_\_UnitDelay\_state, pid\_\_Sd\_Out1) \\
\quad \bullet\ Init\_Derivative\,; \\
\quad \left( \mu\, X \bullet \left( \begin{array}{l} \left( \begin{array}{l} E?x \to pid\_\_Diff\_In1 := x \\ \quad [\![ \{\, pid\_\_Diff\_In1 \mid pid\_\_Sd\_In1 \,\} ]\!] \\ Kd?x \to pid\_\_Sd\_In1 := x) \end{array} \right)\,; \\ Calc\_Derivative\,; \\ Sd\_out!pid\_\_Sd\_Out1 \to Skip\,; \\ end\_cycle \to X \end{array} \right) \right) \\
\textbf{end}
\end{array}
$$

**Fig. 24.** Process *DiffSd* - end of phase **Pr**

2. **Establish that the data operations are refined by a call to the new action**. In our example, we prove the refinement below.

$$
\left( \begin{array}{l} Calculate\_Diff\_out; \\ Calculate\_Diff\_State; \\ pid\_\_Sd\_In2 := pid\_\_Diff\_Out1; \\ pid\_\_Sd \end{array} \right) \sqsubseteq Calc\_Derivative
$$

   In this proof, we can use the *Circus* refinement calculus, which includes all the laws of the Z refinement calculus [16], or the existing technique based on ClawZ. In [13], we provide a strategy to automate the use of ProofPower tools based on ClawZ to carry out this step. We reduce the sequence in the specification to a schema, and then to a specification statement. ClawZ can then establish that the body of the procedure, *Diff* in our example, refines it. Finally, we use the copy rule (Law (copy-rule-action)) to introduce the call: in the example, to *Diff*, and afterwards to *Calc_Derivative*.

3. **Remove unused actions**, mostly the ClawZ schemas of the model. This is justified by a reverse application of Law (action-intr).

At the end of this phase, we obtain the process *DiffSd* presented in Figure 24.

## 6.4. Phase Sc: scheduling introduction

In this phase, we already have a process corresponding to each of the schedulers. For our example, we have the processes *SpSum*, which corresponds to the procedures that are scheduled in `Exec_1`, *SiInt* whose corresponding procedure is handled by `Exec_2`, and, finally, *DiffSd* in correspondence with `Exec_3`. In `Exec_0`, we have just the definition of the time frames. We now verify the scheduling order.

The steps in this phase are summarised in Figure 25, and further explained and illustrated below.

1. **Declare** *FrameIndex* **and the channel** *frame*. In this step, we use our knowledge of the number of frames used in the implementation; as said before, this can be extracted from the program (or from its

1. **Declare** *FrameIndex* **and the channel** *frame*. Apply Law (parag-intr).
2. **Introduce the timer model**.
   (a) **Split into a conditional the body of the recursion**, in the main action of each of the processes. Apply a fixed-point law like Law (frame-intr) for the right number of frames. In each frame, call the appropriate procedure under the control of the scheduler. Afterwards, apply Law (main-var-state) to make the frame index a state component.
   (b) **Extract the timer**. Apply a to the diagram process a version of Law (timer-intr) for the right number of frames and schedulers.
   (c) **Extract the assignments to** *cur_f* **to procedures** in the timer model. Apply Law (action-intr) and then Law (copy-rule-action).
3. **Introduce the** `Step` **procedure**. Apply Law (action-intr) and then Law (copy-rule-action) to each process corresponding to a scheduler.

**Fig. 25.** Refinement strategy: phase **Sc**

model). For the PID, there are 2 frames, and therefore the set *FrameIndex* of frame indices is $\{1, 2\}$.

$$FrameIndex == 1 \mathinner{..} 2$$

**channel** *frame* : *FrameIndex*

This step is justified by a program law for introduction of paragraphs: Law (parag-intr).

2. **Introduce the timer model**. The following steps need to be carried out.

(a) **Split into a conditional the body of the recursion** in the main action of all processes. The conditional is used to determine the current frame, and schedule the procedures accordingly. We first use a version of Law (frame-intr) below, which is appropriate for a two-frame implementation; its generalisation to an arbitrary number of frames is simple.

**Law[frame-intr]**

$$(\mu\ X \bullet A_1;\ A_2;\ X)$$
$$=$$
$$\left(\begin{array}{l} \textbf{var}\ cf : FrameIndex \bullet \\ \quad cf := 1; \\ \quad \left(\mu X \bullet \left(\begin{array}{l} \textbf{if}\ (cf = 1) \to A_1 [\!] (cf = 2) \to A_2\ \textbf{fi}; \\ cf := (cf \bmod 2) + 1; \\ X \end{array}\right)\right) \end{array}\right)$$

**provided** *X is not free in* $A_1$ *and* $A_2$*; cf is fresh; and* $\{1, 2\} \subseteq FrameIndex$.

This law splits the sequence of actions $A_1;\ A_2$ in the body of a recursion so that it now takes two iterations of the recursion. The fresh variable *cf* is used to keep track of the iterations; its type *FrameIndex* must include enough values to index the required number of frames. After we apply Law (frame-intr), we use Law (main-var-state) to make *cf* a state component.

The appropriate application of Law (frame-intr) requires the information about how each procedure is allocated to a frame. For *DiffSd* (see Figure 24), we get the main action below, based on the

information that *Calc_Derivative* is scheduled in frame 1 (see Table 1).

$$
\begin{array}{l}
\mathit{Init};\\
\mathit{cur\_f} := 1;\\
\left(\begin{array}{l}
\mu\, X \bullet \left(\begin{array}{l}
\textbf{if } (\mathit{cur\_f} = 1) \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l} E?x \rightarrow \mathit{pid\_\_Diff\_In1} := x \\ \quad [\!|\{\, \mathit{pid\_\_Diff\_In1} \,\}|\!] \\ Kd?x \rightarrow \mathit{pid\_\_Sd\_In1} := x \end{array}\right) ; \\
\mathit{Calc\_Derivative}
\end{array}\right) \\
[\!] \ (\mathit{cur\_f} = 2) \rightarrow \mathit{Sd\_out}!\mathit{pid\_\_Sd\_Out1} \rightarrow \mathit{end\_cycle} \rightarrow \mathit{Skip} \\
\textbf{fi}; \\
\mathit{cur\_f} := (\mathit{cur\_f} \bmod 2) + 1; \\
X
\end{array}\right)
\end{array}\right)
\end{array}
$$

The variable *cur_f* is now a component of the state of *DiffSd*, and the appropriate declaration of *FrameIndex* is guaranteed by Step (2) above.

(b) **Extract the timer**, by applying (to the process that models the diagram) a version of Law (timer-intr) presented below. It considers an implementation with two frames and two schedulers, but the generalisation for an arbitrary number of frames and schedulers is simple. One of the resulting parallel processes should model the timer; in our example, this is the process $\mathit{Exec}_0$ corresponding to `Exec_0`.

Law (timer-intr) applies to processes whose main actions take a specific form (involving a recursion whose body includes a conditional). Since this is a law of processes, we have no need for provisos concerning the access to state components (and local variables), which is partitioned by the processes. This simplifies the law and its application. We need, however, a notation to refer to the main action of a process, so that we can specify the necessary restrictions over it. Accordingly, we use $\mathcal{P}(A)$ to denote any process whose main action is $A$. We, therefore, for example, state below that Law (timer-intr) applies to a parallelism of processes whose main actions are a sequence of an action $A_1$ (or $A_4$), followed by an assignment $\mathit{cf} := 1$, followed by a recursion, whose body is a conditional, followed by the assignment $\mathit{cf} := (\mathit{cf} \bmod 2) + 1$.

