A Tactic Language for Refinement of State-Rich Concurrent Specifications

M. V. M. Oliveira [*,1]

Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte, Natal, Brazil

A. L. C. Cavalcanti

Department of Computer Science, University of York, York, YO10 5DD, England

Abstract

Circus is a refinement language, in which specifications define both data and behavioural aspects of concurrent systems using a combination of Z and CSP. Its refinement theory and calculus are distinctive, but since refinements may be long and repetitive, using this technique can be hard. Some useful refinement strategies have already been identified, described, and used. By documenting and using them as tactics, they can be repeatedly used as single transformation rules. Here, we present ArcAngel*C*, a language for defining such tactics; we present the language, its semantics, and its application in the formalisation of an existing informal strategy for verification of SPARK Ada implementations of control systems specified using Simulink diagrams.

Key words: Concurrency, refinement calculus, tactics, control law diagrams

1. Introduction

Circus [3] is a formalism that combines Z [31] and CSP [7] to cover both data and behavioural aspects of a system development or verification. It distinguishes itself from other such combinations like CSP-Z [4], TCOZ [11], and CSP-B [28], in that it has a refinement theory and calculus for code development and verification [18]. Using *Circus*, we can develop state-rich reactive systems in a calculational style [14].

In this approach, the repeated application of refinement laws to an abstract specification produces a concrete specification that correctly implements it. This, however, is a hard task, since developments are typically long and repetitive. If refinement strategies can be captured as sequences of law applications, they can be used in different developments, or even many times within a single development. Identifying these strategies, documenting them as tactics, and using them as single refinement laws can save time and effort.

We present ArcAngel*C*, a refinement-tactic language for *Circus* whose constructs are similar to those in ArcAngel [21], a refinement-tactic language for sequential programs. Both languages are based on a general

* Corresponding author.
Email address: `marcel@dimap.ufrn.br` (M. V. M. Oliveira).
URL: `http://www.dimap.ufrn.br/~marcel` (M. V. M. Oliveira).

[1] CNPq supports the work of Marcel Oliveira: grant 550946/2007-1.

tactic language, Angel [13], which is not tailored to any particular proof tool and assumes only that rules transform proof goals. Angel allows the use of angelic choice to define tactics that backtrack to search for successful proofs. Furthermore, it has a formal semantics and an extensive set of laws that provide a complete tool to reason about tactics. The semantics of ArcAngel and its set of laws can be found in [17] along with the formalisation of useful refinement strategies.

Like ArcAngel, as a refinement-tactic language, ArcAngel*C* must take into account the fact that the application of refinement laws yields not only a program, but proof obligations as well. So, the result of applying a tactic is a program and a set of all the proof obligations generated by each law application. In the design of ArcAngel*C*, we adapted the Angel approach to refinements. The constructs of ArcAngel*C* are similar to Angel's, but adapted to deal with the application of the *Circus* refinement laws: its structural combinators are used to apply tactics to *Circus*' programs, processes, and actions.

Many tactic languages can be found in the literature [5, 29, 1, 30]. However, as far as we know, none of them present a formal semantics and support a refinement calculus for concurrent systems. Furthermore, some of these languages do not present operators like recursion and alternative.

In [20], we have presented the novel combinators of ArcAngel*C*. In this paper, besides an informal introduction to the language, we also present the formalisation of ArcAngel*C*'s semantics in Z. This formalisation fosters the mechanisation of the semantics in theorem provers like Z-Eves and ProofPower-Z, and the reasoning about ArcAngel*C* algebraic laws. The semantics of ArcAngel*C* is based on the semantics of ArcAngel [21], but uses more generic definitions to allow the application of tactics to different types of components of a *Circus* specification: actions, processes, and *Circus* programs. In this paper, we focus on novel aspects of the ArcAngel*C* semantics; a full account on the ArcAngel*C* semantics can be found elsewhere [19].

In order to illustrate the usefulness of ArcAngel*C* in practice, in [20], we used it to formalise and generalise the first part of a refinement strategy [2] to prove the correctness of implementations of Simulink diagrams [8] in SPARK Ada. In this paper, we extend this formalisation by also providing the formalisation of the second part of this refinement strategy. This formalisation provides structure and abstraction to the refinement strategy, and fosters its automation [25, 24]; the implementation of ArcAngel*C* is currently in progress.

The next section describes *Circus*. In Section 3, our tactic language for *Circus*, ArcAngel*C*, is presented; its semantics is described in Section 4. The Section 5 describes control law diagrams and uses a simple controller to illustrate them; it also informally describes the refinement strategy that can be used to prove that a given Ada code correctly implements a particular control law diagram [2]. In Section 6, we formalise parts of the refinement strategy presented in [2] as ArcAngel*C* tactics and use them in the verification of a simple controller. Finally, in Section 7, we draw our conclusions and discuss some future work.

2. *Circus*

In *Circus*, programs are declared as a sequence of paragraphs. Each paragraph may be a channel declaration, a Z paragraph, or a process definition. A process defines a system that contains its own state, and communicates with the environment via channels. The main constructs of *Circus* are illustrated in the specification of a register presented below. The register stores a value, which is initialised with zero, and can store or add a given value to its current value. The stored value can also be output or reset.

$$
\begin{array}{l}
\mathbf{channel}\ store, add, out : \mathbb{N};\ result, reset \\
\mathbf{process}\ Register \mathrel{\widehat=} \mathbf{begin\ state}\ RegSt \mathrel{\widehat=} [value : \mathbb{N}] \\
\qquad RegCycle \mathrel{\widehat=} store?newValue \rightarrow value := newValue \\
\qquad\qquad\qquad \Box\ add?newValue \rightarrow value := value + newValue \\
\qquad\qquad\qquad \Box\ result \rightarrow out!value \rightarrow Skip \\
\qquad\qquad\qquad \Box\ reset \rightarrow value := 0 \\
\quad \bullet\ value := 0;\ (\mu X \bullet RegCycle;\ X) \\
\mathbf{end}
\end{array}
$$

Channel declarations **channel** $c : T$ introduce a channel c that communicates values of type T. For instance, **channel** $store, add, out : \mathbb{N}$ declares three different channels that communicate natural numbers.

Processes may be declared in terms of other processes or explicitly. An explicit definition is composed of a state definition, a sequence of paragraphs, and finally, a nameless main action that defines the behaviour of the process. The state is defined as a Z schema; the remaining paragraphs can either be Z paragraphs, or named actions. For instance, the state of process *Register* is defined by the Z schema *RegSt*; it contains a component that stores its *value*.

Three primitive actions are *Skip*, *Stop*, and *Chaos*. The first finishes with no change to the state, the second deadlocks, and the third diverges. Other *Circus* actions may be defined using Z schemas. Finally, actions may be defined as a guarded command, an invocation to other actions, or the combination of actions using CSP operators like hiding, sequence, external and internal choice, parallelism, interleaving, or their corresponding iterated operators.

The process *Register* initialises its *value* to zero and then, has a recursive behaviour. The action *RegCycle* is an external choice: a new value can be stored or accumulated using the channels *store* and *add*; the current *value* is requested through *result*, and then received through *out*, or *reset*.

Circus prefixing is as in CSP. However, it may have a guard associated with it. If the predicate p is true, the action $p \mathbin{\&} c?x \rightarrow A$ assigns the value input through c to a new implicitly declared variable x; it deadlocks otherwise.

Besides the set of channels in which the actions synchronise, the parallelism of actions requires additional information in order to avoid conflicts in the access to the variables in scope: two sets that partition all the variables in scope. In the action $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, the actions synchronise on the channels in the set cs and have access to the initial values of all variables in scope. However, only A_1 and A_2 may modify the values of the variables in ns_1 and ns_2, respectively. The interleaving $A_1 \parallel\!\![ns_1 \mid ns_2]\!\!\parallel A_2$ has a similar behaviour. However, the actions do not synchronise on any channel.

Parametrised actions (and processes) and their instantiation are also available in *Circus*. When applied to actions, the renaming operator substitutes state components and local variables. Finally, actions may be assignments, alternations, variable blocks, or specification statements in the form of [14]. The CSP operators of sequence, choice, parallelism, interleaving, event hiding and renaming may also be used to define processes.

In *Circus*, the basic notion of refinement is that of action refinement [26]. Here, we use some of the refinement laws from [18] like the Law 1 (par-inter) presented below, which transforms a parallel composition into an interleaving.

Law 1 (par-inter) $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = A_1 \parallel\!\![ns_1 \mid ns_2]\!\!\parallel A_2$
provided $(usedC(A_1) \cup usedC(A_2)) \cap cs = \emptyset$

Proof obligations of refinement laws are described in their **provided** condition. They are conditions that need to be met in order to validated the application of the corresponding refinement law. For instance, the application of Law 1 is only valid if none of the channels used in actions A_1 and A_2 are in cs; the function *usedC* returns the set of all channels used in a given action.

Process refinement is defined in terms of action refinement: a process P_2 refines a process P_1 ($P_1 \sqsubseteq_{\mathcal{P}} P_2$) if its main action ($P_2.Act$) refines the main action of P_1 ($P_1.Act$). Both main actions may act on different states and their dashed counterparts, and so may not be comparable. Hence, we compare the actions we obtain by hiding the state components of P_1 and P_2, as if they were declared in a local variable blocks.

Definition 2.1 (Process Refinement) $P_1 \sqsubseteq_{\mathcal{P}} P_2$ *if, and only if,*
$(\exists P_1.State;\ P_1.State' \bullet P_1.Act) \sqsubseteq_{\mathcal{A}} (\exists P_2.State;\ P_2.State' \bullet P_2.Act)$

As discussed above, the state of a process is private. This allows processes' components to be changed during a refinement. This can be achieved in much the same way as we can data refine variable blocks and modules in imperative programs [15]. A well-known technique of data refinement in those contexts is forwards simulation [9]. Details of *Circus* data refinement can be found in [3].

$$
\begin{array}{rlr}
\mathsf{TacticDecl} ::= & \textbf{Tactic}\, \mathsf{N}\ (\, \mathsf{Decl}\,)\ \mathsf{Tactic} & [\textbf{tactic declaration}] \\
& [\, \textbf{generates}\ \mathsf{Prog}\,] & \\
& [\, \textbf{proof obligations}\ \mathsf{Pred}^{+}\,]\ \textbf{end} & \\
\mathsf{Tactic}\ ::= & \textbf{law}\ \ \mathsf{N}\ (\, \mathsf{Exp}^{*}\,) & [\textbf{law application}] \\
\mid & \textbf{tactic}\ \ \mathsf{N}\ (\, \mathsf{Exp}^{*}\,) & [\textbf{tactic application}] \\
\mid & \textbf{skip}\ \mid\ \textbf{fail}\ \mid\ \textbf{abort} & [\textbf{basic tactics}] \\
\mid & \textbf{applies to}\ \ \mathsf{Prog}\ \textbf{do}\ \mathsf{Tactic} & [\textbf{patterns}] \\
\mid & \mathsf{Tactic}\ ;\ \ \mathsf{Tactic}\ \mid\ \mathsf{Tactic} \mid \mathsf{Tactic} & [\textbf{sequence / alternative}] \\
\mid & \mu_{T}\ \mathsf{N} \bullet \mathsf{Tactic}\ \mid\ !\, \mathsf{Tactic} & [\textbf{recursion / cut}] \\
\mid & \textbf{succs}\, \mathsf{Tactic}\ \mid\ \textbf{fails}\, \mathsf{Tactic} & [\textbf{assertions}] \\
\mid & \boxed{\rightarrow}\, \mathsf{Tactic}\ \mid\ \boxed{\&}\, \mathsf{Tactic} & [\textbf{action combinators}] \\
\mid & \boxed{\mu}\, \mathsf{Tactic}\ \mid\ \boxed{\textbf{if}}\, \mathsf{Tactic}^{+}\, \boxed{\textbf{fi}}\ \mid\ \boxed{\textbf{var}}\, \mathsf{Tactic} & \\
\mid & \boxed{\textbf{val}}\, \mathsf{Tactic}\ \mid\ \boxed{\textbf{res}}\, \mathsf{Tactic}\ \mid\ \boxed{\textbf{vres}}\, \mathsf{Tactic} & \\
\mid & \boxed{\textbf{beginend}}\, ((\mathsf{N}, \mathsf{Tactic})^{*}, \mathsf{Tactic}) & [\textbf{process combinators}] \\
\mid & \boxed{\odot}\, \mathsf{Tactic}\ \mid\ \boxed{\odot_{inst}}\, \mathsf{Tactic} & \\
\mid & \boxed{\widehat{=}}\, \mathsf{Tactic}\ \mid\ \mathsf{Tactic}\boxed{;}\mathsf{Tactic} & [\textbf{action/process combinators}] \\
\mid & \mathsf{Tactic}\boxed{\Box}\mathsf{Tactic}\ \mid\ \mathsf{Tactic}\boxed{\sqcap}\mathsf{Tactic} & \\
\mid & \mathsf{Tactic}\boxed{\parallel}\mathsf{Tactic}\ \mid\ \mathsf{Tactic}\boxed{|||}\mathsf{Tactic} & \\
\mid & \boxed{;}\, \mathsf{Tactic}\ \mid\ \boxed{\Box}\, \mathsf{Tactic}\ \mid\ \boxed{\sqcap}\, \mathsf{Tactic}\ \mid\ \boxed{\parallel}\, \mathsf{Tactic} & \\
\mid & \boxed{|||}\, \mathsf{Tactic}\ \mid\ \boxed{\setminus}\, \mathsf{Tactic}\ \mid\ \boxed{\setminus}\, \mathsf{Tactic}\ \mid\ \boxed{:=}\, \mathsf{Tactic} & \\
\mid & \boxed{\bullet}\, \mathsf{Tactic}\ \mid\ \boxed{\bullet_{inst}}\, \mathsf{Tactic} & \\
\mid & \textbf{program}\ \ (\mathsf{N}, \mathsf{Tactic})^{*} & [\textbf{program combinator}]
\end{array}
$$