**Law[timer-intr]**

$$
\begin{array}{l}
\left(\begin{array}{l}
\mathcal{P}(A_1 \,;\, \mathit{cf} := 1 \,;\, \mu\, X \bullet \textbf{if } \mathit{cf} = 1 \rightarrow A_2 \ [\!] \ \mathit{cf} = 2 \rightarrow A_3;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ \mathit{cf} := (\mathit{cf} \bmod 2) + 1;\ X) \\
\qquad [\!|\, \{\!|\, c \,|\!\} \cup \mathit{cs} \,|\!] \\
\mathcal{P}(A_4 \,;\, \mathit{cf} := 1 \,;\, \mu\, X \bullet \textbf{if } \mathit{cf} = 1 \rightarrow A_5 \ [\!] \ \mathit{cf} = 2 \rightarrow A_6;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ \mathit{cf} := (\mathit{cf} \bmod 2) + 1;\ X)
\end{array}\right) \\
= \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathcal{P}(A_1;\ \mu\, X \bullet f?\mathit{cf} \rightarrow \textbf{if } \mathit{cf} = 1 \rightarrow A_2 \ [\!] \ \mathit{cf} = 2 \rightarrow A_3;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ X) \\
\qquad [\!|\, \{\!|\, c, f \,|\!\} \cup \mathit{cs} \,|\!] \\
\mathcal{P}(A_4;\ \mu\, X \bullet f?\mathit{cf} \rightarrow \textbf{if } \mathit{cf} = 1 \rightarrow A_5 \ [\!] \ \mathit{cf} = 2 \rightarrow A_6;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ X)
\end{array}\right) \\
\qquad [\!|\, \{\!|\, c, f \,|\!\} \,|\!] \\
\mathcal{P}\left(\begin{array}{l} \mathit{cf} := 1; \\ \mu\, X \bullet f!\mathit{cf} \rightarrow \textbf{if } \mathit{cf} = 1 \rightarrow \mathit{Skip} \ [\!] \ \mathit{cf} = 2 \rightarrow c \rightarrow \mathit{Skip}\ \textbf{fi};\ \mathit{cf} := (\mathit{cf} \bmod 2) + 1;\ X \end{array}\right)
\end{array}\right) \setminus \{\!|f|\!\}
\end{array}
$$

**provided** *f is fresh*;
$\mathit{cf} \notin \mathit{wrtV}(A_1, A_2, A_3, A_4, A_5, A_6) \cup \mathit{usedV}(A_1, A_2, A_3, A_4, A_5, A_6)$;
$\mathit{usedC}(A_1, A_4) = \varnothing$; $\mathit{usedC}(A_2) \cap \mathit{usedC}(A_6) = \varnothing$; and $\mathit{usedC}(A_3) \cap \mathit{usedC}(A_5) = \varnothing$.

The purpose of this law is to extract from the main actions of each of the parallel processes the control of the frames, and create a new single process that controls the frames. Therefore, the parallelism of two processes becomes a parallelism of three processes after the application of this law. The original parallel processes synchronise on the set of channels $\{\!|c|\!\} \cup \mathit{cs}$. In the new parallelism, the original processes synchronise on the same channels, but we add a new local channel $f$ that is used to exchange information with the timer process, namely the value of the variable $\mathit{cf}$.

The original parallel processes keep information about the current frame themselves, using a state component *cf* that they each initialise and increment. A proviso requires that *cf* is used and updated only where explicitly shown, so that the actions $A_1$, $A_2$, $A_3$, $A_4$, $A_5$, and $A_6$ are not related to

framing tasks at all. Law (timer-intr) introduces a single process that keeps the framing information, and provides it to the parallel processes, as they need it at the start of each frame.

The actions $A_1$ and $A_4$ are supposed to be initialisations, and are required not to use any channels.

The framing information, that is, the value of *cf* kept by the timer, is shared among all processes; this guarantees that their frames are in lock-step. In the original parallel processes, however, the frames proceed independently in each process. The channel $c$ keeps the sequence of frames of a single cycle in step, but not the frames themselves, which can potentially start and finish independently. The change carried by the Law (timer-intr), therefore, is only possible if there are no communications between the original parallel processes that take place in different frames of their behaviour. For example, the behaviour of the first parallel process in the first frame, as described by action $A_2$, is required to be independent of that of the second parallel process in the second frame, as defined by $A_6$. Similarly, $A_3$ and $A_5$ are required not to share any channels. In this way, we can keep the frames of all processes in synchrony, without introducing a deadlock.

With the application of this law, or more precisely, of its version for three schedulers, to our example, we get the following definition for the process *PID*.

$$
\begin{array}{l}
\textbf{process}\ PID \mathrel{\widehat=} \\
\left(\begin{array}{l}
\left(\begin{array}{l}
SiInt \ \{\!|\ E, Ki, Int\_out, end\_cycle, frame\ |\!\} \\
\quad || \\
DiffSd \ \{\!|\ E, Kd, Sd\_out, end\_cycle, frame\ |\!\} \\
\quad || \\
SpSum \ \{\!|\ Sd\_out, E, Kp, Int\_out, Y, end\_cycle, frame\ |\!\}
\end{array}\right) \\
\quad [\![ \{\!|\ end\_cycle, frame\ |\!\} ]\!] \\
Exec_0
\end{array}\right) \setminus \{\!|\ frame, Int\_out, Sd\_out\ |\!\}
\end{array}
$$

The main action of the process *DiffSd*, for example, is now as follows.

$$
\begin{array}{l}
Init\_Derivative; \\
\left(\begin{array}{l}
\mu\, X \bullet \left(\begin{array}{l}
frame?cur\_f \rightarrow \\
\left(\begin{array}{l}
\textbf{if}\ (cur\_f = 1) \rightarrow \\
\quad \left(\begin{array}{l}
\left(\begin{array}{l}
E?x \rightarrow pid\_\_Diff\_In1 := x \\
\quad [\![ \{\ pid\_\_Diff\_In1\ \} \mid \{\ pid\_\_Sd\_In1\ \} ]\!]
\end{array}\right); \\
Kd?x \rightarrow pid\_\_Sd\_In1 := x
\end{array}\right) \\
\quad Calc\_Derivative \\
[\!]\ (cur\_f = 2) \rightarrow Sd\_out!pid\_\_Sd\_Out1 \rightarrow end\_cycle \rightarrow Skip \\
\textbf{fi}; \\
X
\end{array}\right)
\end{array}\right)
\end{array}\right)
\end{array}
$$

The process $Exec_0$, on the other hand, is as follows at this stage.

$$
\begin{array}{l}
\textbf{process}\ Exec_0 \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ [cur\_f : FrameIndex] \\
\quad \bullet\ cur\_f := 1\,; \\
\quad \left(\begin{array}{l}
\mu\, X \bullet frame!cur\_f \rightarrow \left(\begin{array}{l}
\textbf{if}(cur\_f = 1) \rightarrow Skip \\
[\!]\ (cur\_f = 2) \rightarrow end\_cycle \rightarrow Skip \\
\textbf{fi}\,; \\
cur\_f := (cur\_f \bmod 2) + 1\,; \\
X
\end{array}\right)
\end{array}\right) \\
\textbf{end}
\end{array}
$$

The only difference between this process and the model of our timer in Figure 14 is the use of procedures to initialise and update $cur\_f$. The next step sorts this out.