Fig. 1. Abstract Syntax of ArcAngel*C*

3. ArcAngelC

ArcAngel*C* is a refinement-tactic language similar to ArcAngel [21], which is a tactic language tailored for Morgan's refinement calculus. It includes basic tactics, like a law application, for example; tacticals, which are general tactic combinators; and structural combinators, which support the application of tactics to components of *Circus* programs. The basic tactics and tacticals of ArcAngel*C* are inherited from Angel, and some of its structural combinators are inherited from ArcAngel; nevertheless, the ArcAngel*C*'s structural combinators that are related to the CSP part of *Circus* are a new feature. Furthermore, unlike ArcAngel tactics that can be applied to programs only, ArcAngel*C*'s tactics can be applied to *Circus* programs, processes, and actions. Hence, tactics can be used to prove proof obligations raised in the application of refinement laws like process refinement laws whose proof obligations may contain action refinement statements.

The syntax of ArcAngel*C* is displayed in Figure 1. We use Exp^{*} to denote a possibly empty sequence of elements of the syntactic category Exp of expressions. We use Tactic^{+} to denote a non-empty sequence of tactics. The categories N, Number, Pred, and Decl include the Z identifiers, numbers, predicates and declarations defined in [27]. Finally, the syntactic category Prog denotes the *Circus* programs as in [18].

3.1. *Tactic Declarations*

A tactic program consists of a sequence of tactic declarations. We declare a tactic t named n with arguments a using **Tactic** $\mathsf{n}(a)\ t$ **end**. For documentation purposes, we may include the clause **proof obligations** and the clause **generates**; the former enumerates the proof obligations generated by the application of t, and the latter shows the program generated.

3.2. *Basic Tactics*

The most basic tactic is a law application: **law** $\mathsf{n}(a)\ p$. If the law n with arguments a is applicable to the *Circus* program p, the application succeeds: a new program is returned, possibly generating proof obligations. However, if it is not applicable to p, the application of the tactic fails. A similar construct, **tactic** $\mathsf{n}(a)$, applies the tactic n as though it were a single law.

By way of illustration, the tactic **law** copy-rule-action(N) applies to an action the refinement Law 6 (copy-rule-action), which takes the name N of the action as argument. As a result, it replaces all the references to N by the definition of N. In this case, no proof obligation is generated. A list of the refinement laws used in this paper can be found in Appendix B.

Other basic tactics are provided: the trivial tactic **skip** always succeeds, and the tactic **fail** always fails; finally, the tactic **abort** neither succeeds nor fails, but runs indefinitely.

3.3. *Tacticals*

The tactic **applies to** p **do** t introduces a meta-program p that characterises the programs to which the tactic t is applicable; the meta-variables used in p can then be used in t. For example, the meta-program $A \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ \mathit{Skip}$ characterises those parallel compositions whose right-hand action is *Skip*; here, A, ns_1, cs and ns_2 are the meta-variables. We consider as an example a refinement tactic that transforms a parallel composition into an interleaving: **applies to** $A \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ \mathit{Skip}$ **do law** par-inter().

The tactical $t_1;\ t_2$ applies t_1, and then applies t_2 to the outcome of the application of t_1. If either t_1 or t_2 fails, then so does the whole tactic. When it succeeds, the proof obligations generated are those resulting from the application of t_1 and t_2. For example, we may define a tactic that removes a parallel composition by first transforming it into an interleaving using Law 1 (par-inter), and then simplifies this interleaving using the unit law for interleaving, Law 16 (inter-unit). These two law applications are composed in sequence. The tactic interIntroAndSimpl presented below formalises this tactic. It applies to parallel compositions in which the right-hand action is *Skip* and returns the action A and the proof obligation originated from the application of inter-unit.

```
Tactic interIntroAndSimpl( ) $\widehat{=}$
    applies to $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket\ Skip$
    do law par-inter(); law inter-unit() generates $A$
    proof obligations $usedC(A) \cap cs = \emptyset$
end
```

Tactics may also be combined as alternatives: $t_1 \mid t_2$. First t_1 is applied to the program. If the application of t_1 succeeds, then the composite tactic succeeds; otherwise t_2 is applied to the program. If the application of t_2 succeeds then the composite tactic succeeds; otherwise the composite tactic fails. If one of the tactics aborts, the whole tactic aborts.

The definition of the tactic below uses alternatives. It promotes the local variables declared in the main action to state components. This is the result of an application of either Law 29 (prom-var-state) or Law 30 (prom-var-state-2) depending on whether the process has state or not.

Tactic promoteVars( ) $\widehat{=}$ **law** prom-var-state() | **law** prom-var-state-2()
end

Angelic nondeterminism is implemented through backtracking: on failures, law applications are undone up to the last point where further alternatives are available (as in $t_1 \mid t_2$) and can be explored. This, however, may result in inefficient searches. Some control is given to the programmer through the cut operator: the tactic $!\,t$ behaves like t, except that it returns the first successful application of t. If a subsequent tactic application fails, the whole tactic fails.

ArcAngel*C* has a fixed-point operator that allows us to define recursive tactics. Using μ, we can define a tactic like the one below that exhaustively applies a given tactic t, terminating with success when its application fails.

Tactic EXHAUST(t) $\widehat{=}$ $\mu X \bullet (\, t;\ X \mid \mathbf{skip}\,)$
end

Recursive application of a tactic may lead to nontermination, in which case the result is the same as the basic tactic **abort**.

Two tactics are used to assert the outcome of applying a tactic. The tactic **succs** t fails whenever t fails, and behaves like **skip** whenever t succeeds. On the other hand, **fails** t behaves like **skip** if t fails, and fails if t succeeds. If the application of t runs indefinitely, then these tacticals behave like **abort**. A simple example is a test to check whether a program is a parallel composition. The commutativity law for parallel composition applies only (and always) to parallel compositions. So, our test may be coded as **succs**(**law** par-com()).

3.4. *Structural Combinators*

Often, we want to apply individual tactics to parts of a *Circus* program. In [21], we defined structural combinators that apply to subprograms of sequential programs. ArcAngel*C* extends the number of structural combinators; essentially, there is one combinator for each syntactic construct in *Circus*.

The **Action Structural Combinators** are the ones that allow us to apply a tactic to parts of a *Circus* action. The first one we present allows us to apply a tactic to an action prefixed by an event. The tactic $\boxed{\rightarrow}\, t$ applies to actions of the form $c \rightarrow A$. It returns the prefixing $c \rightarrow B$, where B is the program obtained by applying t to A; the proof obligations generated are those arising from the tactic application. As for the other structural combinators, if the tactic application fails or aborts, so does the application of the whole tactic.

Similarly, the combinator $\boxed{\&}\, t$ applies to a guarded action $g \,\&\, A$ and returns the result of applying t to A; the guard is unaffected in the resulting program. For recursive actions $\mu X \bullet A(X)$, there is the structural combinator $\boxed{\mu}\, t$; it returns recursion obtained by applying t to $A(X)$.

For alternation, there is the structural combinator $\boxed{\mathbf{if}}\, t_1 \boxed{[\!]} \ldots \boxed{[\!]}\, t_n \boxed{\mathbf{fi}}$, which applies to an alternation $\mathbf{if}\ g_1 \rightarrow\ p_1 [\!] \ldots [\!] g_n \rightarrow\ p_n\ \mathbf{fi}$. It returns the result of applying each tactic t_i to the corresponding program p_i. For example, if we apply the tactic $\boxed{\mathbf{if}}$ **law** assign-intro($x := -1$) $\boxed{[\!]}$ **law** assign-intro($x := 1$) $\boxed{\mathbf{fi}}$ to the program $\mathbf{if}\ a \leq b \rightarrow\ x : [\, x < 0\,] [\!]\ a > b \rightarrow\ x : [\, x > 0\,]\ \mathbf{fi}$ we obtain two proof obligations $\mathit{true} \Rightarrow -1 < 0$ and and $\mathit{true} \Rightarrow 1 > 0$, and $\mathbf{if}\ a \leq b \rightarrow\ x := -1 [\!]\ a > b \rightarrow\ x := -1\ \mathbf{fi}$.

The structural combinator $\boxed{\mathbf{var}}\, t$ applies to a variable block; it applies t to the body of the block. By way of illustration, if we apply the tactic $\boxed{\mathbf{var}}$ **law** assign-intro($x := 10$) to $\mathbf{var}\ x : \mathbb{N} \bullet x : [\, x \geq 0\,]$, we get $\mathbf{var}\ x : \mathbb{N} \bullet x := 10$ and the proof obligation $\mathit{true} \Rightarrow 10 \geq 0$. For argument declaration, the combinators $\boxed{\mathbf{val}}\, t$, $\boxed{\mathbf{res}}\, t$, and $\boxed{\mathbf{vres}}\, t$ are used, depending on whether the arguments are passed by value, result, or value-result.

The **Process Structural Combinators** are those combinators that can be applied only to processes bodies. The only *Circus* constructs that are particular to process are the explicit processes definitions (enclosed by the keywords **begin** and **end**) and indexing processes declarations and instantiations.

In order to apply tactics to components of a process explicit declaration we may use the structural combinator $\boxed{\mathbf{beginend}}$. This combinator receives two arguments: a possibly-empty sequence of pairs (n, t) of names

n and tactics t, and another tactic. For each element (n, t) in the sequence received as second argument, this combinator applies t to the paragraph named n of the process; and finally, the second argument is applied to the process main action. For example, the tactic $\boxed{\textbf{beginend}}(\langle(\mathit{RegCycle}, \textbf{tactic}\ T_1())\rangle, \textbf{tactic}\ T_2())$ could be used to apply a tactic T_1 to the body of *RegCycle* and a tactic T_2 to the main action of process *Register*.

Most of the *Circus* constructs originating from CSP can be used in the definition of both processes and actions; therefore, for each of these constructs we define a single **Action/Process Structural Combinator**. Their application are oblivious to whether we are applying the tactic to an action or a process: in both cases they have the same behaviour.