(c) **Extract the assignments to $cur\_f$ to procedures**. This is achieved by applying, to the process that models the timer, Law (action-intr) to introduce the model of the procedures that initialise and increment the frame counter. In our example, they are the actions $Init\_F$ and $Next\_F$. Afterwards, we use Law (copy-rule-action) to replace the uses of these procedures with calls to them. After this step, the process $Exec_0$ introduced in the previous step is exactly as shown in Figure 14.

3. **Introduce the `Step` procedure**. In each of the processes that model a scheduler, it remains for us to

$$
\begin{array}{l}
\textbf{process}\ DiffSd \mathrel{\widehat=} \textbf{begin} \\
\quad Diff\_State == [\, pid\_\_Diff\_\_UnitDelay\_state : \mathbb{U};\ pid\_\_Diff\_In1 : \mathbb{U} \,] \\
\quad Sd\_State == [\, pid\_\_Sd\_In1, pid\_\_Sd\_Out1 : \mathbb{U} \,] \\
\quad \textbf{state}\ DiffSd\_State == Diff\_State \land Sd\_State \\
\quad Init\_Derivative \mathrel{\widehat=} pid\_\_Diff\_\_UnitDelay\_state := 0\ e\ 0 \\
\quad Diff \mathrel{\widehat=} \left( \begin{array}{l} \textbf{val}\ Input, K : \mathbb{U};\ \textbf{vres}\ Mem : \mathbb{U};\ \textbf{res}\ Output : \mathbb{U} \bullet \\ \quad Output := K \times (Input - Mem)\,;\, Mem := Input \end{array} \right) \\
\quad Calc\_Derivative \mathrel{\widehat=} Diff(pid\_\_Diff\_In1, pid\_\_Sd\_In1, pid\_\_Diff\_\_UnitDelay\_state, pid\_\_Sd\_Out1) \\
\quad Step \mathrel{\widehat=} \left( \begin{array}{l} frame?cur\_f \rightarrow \\ \left( \begin{array}{l} \textbf{if}\ (cur\_f = 1) \rightarrow \\ \quad \left( \begin{array}{l} \left( \begin{array}{l} E?x \rightarrow pid\_\_Diff\_In1 := x \\ \quad [\!|\{\, pid\_\_Diff\_In1 \,\} \mid \{\, pid\_\_Sd\_In1 \,\}|\!] \\ Kd?x \rightarrow pid\_\_Sd\_In1 := x \end{array} \right)\,; \\ Calc\_Derivative \end{array} \right) \\ [\!]\ (cur\_f = 2) \rightarrow Sd\_out!pid\_\_Sd\_Out1 \rightarrow end\_cycle \rightarrow Skip \\ \textbf{fi} \end{array} \right) \end{array} \right) \\
\quad \bullet\ Init\_Derivative\,;\,(\mu\, X \bullet Step\,;\, X) \\
\textbf{end}
\end{array}
$$

**Fig. 26.** Process *DiffSd* - end of phase Sc

introduce the model of the `Step` procedure. Each process already has in its main action the body of its corresponding `Step` procedure. All that we need to do is to use Law (action-intr) to introduce an action *Step* that models the procedure, and afterwards Law (copy-rule-action) to introduce a call. At the end, of this step, the *DiffSd* process, for example, is as shown in Figure 26

If we compare the process in Figure 26 with that in Figure 15, which is the model of the main Ada program Exec_3, we observe that the only differences are in the names of state components, local input variables, and hidden channels, and in the use of schema calculus to define the state. These are purely syntactic differences that can, for instance, be checked automatically to be indeed the only discrepancies.

Alternatively, from the point of view of program transformation, first the definition of schema conjunction can be used to flatten the state definitions of the scheduler processes. Renaming the model variables to the program variables is a simple functional data refinement, whose retrieve relation is the conjunction of the equalities that relate the variables. Distributivity of data refinement for *Circus* is shown in [15]. For *DiffSd*, for instance, the retrieve relation equates $pid\_\_Diff\_\_UnitDelay\_state$ to `Diff_Mem`, $pid\_\_Diff\_In1$ to `Error`, $pid\_\_Sd\_In1$ to `Kd`, and $pid\_\_Sd\_Out1$ to `D`. This is based on the correspondence between model and program variables that has been already used previously. Renaming of local variables, including those introduced by input communications, is justified by standard refinement laws. Finally, internal channels, which are used to communicate shared data, can be renamed using Law (hid-ren) because they are hidden. In the example, we can rename *Sd_out* to *Dsh* and *Int_out* to *Ish*. Again, this correspondence is already established.

In summary, all this indicates that we can either carry out the additional steps above to obtain models that are in exact correspondence with the model of the program, or just check that the only differences between the models are of this nature. If the transformations are carried out, the *Circus* program obtained is the model of our implementation, except only for the names of processes. For our example, we have seen that the refinement creates processes named *SpSum*, *SiInt*, and *DiffSd*, instead of $Exec_1$, $Exec_2$, and $Exec_3$. These differences, however, have no impact on the semantics of the processes [46].

Many of the steps of our refinement strategy are trivial, but as we explained some rely on more elaborate and specialised laws. What they have in common is that they are all based on *Circus* refinement laws. Consequently, we have is a sound verification procedure that can be automated.

Compositionality stems from the use of refinement. If, for instance, the PID is used in a larger diagram,

and its implementation is, therefore, part of a larger program, all the verification steps for the PID are exactly as shown above, except only for those in phase **Sc**. It is just that last phase that is specific to the structure of the closed program, which must correspond to that of a complete diagram. Additionally, if we consider an alternative implementation for the PID, which uses the same Ada procedures, but schedules them in a different way, the only verification effort affected is the last part of phase BJ, which joins the processes that model procedures scheduled together. Furthermore, and most importantly, all the steps of our refinement strategy can be carried out automatically, with the proof effort completely concentrated in phase Pr. This phase is only ever affected if different procedures or different procedure implementations are considered. What we have achieved is a strategy that can build on the ClawZ automation infrastructure to cover the verification of complete programs, without adding to the proof effort required.

## 7. Related work

The work that we have presented in this paper is distinctive in its aim to verify implementations of (discrete-time) Simulink diagrams, rather than validate properties of the diagrams or of the systems that they specify. The literature, however, provides a wealthy of approaches for modelling and analysis of Simulink diagrams and other similar notations; several of them rely on a later use of an automatic code generator. In this section, we discuss some of these works. As already explained, automatically generated code may not be appropriate for specialised hardware, and, in addition, for safety-critical systems use of a code generator is often not regarded as enough assurance of correctness of an implementation. In this case, validation of the models and designs needs to be followed up by a verification of their implementations.

The need to verify code that is automatically generated is also recognised in the work in [6]. It provides a technique for mechanising the construction of safety cases using assertion-based verification of properties identified in the requirements. The verification tool used is AutoCert. The case study is code generated from Simulink diagrams. To provide independent arguments, diagrams are not used to identify the properties like we do here. These properties, however, are formalised in terms of signals of the diagram, and a mapping between them and variables of the program is necessary, like in our work. The results in [6] indicate how a formal verification can be used for certification as part of a safety case, and in that way they go beyond what we achieve here: we are only concerned with the verification itself. On the other hand, there is no mention of concurrency and scheduling in [6]. For the verification of sequential code, it seems that we could equally use ClawZ or AutoCert, since our technique identifies the properties of interest for each procedure.

For analysis of discrete transition systems, abstraction and model checking have been effective [47]. Model checking replaces simulation with an exhaustive analysis, but restricts the data types that can be handled.

For hybrid systems, diagrams with discrete and continuous blocks, and automata with continuous dynamics associated with discrete states are used [36]. Tools are available to model and analyse such diagrams [37]. Abstraction makes the models tractable; often they are discretised [3, 53, 54]. This approach is used in [52] for Simulink specifications of hybrid systems: automata models are used to analyse the diagrams.