The tactic $t_1 \boxed{;} t_2$ applies to actions/processes of the form $p_1;\ p_2$. It returns the sequential composition of the actions/processes obtained by applying t_1 to p_1 and t_2 to p_2; the proof obligations generated are those arising from both tactic applications. This structural combinator is widely used in Section 6. For instance, one of the steps of the refinement strategy is defined as **skip** $\boxed{;}$ **tactic** interIntroAndSimpl() (See Page 22 for details). This tactic applies to a sequential composition: the left-hand action is left unchanged and the tactic interIntroAndSimpl is applied to right-hand action.

As for the sequential composition, similar structural combinators are available for external choice $(t_1 \boxed{\Box} t_2)$, internal choice $(t_1 \boxed{\sqcap} t_2)$, parallel composition $(t_1 \boxed{\|} t_2)$, interleaving $(t_1 \boxed{|||} t_2)$, event hiding $(\boxed{\setminus}\ t)$, and renaming $(\boxed{:=}\ t)$.

As for the binary constructs, we also have a corresponding structural combinator for each of the indexed CSP constructs that can be used in *Circus*. For instance, $\boxed{;}\ t$ can be applied to an indexed sequential composition $;\ \mathit{decl} \bullet \mathit{body}$: the result is that obtained by the application of t to *body*. For instance, assuming that s is a natural variable that has already been initialised to 0, a program that assigns the sum of all elements of a sequence sq of natural numbers to s can be specified as $;\ i : 0 \mathinner{..} \#sq \bullet s : [s' = s + sq[i]]$. If we apply $\boxed{;}$ **law** assign-intro$(s := s + sq[i])$, we get the program $;\ i : 0 \mathinner{..} \#sq \bullet s := s + sq[i]$ and proof obligations $\mathit{true} \Rightarrow s + sq[i] = s + sq[i]$, for every i in $0 \mathinner{..} \#sq$.

As for indexed sequential composition, we have $\boxed{\Box}$ for indexed external choices, $\boxed{\sqcap}$ for indexed internal choices, $\boxed{\|}$ for indexed parallel composition, and $\boxed{|||}$ for indexed interleaving.

There is only one **Program Structural Combinator**; it can be used to apply tactics to specific paragraphs of a *Circus* program. The tactical **program** receives a sequence of pairs (n, t) of names and tactics: for each element (n, t) in the received sequence, it applies the tactic t to the paragraph named n of the *Circus* program. The tactics used in our case study in Section 6 illustrates the use of this constructor.

4. The Semantics of ArcAngelC

In this section, we describe the semantics of ArcAngel*C*, which is based on the semantics of ArcAngel [21], a refinement-tactic language for sequential programs. The constructors for law and tactic application, the basic tactics, the pattern matching operator, sequence, alternative, cut, recursion, and assertions follow the definitions from [21]. The structural combinators however, although following the approach from [21], use more generic definitions. This derives from the fact that *programs* that can be transformed, in the case of *Circus*, may be different types of components of a *Circus* specification: actions, processes, and *Circus* programs. In this paper, we do not describe the whole semantics in detail; we focus on the main novelties of the ArcAngel*C* semantics, when compared to the ArcAngel work.

As opposed to the ArcAngel semantics, the formalisation of ArcAngel*C* uses Z as a meta-language; we have embedded the syntax of *Circus* in Z. This allows the mechanisation of the semantics using Z theorem provers like ProofPower-Z and Z-Eves.

Tactics are applied to a pair: the first element of this pair is a term to which the tactic is applied, and the second element is the set of proof obligations generated. This pair is called refinement cell. In order to handle different sorts of tactic (action tactics, data refinement tactics, and process tactics), we define a general type of refinement cell, which can have programs, processes, or actions as its first element. First we define a *Cell*: it can be a parametrised action, a parametrised process, a sequence of process paragraphs, or a sequence of program paragraphs.

$$\begin{aligned}
&\mathit{Cell} ::= \mathit{ParActC}\langle\!\langle \mathit{ParAct}\rangle\!\rangle \mid \mathit{ParProcC}\langle\!\langle \mathit{ParProc}\rangle\!\rangle\\
&\qquad\quad \mid \mathit{ProcParC}\langle\!\langle \mathrm{seq}\,\mathit{ProcPar}\rangle\!\rangle \mid \mathit{ProgC}\langle\!\langle \mathrm{seq}\,\mathit{ProgPar}\rangle\!\rangle
\end{aligned}$$

The sets *ParAct*, *ParProc*, *ProcPar*, and *ProgPar* contain (Z representations of) parametrised action, parametrised processes, processes paragraphs, and program paragraphs, respectively. For conciseness, we also define the sets that contains each sort of cell. For instance, the set *ParProcCell* is the set that contains all cells corresponding to parametrised processes.

$$\mathit{ParProcCell} == \mathrm{ran}\,\mathit{ParProcC}$$

A refinement can only transform an action to an action (action refinement), a process to a process (process refinement), or a program to a program (program refinement). We may also have data refinement laws, in which case we need to take into account the retrieve relation (given as an schema expression - *SchemaExp*) and the declaration (*Decl*) of any local variables.

$$\begin{aligned}
&\mathit{Refinement} ::= \mathit{ActRefinement}\langle\!\langle \mathit{ActBody} \times \mathit{ActBody}\rangle\!\rangle\\
&\qquad\quad \mid \mathit{ProcRefinement}\langle\!\langle \mathit{ProcBody} \times \mathit{ProcBody}\rangle\!\rangle\\
&\qquad\quad \mid \mathit{ProgRefinement}\langle\!\langle \mathit{Program} \times \mathit{Program}\rangle\!\rangle\\
&\qquad\quad \mid \mathit{DataRefinement}\langle\!\langle \mathit{SchemaExp} \times \mathit{Decl} \times \mathit{ActBody} \times \mathit{ActBody}\rangle\!\rangle
\end{aligned}$$

The proof obligations can be simple Z predicates (*Pred*) or refinements.

$$\mathit{PObs} == \mathrm{iseq}\,\mathit{Pred} \times \mathrm{iseq}\,\mathit{Refinement}$$

Refinement cells are pairs $(c, pobs)$, where c is a cell and $pobs$ are proof obligations.

$$\mathit{RCell} == \mathit{Cell} \times \mathit{PObs}$$

A refinement law (*Law*) is a function from a *Cell* to a refinement cell. If applied to a certain type of cell it can only return a refinement cell whose first element is of the same type.

$$\begin{aligned}
\mathit{Law} == \{L : \mathit{Cell} \nrightarrow \mathit{RCell} \mid\ & L(\!|\ \mathit{ParActCell}\ |\!) \subseteq \mathit{ParActRCell}\\
&\wedge L(\!|\ \mathit{ParProcCell}\ |\!) \subseteq \mathit{ParProcRCell}\\
&\wedge L(\!|\ \mathit{ProcParCell}\ |\!) \subseteq \mathit{ProcParRCell}\\
&\wedge L(\!|\ \mathit{ProgCell}\ |\!) \subseteq \mathit{ProgRCell}\}
\end{aligned}$$

The restriction is expressed using the Z relational image. For instance, the relational image of the set of all cells that represent a parametrised action (*ParActCell*) is the set of all refinement cells that contain a parametrised action cell as its first element (*ParActRCell*). Informally, this means that the application of a refinement law to a parametrised action may only return a parametrised action (possibly with some proof obligations). The same restrictions apply to parametrised processes (*ParProcCell*), process paragraphs (*ProcParCell*), and programs (*ProgCell*).

The result of a tactic application is a possibly infinite list of *RCell*s that contains all possible outcomes of its application: every program it can generate, together with the corresponding proof obligations (existing obligations and those generated by the tactic application). Different possibilities arise from the use of alternation, and the list can be infinite, since the application of a tactic may run indefinitely. If the application of some tactic fails, then the empty list is returned. The same restrictions on the type of the refinement cells applies to tactics.

$$\begin{aligned}
\mathit{Tactic} == \{\,& T : \mathit{RCell} \rightarrow \mathrm{pfiseq}[\mathit{RCell}]\\
&\mid T(\!|\ \mathit{ParActRCell}\ |\!) \subseteq \mathrm{pfiseq}[\mathit{ParActRCell}]\\
&\quad \wedge T(\!|\ \mathit{ParProcRCell}\ |\!) \subseteq \mathrm{pfiseq}[\mathit{ParProcRCell}]\\
&\quad \wedge T(\!|\ \mathit{ProcParRCell}\ |\!) \subseteq \mathrm{pfiseq}[\mathit{ProcParRCell}]\\
&\quad \wedge T(\!|\ \mathit{ProgRCell}\ |\!) \subseteq \mathrm{pfiseq}[\mathit{ProgRCell}]\}
\end{aligned}$$

The type pfiseq[*RCell*] is that of possibly infinite lists of *RCell*s. We use the model for infinite lists proposed in [12]; it is summarised in the Appendix A. In this model, finite, partial, and infinite lists are considered. A partial list ends in an undefined list, denoted $\perp$. An infinite list is a limit of a directed set of partial lists.

To give semantics to named laws and tactics, we need to maintain two environments, one for refinement laws and one for refinement tactics. A law environment records the known laws; it is a partial function whose domain is the set of the names of these laws. For a law environment env_L and a given law name n, we have that $env_L\ n$ is also a partial function that relates all valid arguments of n (sequence of terms) to yet another function, a Law.

$$LEnv == N \nrightarrow ((\mathrm{seq}\ TERM) \nrightarrow Law)$$

Similarly, a tactic environment is a function that takes a tactic name and a sequence of arguments, and returns a $Tactic$.

4.1. *Legacy Tactics*

The definitions of the basic tactics are very similar to those in [21]. For instance, the basic tactic **law** n(a) applies a simple law to an $RCell$; it is defined as follows.

$$\begin{array}{|l}
\mathbf{law}\colon (N \times (\mathrm{seq}\ TERM)) \rightarrow LEnv \rightarrow Tactic \\
\hline
\forall\, n : N;\ args : \mathrm{seq}\ TERM;\ r : RCell;\ lenv : LEnv \bullet \\
\quad \mathbf{law}\ (n, args)\ lenv\ r = \\
\qquad \mathbf{if}\ (n \in \mathrm{dom}(lenv) \wedge args \in \mathrm{dom}(lenv(n)) \wedge r.1 \in \mathrm{dom}((lenv(n)(args)))) \\
\qquad \mathbf{then}\ [_\infty((lenv(n)(args)(r.1)).1,\ MPObs(r.2, (lenv(n)(args)(r.1)).2))\,]_\infty \\
\qquad \mathbf{else}\ [_\infty\,]_\infty
\end{array}$$

If the law name n is in the given law environment $lenv$, and if the arguments a and cell $r.1$ are appropriate, then the tactic succeeds, and returns a list with a new $RCell$. The refinement cell r is transformed by applying the law to the cell $r.1$; the new proof obligations are merged with the proof obligations $r.2$ of the original $RCell$. Otherwise, the tactic fails with the empty list as result. The function $MPObs$ merges two sets of proof obligations. The brackets subscripted with ∞ indicate that this is a possibly infinite list (pfiseq).

In this work, we use a simple approach for expression arguments. They are used as they were already evaluated. However, they should be evaluated before being used. This is left as future work.

The semantics of **tactic** $n(a)$ is similar to that of the **law** construct. The tactic **skip** returns its argument unchanged.

$$\begin{array}{|l}
\mathbf{skip} : Tactic \\
\hline
\forall\, r : RCell \bullet \mathbf{skip}(r) = [_\infty r\,]_\infty
\end{array}$$

The tactic **fail** always fails. It returns the empty list $[_\infty\,]_\infty$.