Simulations of a Simulink diagram are used in the construction of a formal model for an electronic throttle controller in [20]. The model is a hybrid automata, and is analysed using the model checker CheckMate. In the work reported in [33], a Simulink model of a wheel brake system described in a standards document (ARP 4761), and a corresponding faulty Simulink model are imported to SCADE for model checking. SCADE makes assumptions about the implementation environment, but works very well for systems that follow these restrictions. SCADE provides a code generator that has been developed to satisfy stringent quality requirements of the avionics industry, although it has not been formally verified. Another example is tackled with SVM, which is used in [19] to model check a triplex sensor voter specified in Simulink.

The combined use of UML and Simulink is supported by the approach in [26], which is used for hybrid mechatronic systems. This work presents a technique to verify real-time properties of a distributed design compositionally using model checking. It is also part of the trend to verify models and designs, and rely on code generators for the automatic production of programs [27].

Model checking is also considered in [2], which presents a management tool for model-based design of embedded systems, from requirements to code, integrated with Matlab. The tool records the verification activities, including model checking, their results, and their associations to requirements.

An extension of Simulink to specify real-time interactions is used in [35] to model a helicopter control system. The approach is based on a programming language called Giotto. The extended model is translated

to Simulink, and then to a program that combines the result of the Simulink code generator with a Giotto program that handles the scheduling. This program runs in an embedded machine that is platform dependent.

Analysis across boundaries of different models is tackled in [32]. This work uses an intermediate notation, SPI, to combine models written in different languages. It is based on communicating processes, but does not incorporate data operations; the focus is on timing requirements. The translation of Simulink diagrams to SPI defines a timing model for Simulink. Code generators handle the data aspects of the input models.

There have been efforts to use logic to capture the meaning of control diagrams and support reasoning. Weakest preconditions are used in [7]; preconditions and postconditions are predicates over elements of traces of values of variables over the cycles. Concurrency is not handled, but is pointed out as future work. The work in [10] proposes a compositional technique in the style of a Hoare logic to reason about properties of the frequency response of continuous-time control diagrams. The feasibility and practical relevance of their approach is further detailed in [9]. In [38], Mahony used Isabelle/HOL tools to mechanise an assertion technique based on predicate transformers for dataflow networks with feedback. This is a graphical notation like control law diagrams; however, parallelism is indicated explicitly, and in [38] nodes are dynamic processes.

Industrial examples of control software have guided the work in [28], where models for sequential function charts directly related to their shape are proposed. Reasoning tools are indicated as future work.

Functional and timing requirements of Simulink diagrams are modelled in [18] using a Timed Interval Calculus (TIC). The technique is based on a translation of Simulink diagrams to TIC specifications, which use the Z mathematical and schema notations to structure interval time properties. Support for automated proof of properties of the TIC specifications using PVS is also provided.

Refinement is also the basis of the work in [8], where Simulink is extended with a specification block to allow an action-system style of formal stepwise model development. The focus is refinement of models, rather than refinement to code. Specification blocks are used to define preconditions and postconditions for diagram fragments yet to be developed. In this way, it is possible to reason about diagrams in terms of abstract specifications of sub-diagrams. We, on the other hand, generate specifications from existing diagrams, and give a path to establish refinement by code.

*Circus* has been applied for the refinement and verification of several industrial and sizeable applications [44, 25, 23]. It has a refinement theory and technique [15] that allows the development of distributed and concurrent applications from centralised specifications, composed of a single process. The technique presented here is different; it starts from a highly distributed model of a diagram, and reduces parallelism to match that of the program. Some of the novel laws that we propose, however, are of general interest, and complement those already available [42] to formalise the refinement strategy that we propose. Like for those in [42], the soundness of the new laws is based on the UTP model of *Circus*. Moreover, the very large *Circus* models that we generate using the semantics presented here are a valuable source of validation for *Circus* tools. This work makes the applicability of *Circus* to an important class of industrial applications a reality.

## 8. Conclusions and future work

We have presented a semantics for discrete-time Simulink diagrams using a combination of Z and CSP: *Circus*. Our model captures the functionality of a diagram over any number of cycles, and the inherent parallelism between blocks. We can handle enabled subsystems, blocks whose outputs depend on the order of arrival of the inputs, and independent flows of execution inside blocks. Feedback loops are also covered, by catering for blocks that do not require all the inputs before producing the outputs.

There are several combinations of a state-based formalism with a process algebra [21, 55, 39, 22, 31]; *Circus* is distinctive in its refinement theory. Our semantics opens the possibility of reasoning about diagrams and proving the correctness of implementations using refinement. We have presented a refinement strategy that can be used to verify parallel Ada programs that implement Simulink diagrams. Each step of the refinement strategy is justified by *Circus* refinement laws. They guarantee the soundness of the verification, and provide a basis for automation. The use of *Circus* and refinement puts us in a position to handle a comprehensive diagrammatic notation, large data sets, and dynamic scheduling.

If a law is not applicable, because a syntactic constraint or a proviso is not satisfied, we have an indication that there is a mistake in the implementation, or in the analysis that matches the diagram and program components. The graph model of a Simulink diagram, and the associated *Circus* model, can be automatically generated [57]; the same is also possible for the *Circus* model of the Ada implementation. The specification of the correspondence between components of the diagram and of the of the program, however, is not

fully automated. ClawZ provides support for the definition of the association between wires and program variables, including the automatic generation of a suggested mapping, but the process is interactive. The theorem-proving effort is only in phase **Pr**, where current practice supported by ClawZ can be adopted and high levels of automation can be achieved [13, 1]. Automation here ensures practicality, and makes it possible to keep the formalism mostly hidden from engineers and programmers.

Our example is small, but illustrates how the issues related to the architecture of the implementation and reuse of ClawZ are addressed. Its implementation is representative of the use of time frames and shared variables. We have considered a number of industrial examples, including applications provided by two aircraft manufacturers. The most complex *Circus* model is for a diagram whose structure includes up to four nesting levels, with 155 elementary blocks and 14 subsystems. A large QinetiQ case study is a Non-linear Dynamic Inversion controller; in that example, we have a three-processor implementation, with three frames and four shared variables. There are over 1500 lines of code; the *Circus* model has 200 pages. Additional tool development is necessary before we can carry out larger case studies; this work is under way.

As a next step, we will consider the automatic generation of *Circus* models also for Ada programs. The refinement strategy can be formally described using tactics [43], and that is the basis for its automation using ProofPower-Z. In summary, we are working on a toolset to automate the application of our technique; it will be a powerful resource in the analysis of control diagrams and their implementations.

With full automation, and consequent possibility of carrying out a wider collection of case studies, we are going to be in a position to consider the issue of error management. It will be interesting to determine how failure in the refinement can be conveyed in a way that helps identification of the source of the problem. At the moment, failure in the **Pr** phase displays a simplified version of the unproved verification conditions. The prototype of the *Circus* model checker [24] combines refinement checking and theorem-proving techniques. It may provide a route for effective error reporting and even further automation of the phase **Pr**.

The refinement strategy is very general and modular. The first three phases handle the models of blocks; they match the structure of the specification to that of the implementation, and prove the correctness of the individual procedures. These phases are very stable and widely applicable. The fourth phase is dependent on the architecture of the scheduler, and on the scheduling policy. In our experience, what we have presented here is enough to cope with applications in the area of military avionics.