The operators are also very similar to the definitions in [21]. The sequence operator, for instance, uses a construction known as the Kleisli composition [10]. It applies its first tactic to its argument, producing a list of cells; it then applies the second tactic to each member of this list; finally, this list-of-lists is flattened to produce the result.

$$\begin{array}{|l}
\_\,;\ \_ : (Tactic \times Tactic) \rightarrow Tactic \\
\hline
\forall\, t1, t2 : Tactic;\ r : RCell \bullet (t1\ ;\ t2)(r) = \overset{\infty}{\frown}/[RCell](t2 * (t1(r)))
\end{array}$$

For a total function $f : A \rightarrow B$, $f* : \mathrm{pfiseq}\, A \rightarrow \mathrm{pfiseq}\, B$ is the map function that operates on a list by applying f to each of its elements; the operator $\overset{\infty}{\frown}/$ is the distributed concatenation operation. Formal definitions of these operators and others to follow can be found in Appendix A.

The semantics of the alternation $(t_1 \mid t_1)$, cut $(!t)$, recursion $(\mu\, X \bullet t)$, **abort**, assertions **succs** t and **fails** t, and pattern matching are also very similar to those from [21] and are omitted here for conciseness. A full account on the semantics of ArcAngel$C$ can be found in [19].

4.2. *Structural Combinators*

The structural combinators apply tactics to components of a program independently (and so can be thought of as in parallel), and then reassemble the results in all possible ways. For instance, let us suppose that we want to apply tactics t_1 and t_2 to two different components p_1 and p_2 of a program P (for this example, let us use $P[p_1, p_2]$ to denote that the program P has components p_1 and p_2). A structural combinator allows us to make this application independently. In our example, let us also assume that $t_1\,(p_1) = [_\infty r_1, r_2\,]_\infty$ and that $t_2\,(p_2) = [_\infty s_1, s_2\,]_\infty$. A combination of these results gives us a list $[_\infty (r_1, s_1), (r_1, s_2), (r_2, s_1), (r_2, s_2)\,]_\infty$. The application of the structural combinator yields a list of programs based on this combination by reassembling the original program and replacing in P each component by the corresponding result in the combined list. In our example, we have the following list of programs as a result.

$$[_\infty P[r_1, s_1], P[r_1, s_2], P[r_2, s_1], P[r_2, s_2]\,]_\infty$$

There is one combinator for each construct in the programming language. In our case, since tactics may target different components of a *Circus* program, we actually have four groups of structural combinators: those that can either be applied to actions or processes, those that can only be applied to actions, those that can only be applied to processes, and those that can be applied to programs.

In this work, the reassembling previously mentioned is done by Ω functions. The semantics of the vast majority of the structural combinators has the following structure.

$$(\mathit{structComb}\ \mathit{tacs})\ \mathit{rc} = (\Omega\ \mathit{args}) * \mathit{tacApp}$$

In this template, *structComb* stands for the structural combinator, *tacs* is either a single tactic, a pair of tactics, or a sequence of tactics, and *rc* is a refinement cell. Their definitions apply the tactic(s) to the refinement cell following some structure (this application is represented by *tacApp* above) and maps $(*)$ a given Ω function using the arguments required by this function (like the original *Circus* construct) to the list *tacApp* resulting from the tactic application.

The naming and typing of Ω functions follows a template: essentially, they are generic functions on X as the one presented below.

$$\Omega_D^R : X \to ((X \times D) \to R) \to \mathit{RCell} \nrightarrow \mathit{RCell}$$

They receive an element x of type X, a function $f : (X \times D) \to R$, and a refinement cell. In this template, X is the type variable, and D and R are to be replaced by *Circus* syntactic categories in the actual definitions (*ActBody*, *ProcPar*, etc). The Ω functions return the refinement cell with same proof obligations and with a cell of type R, which results from the application of f to (x, p), where p is the program structure of the cell in the original refinement cell. In a few cases, the function they receive has type $f : (X \times D \times D) \to R$; this is denoted in the name of the function by subscripting the D with a 2 as in $\Omega_{D_2}^R$. Furthermore, in cases where D and R are the same we may omit R in the name of the function.

4.2.1. *Action Combinators*

The first Ω function, $\Omega_{\mathit{ActBody}}$, instantiates both D and R to the syntactic category of action bodies, *ActBody*. It receives an element x of a certain type X, a function $f : (X \times \mathit{ActBody}) \to \mathit{ActBody}$, and a refinement cell. It returns the refinement cell with same proof obligations and with a basic action as its cell, which results from the application of f to (x, a), where a is the action body of the cell in the original refinement cell.

$$
\begin{array}{|l}
[X] \\
\hline
\Omega_{\mathit{ActBody}} : X \to ((X \times \mathit{ActBody}) \to \mathit{ActBody}) \to \mathit{RCell} \nrightarrow \mathit{RCell} \\
\hline
\forall\, x : X;\ f : (X \times \mathit{ActBody}) \to \mathit{ActBody};\ a : \mathit{ActBody};\ \mathit{pobs} : \mathit{PObs} \bullet \\
\qquad \Omega_{\mathit{ActBody}}\, x\, f\, (\mathit{ParActC}(\mathit{BaseAct}(a)), \mathit{pobs}) = (\mathit{ParActC}(\mathit{BaseAct}(f(x, a))), \mathit{pobs}) \\
\hline
\end{array}
$$

By way of illustration, if we have $c : \mathit{Comm}$ (c is a *Circus* communication) we can have the following application of $\Omega_{ActBody}$. The prefixing $\rightarrow$ is a function $\_ \rightarrow \_ : (Comm \times ActBody) \rightarrow ActBody$.

$$\Omega_{ActBody}\,(c)\,(\rightarrow)\,(ParActC(BaseAct(Skip)), pobs) = (ParActC(BaseAct(\rightarrow (c, Skip))), pobs)$$

If we rebuild a cell by giving to $\Omega_{ActBody}$ a communication c, the function $\rightarrow$, and the cell that contains the basic action $Skip$ and proof obligations $pobs$, as arguments, we get a new refinement cell that contains the action $c \rightarrow Skip$ as its action (represented as $ParActC(BaseAct(\rightarrow (c, Skip)))$ in our embedding of the syntax) and the same proof obligations $pobs$.