In the next phase of our work, we will seek examples in other areas of application; we are now considering civil avionics. IMA applications, in particular, pose an interesting challenge, since their modular architecture provides an opportunity for reuse of *Circus* models and their formal verification. Moreover, with *Circus*, advanced dynamic scheduling policies can be covered.

One of the challenges in considering other areas of application is the programming language used in the implementations. Adapting our technique to subsets of C, like MISRA C and C flat, is not difficult; a version of ClawZ for such a subset is under development. We observe, however, that to consider other programming languages, or even paradigms, all we need is a *Circus* semantics. For functional languages like HUME [29], for example, the CSP subset of *Circus* is likely to be enough to model programs, since CSP is itself a functional language. In this case, we need technology to refine state-based models to functional programs.

One aspect of the verification that is not covered here is timing. We observe, however, that *Circus* does have a conservative timed extension, *Circus Time*, whose semantics preserves the laws of untimed *Circus*. With its use, we will tackle multi-rate diagrams, generate more direct models of the Ada programs, and provide an extended technique for verification of timing, as well as functional and scheduling, properties.

Finally, a Simulink model can include stateflow blocks; they are defined by a diagram including data and finite state machines that react to events in the Simulink model. The reactions lead to state changes that affect the behaviour of the Simulink model. Stateflow diagrams are studied in [52, 51]. We are investigating the use of *Circus* to model stateflow diagrams [12]; it seems promising as *Circus* can cope with 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 and the EPSRC. We discussed various aspects of it with Daniel Boulton, Chris Marriott, Marcel Oliveira, Jim Woodcock, and Frank Zeyda; their comments were very useful. We are also grateful to anonymous referees for very helpful suggestions.

## References


[1] M. M. Adams, P. B. Clayton, Cost-Effective Formal Verification for Control Systems, in: K. Lau, R. Banach (eds.), ICFEM 2005: Formal Methods and Software Engineering, vol. 3785 of Lecture Notes in Computer Science, Springer-Verlag, 2005.

[2] B. Aldrich, A. Fehnker, P. H. Feiler, Z. Han, B. H. Krogh, K. Lim, S. Sivashankar, Managing Verification Activities using SVM, in: J. Davies, W. Schultte, M. Barnett (eds.), 6th International Conference on Formal Engineering Methods, vol. 3308 of Lecture Notes in Computer Science, Springer-Verlag, 2004.

[3] R. Alur, T. A. Henzinger, G. Lafeerriere, G. J. Pappas, Discrete abstractions of hybrid systems, Proceedings of the IEEE 88 (2) (2000) 971 – 984.

[4] R. Arthan, P. Caseley, C. M. O'Halloran, A. Smith, ClawZ: Control laws in Z, in: 3rd International Conference on Formal Engineering Methods, IEEE Press, 2000.

[5] J. Barnes, Programming in Ada 95, Addison-Wesley, 2005.

[6] N. Basir, E. Denney, B. Fischer, Deriving Safety Cases for Hierarchical Structure in Model-based Development, in: Computer Safety, Reliability, and Security, vol. 6351 of Lecture Notes in Computer Science, Springer-Verlag, 2010.

[7] J. Blow, A. Galloway, Generalised Substitution Language and Differentials, in: D. Bert, J. P. Bowen, M. C. Henson, K. Robinson (eds.), ZB 2002: Formal Specification and Development in Z and B, vol. 2272 of Lecture Notes in Computer Science, Springer-Verlag, 2002.

[8] P. Boström, L. Morel, M. Waldén, Stepwise development of Simulink models using the refinement calculus framework, in: J. C. P. Woodcock, C. B. Jones, Z. Liu (eds.), International Colloquium on Theoretical Aspects of Computing, vol. 4711 of Lecture Notes in Computer Science, Springer-Verlag, 2007.

[9] R. J. Boulton, H. Gottliebsen, R. Hardy, T. Kelsy, U. Martin, Design Verification for Control Engineering, in: E. A. Boiten, J. Derrick, G. Smith (eds.), IFM 2004: Integrated Formal Methods, vol. 2999 of Lecture Notes in Computer Science, Springer-Verlag, 2004, invited paper.

[10] R. J. Boulton, R. Hardy, U. Martin, A Hoare-Logic for Single-Input Single-Output Continuous-Time Control Systems, in: 6th International Workshop on Hybrid Systems: Computation and Control, vol. 2623 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

[11] P. Caspi, A. Curic, A. Maignan, C. Sofronis, S. Tripakis, Translating Discrete-Time Simulink to Lustre, in: R. Alur, I. Lee (eds.), EMSOFT 2003, vol. 2855 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

[12] A. L. C. Cavalcanti, Stateflow diagrams in *Circus*, in: P. Machado (ed.), SBMF 2008: Brazilian Symposium on Formal Methods, Electronic Notes in Theoretical Computer Science, Elsevier B. V., 2008, invited paper.

[13] A. L. C. Cavalcanti, P. Clayton, Verification of Control Systems using *Circus*, in: 11th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, 2006.

[14] A. L. C. Cavalcanti, P. Clayton, C. O'Halloran, Control Law Diagrams in *Circus*, in: J. Fitzgerald, I. J. Hayes, A. Tarlecki (eds.), FM 2005: Formal Methods, vol. 3582 of Lecture Notes in Computer Science, Springer-Verlag, 2005.

[15] A. L. C. Cavalcanti, A. C. A. Sampaio, J. C. P. Woodcock, A Refinement Strategy for *Circus*, Formal Aspects of Computing 15 (2 - 3) (2003) 146 — 181.

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

[17] C. Chen, J. S. Dong, Applying Timed Interval Calculus to Simulink Diagrams, in: Z. Liu, H. Jifeng (eds.), International Conference on Formal Engineering Methods, Lecture Notes in Computer Science, Springer-Verlag, 2006.

[18] C. Chen, J. S. Dong, J. Sun, A formal framework for modeling and validating simulink diagrams, Formal Aspects of Computing 21 (5) (2009) 451 – 484.

[19] S. Dajani-Brown, D. Cofer, G. Hartmann, S. Pratt, Formal Modeling and Analysis of an Avionics Triplex Sensor Voter, in: T. Ball, S. K. Rajamani (eds.), SPIN 2003, vol. 2648 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

[20] A. Fehnker, B. H. Krogh, Hybrid System Verification is not a Sinecure: Electronic Throttle Control Case Study, in: F. Wang (ed.), ATVA 2004, vol. 3299 of Lecture Notes in Computer Science, Springer-Verlag, 2004.

[21] C. Fischer, How to Combine Z with a Process Algebra, in: J. Bowen, A. Fett, M. Hinchey (eds.), ZUM'98: The Z Formal Specification Notation, Springer-Verlag, 1998.

[22] C. Fischer, Combination and Implementation of Processes and Data: from CSP-OZ to Java, Ph.D. thesis, Fachbereich Informatik Universität Oldenburg (2000).

[23] A. F. Freitas, A. L. C. Cavalcanti, Automatic Translation from *Circus* to Java, in: J. Misra, T. Nipkow, E. Sekerinski (eds.), FM 2006: Formal Methods, vol. 4085 of Lecture Notes in Computer Science, Springer-Verlag, 2006.

[24] L. J. S. Freitas, Model Checking *Circus*, Ph.D. thesis, University of York, Department of Computer Science (2006).

[25] L. J. S. Freitas, A. L. C. Cavalcanti, J. C. P. Woodcock, Taking our own medicine: applying the refinement calculus to state-rich refinement model checking, in: Z. Liu, J. He (eds.), Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006, vol. 4260 of Lecture Notes in Computer Science, Springer-Verlag, 2006.

[26] H. Giese, M. Hirsch, Modular Verification of Safe Online-Reconfiguration for Proactive Components in Mechatronic UML, in: J.-M. Bruel (ed.), Satellite Events at the MoDELS 2005 Conference, vol. 1618 of Lecture Notes in Computer Science, Springer-Verlag, 2006.

[27] S. Graf, O. Haugen, I. Ober, B. Selic, Modelling and Analysis of Real-time and Embedded Systems, in: J.-M. Bruel (ed.), Satellite Events at the MoDELS 2005 Conference, vol. 1618 of Lecture Notes in Computer Science, Springer-Verlag, 2006.

[28] C. Gurr, K. Tourlas, Towards the Principled Design of Software Engineering Diagrams, in: 22nd International Conference on Software Engineering, ACM Press, 2000.

[29] K. Hammond, G. Michaelson, Hume: A domain-specific language for real-time embedded systems, in: Generative Programming and Component Engineering, vol. 2830 of Lecture Notes in Computer Science, Springer-Verlag, 2003.

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

[31] J. Hoenick, E.-R. Olderog, Combining specification techniques for processes, data and time, in: M. J. Butler, L. Petre, K. Sere (eds.), Integrated Formal Methods, vol. 2335 of Lecture Notes in Computer Science, 2002.

[32] M. Jersak, D. Ziegenbein, F. Wolf, K. Richter, R. Ernst, F. Cieslok, J. Teich, K. Strehl, L. Thiele, Embedded System Design using the SPI Workbench, in: 3rd International Forum on Design Languages, 2000.

[33] A. Joshi, M. P. E. Heimdahl, Model-Based Safety Analysis of Simulink Models using SCADE Design Verifier, in: R. Winther, B. a. Gran, G. Dahll (eds.), SAFECOMP 2005, vol. 3688 of Lecture Notes in Computer Science, Springer-Verlag, 2005.

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

[35] C. M. Kirsch, M. A. A. Sanvido, A Giotto-Based Helicopter Control System, in: A. Sangiovanni-Vincentelli, J. Sifakis (eds.), EMSOFT 2002, vol. 2491 of Lecture Notes in Computer Science, Springer-Verlag, 2002.

[36] B. H. Krogh, Approximating Hybrid System Dynamics for Analysis and Control, in: F. W. Vaandrager, J. H. van Schuppen (eds.), Hybrid Systems: Computation and Control: Second International Workshop, vol. 1569 of Lecture Notes in Computer Science, Springer-Verlag, 1999.

[37] B. H. Krogh, Recent Developments in Modeling and Analysis of Hybrid Dynamic Systems, in: S. Donatelli, J. Kleijn (eds.), Applications and Theory of Petri Nets 1999: 20th International Conference, vol. 1639 of Lecture Notes in Computer Science, Springer-Verlag, 1999.

[38] B. Mahony, 1st International Workshop on Formalising Continuous Mathematics, in: The DOVE Approach to the Design of Complex Dynamic Processes, 2002.

[39] B. Mahony, J. S. Dong, Timed Communicating Object Z, IEEE Transactions on Software Engineering 26 (2) (2000) 150 – 177.

[40] The MathWorks,Inc., Simulink, `http: //www.mathworks.com/products/simulink`.

[41] C. C. Morgan, Programming from Specifications, 2nd ed., Prentice-Hall, 1994.

[42] M. V. M. Oliveira, Formal Derivation of State-Rich Reactive Programs Using *Circus*, Ph.D. thesis, University of York (2006).

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

[44] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock, Formal development of industrial-scale systems, Innovations in Systems and Software Engineering 1 (2) (2005) 126 – 147.

[45] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock, Unifying Theories in ProofPowerZ, Formal Aspects of Computing, online firstDOI 10.1007/s00165-007-0044-5.

[46] M. V. M. Oliveira, A. L. C. Cavalcanti, J. C. P. Woodcock, A UTP Semantics for *Circus*, Formal Aspects of Computing 21 (1-2) (2009) 3 – 32.

[47] S. Ranville, P. E. Black, Automated Testing Requirements — Automotive Perspective, in: 2nd International Workshop on Automated Program Analysis, Testing and Verification, 2001.

[48] A. Sherif, A Framework for Specification and Validation of Real-time Systems using *Circus* Actions, Ph.D. thesis, Centro de Informática/UFPE, Brazil (2006).

[49] A. Sherif, A. L. C. Cavalcanti, H. Jifeng, A. C. A. Sampaio, A process algebraic framework for specification and validation of real-time systems, Formal Aspects of Computing 22 (2) (2010) 153 – 191.

[50] A. Sherif, H. Jifeng, A. L. C. Cavalcanti, A. C. A. Sampaio, A framework for specification and validation of real-time systems using *circus* actions, in: Z. Liu, K. Araki (eds.), International Colloquium on Theoretical Aspects of Computing, vol. 3407 of Lecture Notes in Computer Science, Springer-Verlag, 2005.

[51] C. Spencer, Model Checking for Stateflow Diagram with Floating Point Variables and Complex Expressions, Master's thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University (2002).

[52] A. Tiwari, Formal Semantics and Analysis Methods for Simulink Stateflow Models, Tech. rep., SRI International, `http://www.csl.sri.com/~tiwari/stateflow.html` (2002).

[53] A. Tiwari, G. Khanna, Series of Abstractions for Hybrid Automata, in: F. W. Vaandrager, J. H. van Schuppen (eds.), Hybrid Systems: Computation and Control: Second International Workshop, vol. 2289 of Lecture Notes in Computer Science, Springer-Verlag, 2002.

[54] A. Tiwari, N. Shankar, J. Rushby, Invisible formal methods for embedded control systems, Proceedings of the IEEE 91 (1) (2003) 29 – 39.

[55] H. Treharne, S. Schneider, Using a process algebra to control B OPERATIONS, in: 1st International Conference on Integrated Formal Methods – IFM'99, Springer-Verlag, 1999.

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

[57] F. Zeyda, A. L. C. Cavalcanti, Mechanised Translation of Control Law Diagrams into *Circus*, in: Integrated Formal Methods, Lecture Notes in Computer Science, Springer-Verlag, 2009.

## A. Formalisation of the graph model

Here, we use Z to characterise the form of the data-flow graph defined by the function DF, and used in our formalisation of the construction of the *Circus* model of a diagram. We use given sets to represent the valid specification names, and the sets of signal and block names.

$$[\, NAME, Signal, Block \,]$$

For a given diagram $d$, the graph defined by DF is represented in a record that gives the name of the diagram, its inputs and outputs, and a characterisation of each of its blocks.

*Graph*

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

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.

$$Enabled ::= always \mid esigs\langle\langle \mathbb{P}\, Signal \rangle\rangle$$

Moreover, in a flow, the order in which the signals are received may be relevant. Finally, we also need to know the signals that a flow requires (*rinps*) in order to produce its outputs.

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

Besides including information about flows of execution, the block wiring defines an order for the inputs (*inps*) and outputs (*outs*) of the block. Each flow is characterised by the set of outputs that it produces. Therefore, in the record of a block wiring, *flows* is a function that associates each set of outputs that characterises a flow to the corresponding information about a flow: a record of type *Flow*.

*BlockWiring*

$$\begin{array}{l} inps, outs : \mathrm{seq}\, Signal \\ flows : \mathbb{P}\, Signal \mathrel{+\!\!\!\rightarrow} Flow \\ \hline \forall\, pouts : \mathrm{dom}\, flows \mid flows(pouts).enabled \in \mathrm{ran}\, esigs \bullet (esigs^{\sim})\,(flows(pouts).enabled) \subseteq \mathrm{ran}\, inps \\ \forall\, pouts : \mathrm{dom}\, flows \bullet flows(pouts).rinps \subseteq \mathrm{ran}\, inps \\ \bigcup(\mathrm{dom}\, flows) = \mathrm{ran}\, outs \\ \forall\, pouts_1, pouts_2 : \mathrm{dom}\, flows \bullet pouts_1 \neq pouts_2 \Rightarrow pouts_1 \cap pouts_2 = \varnothing \end{array}$$

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.

The invariant above is certainly not strong enough to characterise the set of graphs that correspond to well-formed Simulink diagrams. Instead, we provide the invariants that clarify the way in which we use the model proposed to capture properties of (independent) flows of execution.

## B. Refinement laws

We present here the laws used in the application of our verification technique to the PID example. Some of these laws can be found in [42]; they are marked with a *. There is an implicit assumption that there are no nested redeclarations of variables. This is a usual assumption in semantic definitions, which simplifies the application of some laws. It needs, however, to be enforced by a syntactic check and renaming, if necessary.

## B.1. Parallelism

**Law[par-inter]** $A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 = A_1 \,[\![ ns_1 \mid ns_2 ]\!]\, A_2$
**provided** $(usedC(A_1) \cup usedC(A_2)) \cap cs = \varnothing$

**Law[*par-seq-step]** $(A_1;\ A_2) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_3 = A_1;\ (A_2 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_3)$
**provided** $usedC(A_1) = \varnothing$; $usedV(A_3) \cap wrtV(A_1) = \varnothing$; and $wrtV(A_1) \subseteq ns_1 \cup ns_1'$.

## B.2. Interleaving

**Law[*inter-unit]** $Skip \,[\![ ns_1 \mid ns_2 ]\!]\, A = A \,[\![ ns_2 \mid ns_1 ]\!]\, Skip = A$

**Law[*inter-assoc]** $(A_1 \,[\![ ns_1 \mid ns_2 ]\!]\, A_2) \,[\![ ns_1 \cup ns_2 \mid ns_3 ]\!]\, A_3 = (A_1\ ns_1) \,|||\, (A_2\ ns_2) \,|||\, (A_3\ ns_3)$

**Law[inter-seq]** $A_1 \,[\![ ns_1 \mid ns_2 ]\!]\, A_2 = A_1 ; A_2$
**provided** $usedC(A_1) \cup usedC(A_2) = \varnothing$;
$usedV(A_2) \cap wrtV(A_1) = \varnothing$; $wrtV(A_1) \subseteq ns_1 \cup ns_1'$; and $wrtV(A_2) \subseteq ns_2 \cup ns_2'$.

**Law[inter-seq-extract-snd]** $(A_1;\ A_2) \,[\![ ns_1 \mid ns_2 ]\!]\, A_3 = (A_1 \,[\![ ns_1 \mid ns_2 ]\!]\, A_3);\ A_2$
**provided** $usedC(A_2) = \varnothing$;
$usedV(A_2) \cap wrtV(A_3) = \varnothing$; $wrtV(A_1) \subseteq ns_1 \cup ns_1'$; and $wrtV(A_2) \subseteq ns_1 \cup ns_1'$.

**Law[inter-unused-name]** $A_1 \,[\![\, \{\, n \,\} \cup ns_1 \mid cs \mid ns_2 \,]\!]\, A_2 = A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2$
**provided** $\{\, n, n' \,\} \cap wrtV(A_1) = \varnothing$

## B.3. Hiding

**Law[hid-join]** $(A \setminus cs_1) \setminus cs_2 = A \setminus (cs_1 \cup cs_2)$

**Law[hid-ren]** $(A \setminus \{\!|c|\!\}) = (A[d/c] \setminus \{\!|d|\!\})$
**provided** $d \notin usedC(A)$

**Law[*hid-step]** $(c \rightarrow A) \setminus \{\!|\, c \,|\!\} = A \setminus \{\!|\, c \,|\!\}$

## B.4. Variable blocks

**Law[*var-exp-seq]** $A_1$; (**var** $x : T \bullet A_2$); $A_3 = ($**var** $x : T \bullet A_1$; $A_2$; $A_3)$
**provided** $\{\, x, x' \,\} \cap (FV(A_1) \cup FV(A_3)) = \varnothing$

The set $FV(A)$ contains the free variables of an action $A$.

**Law[var-exp-par]** $((\mathbf{var}\ x : T \bullet A_1) \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2) = (\mathbf{var}\ x : T \bullet A_1 \,[\![\, ns_1 \cup \{x\} \mid cs \mid ns_2 \,]\!]\, A_2)$
**provided** $\{\, x, x' \,\} \notin FV(A_2)$

**Law[var-exp-rec]** $\mu\, X \bullet (\mathbf{var}\ x : T \bullet F(X)) = \mathbf{var}\ x : T \bullet (\mu\, X \bullet F(X))$
**provided** $x$ *is initialised before use in* $F$.

**Law[join-blocks]** $(\mathbf{var}\ x : T_1 \bullet \mathbf{var}\ y : T_2 \bullet A) = (\mathbf{var}\ x : T_1;\ y : T_2 \bullet A)$

## B.5. Processes and programs

**Law[*copy-rule-action]**

$$\begin{array}{l}\textbf{begin}\ (\textbf{state}\ S)\ (n \mathrel{\widehat=} A)\ LADS(n) \bullet MA(n)\ \textbf{end}\\ =\\ \textbf{begin}\ (\textbf{state}\ S)\ (n \mathrel{\widehat=} A)\ LADS(A) \bullet MA(A)\ \textbf{end}\end{array}$$

We use $LADS(n)$ to denote the fact that the local action definitions $LADS$ may include references to the action $n$; the same holds to for the main action $MA(n)$. The later references to $LADS(A)$ and $MA(A)$ are the result of substituting the body $A$ of $n$ for some or all occurrences of $n$ in $LADS$ and $MA$.

**Law[*action-intr]** $(\textbf{begin state}\ S\ LADS \bullet MA\ \textbf{end}) = (\textbf{begin state}\ S\ LADS\ (n \mathrel{\widehat=} A) \bullet MA\ \textbf{end})$
**provided** $n \notin \alpha(S) \cup \alpha(LADS)$

For a schema $S$, $\alpha(S)$ gives the set of names of its components, and for a sequence of local action definitions $LADS$, $\alpha(LADS)$ gives the names it declares and introduces in the scope of the process in which it occurs.

**Law[hid-par-dist]** $(P_1\ (cs_1 \cup \{\!|c|\!\}) \parallel P_2\ cs_2) \setminus \{\!|c|\!\} = (P_1 \setminus \{\!|c|\!\})\ cs_1 \parallel P_2\ cs_2$
**provided** $c \notin cs_2$

**Law[main-var-state]**

$$\begin{array}{l}\textbf{begin}\ (\textbf{state}\ S)\ LADS(x : T) \bullet (\textbf{var}\ x : T \bullet MA)\ \textbf{end}\\ =\\ \textbf{begin}\ (\textbf{state}\ S \wedge [\, x : T\,])\ LADS(\_) \bullet MA\ \textbf{end}\end{array}$$

We write $LADS(x : T)$ to indicate that the local action definitions include schemas that declare variables $x$ and $x'$ of type $T$. The later reference to $LADS(\_)$ denotes the fact that declarations of $x$ (and $x'$) in schemas, which were used to put the local variable $x$ of the main action into scope, may now be removed.

**Law[*parag-intr]** $cp = par\ cp$
**provided** $\alpha(par) \cap \alpha(cp) = \varnothing$

The sets $\alpha(par)$ and $\alpha(cp)$ contain the names introduced by the new paragraph $par$ and the program $cp$.

## B.6. Specialised laws

**Law[var-int-par-join]**

$$\begin{array}{l}\left(\begin{array}{l}(\textbf{var}\ x_1 : T_1;\ x_2 : T_2 \bullet (c?x \to x_1 := x \ [\![\{x_1\} \mid \{x_2\}]\!]\ d?y \to x_2 := y);\ A_1)\\ \quad [\![ns_1 \mid \{c, d\} \cup cs \mid ns_2]\!]\\ (\textbf{var}\ x_1 : T_1;\ x_2 : T_2 \bullet (c?x \to x_1 := x \ [\![\{x_1\} \mid \{x_2\}]\!]\ d?y \to x_2 := y);\ A_2)\end{array}\right)\\ =\\ \left(\begin{array}{l}\textbf{var}\ x_1 : T_1;\ x_2 : T_2 \bullet\\ \quad (c?x \to x_1 := x \ [\![\{x_1\} \mid \{x_2\}]\!]\ d?y \to x_2 := y);\ (A_1\ [\![\, ns_1 \mid \{c, d\} \cup cs \mid ns_2\,]\!]\ A_2)\end{array}\right)\end{array}$$

**provided** $\{x_1, x_2\} \cap (ns_1 \cup ns_2) = \varnothing$

**Law[rec-sync]**

$$\begin{array}{l}(\mu\, X \bullet A_1;\ c \to X)\ [\![\, ns_1 \mid \{\!|\, c \,|\!\} \cup cs \mid ns_2\,]\!]\ (\mu\, X \bullet A_2;\ c \to X)\\ =\\ \mu\, X \bullet (A_1\ [\![\, ns_1 \mid cs \mid ns_2\,]\!]\ A_2);\ c \to X\end{array}$$

**provided** $c \notin usedC(A_1, A_2);\ wrtV(A_1) \cap usedV(A_2) = \varnothing;$ *and* $wrtV(A_2) \cap usedV(A_1) = \varnothing.$

**Law[par-out-inp-inter-exchange]**

$$
\begin{array}{l}
(A_1;\ c_1 \rightarrow \mathit{Skip}) \llbracket\ ns_1 \mid \{\!|\ c_1\ |\!\} \mid ns_2\ \rrbracket ((c_1 \rightarrow A_2 \ \lll ns_3 \mid ns_4 \ggg\ c_2 \rightarrow A_3);\ A_4) \\
= \\
(A_1;\ c_1 \rightarrow A_2 \ \lll ns_1 \cup ns_3 \mid ns_4 \ggg\ c_2 \rightarrow A_3);\ A_4
\end{array}
$$

**provided** $c_1 \neq c_2;\ c_1 \notin \mathit{usedC}(A_1, A_2, A_3, A_4);$
$ns_3 \cup ns_4 \subseteq ns_2;\ \mathit{wrtV}(A_1) \subseteq ns_1;\ \mathit{wrtV}(A_2) \subseteq ns_3;$ and $\mathit{wrtV}(A_4) \subseteq ns_2.$

**Law[inter-split]**

$$
\begin{array}{l}
((A_1;\ c \rightarrow \mathit{Skip}) \llbracket\ ns_1 \mid \{\!|\ c\ |\!\} \cup cs \mid ns_2\ \rrbracket ((A_3 \ \lll ns_3 \mid ns_4 \ggg\ c \rightarrow A_4);\ A_5)) \setminus \{\!|\ c\ |\!\} \\
= \\
((A_1;\ c \rightarrow \mathit{Skip}) \llbracket\ ns_1 \mid \{\!|\ c\ |\!\} \cup cs \mid ns_2\ \rrbracket (A_3\ ;(c \rightarrow A_4);\ A_5)) \setminus \{\!|\ c\ |\!\}
\end{array}
$$

**provided** $c \notin \mathit{usedC}(A_1, A_3, A_5);\ \mathit{usedC}(A_2, A_4) = \varnothing;$
$\mathit{wrtV}(A_3) \subseteq ns_3;\ \mathit{wrtV}(A_4) \subseteq ns_4;$ and $\mathit{usedV}(A_4) \cap \mathit{wrtV}(A_3) = \varnothing.$

**Law[frame-intr]**

$$
\begin{array}{l}
(\mu\ X \bullet A_1;\ A_2;\ X) \\
= \\
\left(\begin{array}{l}
\textbf{var}\ cf : \mathit{FrameIndex} \bullet \\
\quad cf := 1; \\
\quad \left(\mu\, X \bullet \left(\begin{array}{l} \textbf{if}\ (cf = 1) \rightarrow A_1 [\!]\ (cf = 2) \rightarrow A_2\ \textbf{fi}; \\ cf := (cf \bmod 2) + 1; \\ X \end{array}\right)\right)
\end{array}\right)
\end{array}
$$

**provided** $X$ *is not free in* $A_1$ *and* $A_2$*;* $cf$ *is fresh; and* $\{1, 2\} \subseteq \mathit{FrameIndex}.$

**Law[timer-intr]**

$$
\begin{array}{l}
\left(\begin{array}{l}
\mathcal{P}(A_1\ ; cf := 1\ ;\ \mu\, X \bullet \textbf{if}\ cf = 1 \rightarrow A_2\ [\!]\ cf = 2 \rightarrow A_3;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ cf := (cf \bmod 2) + 1;\ X) \\
\qquad \llbracket\ \{\!|\ c\ |\!\} \cup cs\ \rrbracket \\
\mathcal{P}(A_4\ ; cf := 1\ ;\ \mu\, X \bullet \textbf{if}\ cf = 1 \rightarrow A_5\ [\!]\ cf = 2 \rightarrow A_6;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ cf := (cf \bmod 2) + 1;\ X)
\end{array}\right) \\
= \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mathcal{P}(A_1;\ \mu\, X \bullet f?cf \rightarrow \textbf{if}\ cf = 1 \rightarrow A_2\ [\!]\ cf = 2 \rightarrow A_3;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ X) \\
\qquad \llbracket\ \{\!|\ c, f\ |\!\} \cup cs\ \rrbracket \\
\mathcal{P}(A_4;\ \mu\, X \bullet f?cf \rightarrow \textbf{if}\ cf = 1 \rightarrow A_5\ [\!]\ cf = 2 \rightarrow A_6;\ c \rightarrow \mathit{Skip}\ \textbf{fi};\ X)
\end{array}\right) \\
\quad \llbracket\ \{\!|\ c, f\ |\!\}\ \rrbracket \\
\mathcal{P}\left(\begin{array}{l} cf := 1; \\ \mu\, X \bullet f!cf \rightarrow \textbf{if}\ cf = 1 \rightarrow \mathit{Skip}\ [\!]\ cf = 2 \rightarrow c \rightarrow \mathit{Skip}\ \textbf{fi};\ cf := (cf \bmod 2) + 1;\ X \end{array}\right)
\end{array}\right) \setminus \{\!|f|\!\}
\end{array}
$$

**provided** $f$ *is fresh;*
$cf \notin \mathit{wrtV}(A_1, A_2, A_3, A_4, A_5, A_6) \cup \mathit{usedV}(A_1, A_2, A_3, A_4, A_5, A_6);$
$\mathit{usedC}(A_1, A_4) = \varnothing;\ \mathit{usedC}(A_2) \cap \mathit{usedC}(A_6) = \varnothing;$ and $\mathit{usedC}(A_3) \cap \mathit{usedC}(A_5) = \varnothing.$