Using this Ω function, we can define the structural combinator used for prefixing. As already explained, the structural combinator $\boxed{\rightarrow}\, t$ applies to a prefixing $c \rightarrow A$ ($\rightarrow (c, A)$ in our Z embedding of *Circus*). As a result, t is applied to A; the possible results are assembled back with the prefixing by mapping the function $\Omega_{ActBody}$ with arguments c and $\rightarrow$.

$$\begin{array}{|l}
\boxed{\rightarrow} : Tactic \rightarrow Tactic \\
\hline
\forall\, t : Tactic;\ c : Comm;\ a : ActBody;\ pobs : PObs \bullet \\
\quad (\boxed{\rightarrow}\, t)(ParActC(BaseAct(\rightarrow (c, a))), pobs) = \\
\quad\quad (\Omega_{ActBody}\,(c)\,(\rightarrow)) * (t\,(ParActC(BaseAct(a)), pobs))
\end{array}$$

Basically, for every *Circus* action construct that is defined as a function $f : (X \times ActBody) \rightarrow ActBody$, like $\rightarrow$ above, the semantics of the corresponding structural combinator is similarly defined: we apply the tactic to the action body and reassemble the original action using the function $\Omega_{ActBody}$. This includes the semantics of structural combinators for guarded action ($\boxed{\&}$), recursion ($\boxed{\mu}$), variable blocks ($\boxed{\textbf{var}}$), and parametrised actions ($\boxed{\textbf{val}}$, $\boxed{\textbf{res}}$, and $\boxed{\textbf{vres}}$). The semantics of the structural combinator used for alternation has a more complex definition; it, however, follows the same ideas from [21].

We now turn into the structural combinators that relate to *Circus* process constructs.

4.2.2. *Process Combinators*

The process structural combinators are the ones related to indexed processes ($\boxed{\odot}$ and $\boxed{\odot_{inst}}$) and the one that applies to a process body ($\boxed{\textbf{beginend}}$). The first two of them are similar to those for actions, but take parametrised processes into consideration. For instance, the semantics for the structural combinator $\boxed{\odot}$, uses the function $\Omega_{ParProc}$, which is very similar to $\Omega_{ActBody}$, but applies to parametrised processes instead.

$$\begin{array}{|l}
[X] \\
\hline
\Omega_{ParProc} : X \rightarrow ((X \times ParProc) \rightarrow ParProc) \rightarrow RCell \rightarrow\!\!\!\!\!\!+\ RCell \\
\hline
\forall\, x : X;\ f : (X \times ParProc) \rightarrow ParProc;\ p : ParProc;\ pobs : PObs \bullet \\
\quad \Omega_{ParProc}\, x\, f\,(ParProcC(p), pobs) = (ParProcC(f(x, p)), pobs) \\
\hline
\end{array}$$

This function is used in the following definition.

$$\begin{array}{|l}
\boxed{\odot} : Tactic \rightarrow Tactic \\
\hline
\forall\, t : Tactic;\ p : ParProc;\ d : Decl;\ pobs : PObs \bullet \\
\quad (\boxed{\odot}\, t)(ParProcC(\odot(d, p)), pobs) = (\Omega_{ParProc}\,(d)\,(\odot)) * (t\,(ParProcC(p), pobs))
\end{array}$$

It applies the tactic t to the cell that contains the parametrised process p. We reassemble the cells by mapping the $\Omega_{ParProc}$ function with the original variable declaration d and *Circus* construct $\odot$ to the list that resulted from the application of t.

The structural combinator $\boxed{\textbf{beginend}}$ applies to refinement cells that have an explicit process definition in its cell. It applies each of the received tactics to the corresponding part of the declaration, merges the process paragraphs, and rebuilds the refinement cells. Its definition is omitted here for the sake of conciseness. It, however, like the other definitions that have been omitted here, can be found in [19].

4.2.3. *Action and Process Combinators*

The structure of the definitions of combinators that can be applied to either actions or processes follow the standard way we have used separately for combinators for actions and processes before. Since they are used for both action and process, however, their definition is a conjunction in which each of the conjuncts have the same structure as the simpler definitions previously presented, but define their behaviour for each type of application (rebuilding actions Ω_A and rebuilding process Ω_P).

$$\begin{array}{l}
(\mathit{structComb}_P\ \mathit{tacs})\ \mathit{rc} = (\Omega_P\ \mathit{args}) * \mathit{tacApp} \\
\land\ (\mathit{structComb}_A\ \mathit{tacs})\ \mathit{rc} = (\Omega_A\ \mathit{args}) * \mathit{tacApp}
\end{array}$$

The first definition of a structural combinator that can be applied to processes and actions is for the structural combinator that allows the application of a tactic to the body of a process or action definition, $\boxed{\widehat{=}}$. It uses two Ω functions. The first one receives a generic argument $x : X$, a function f of type $(X \times \mathit{ParAct}) \to \mathit{ProcPar}$, and a refinement cell that contains a parametrised action a and returns a cell that contains a process paragraph resulting from the application of f to (x, a). In our work, ParAct and $\mathit{ProcPar}$ are the syntactic classes of parametrised actions and process paragraphs, respectively.

$$\begin{array}{|l}
[X] \\
\hline
\Omega^{\mathit{ProcPar}}_{\mathit{ParAct}} : X \to ((X \times \mathit{ParAct}) \to \mathit{ProcPar}) \to \mathit{RCell} \nrightarrow \mathit{RCell} \\
\hline
\forall x : X;\ f : (X \times \mathit{ParAct}) \to \mathit{ProcPar};\ a : \mathit{ParAct};\ \mathit{pobs} : \mathit{PObs} \bullet \\
\quad \Omega^{\mathit{ProcPar}}_{\mathit{ParAct}}\ x\ f\ (\mathit{ParActC}(a), \mathit{pobs}) = (\mathit{ProcParC}(\langle f(x, a)\rangle), \mathit{pobs})
\end{array}$$

The second function is similar, but the functions it receives are functions from $(X \times \mathit{ParProc})$ to $\mathit{ProgPar}$. These are the syntactic classes of parametrised processes and program paragraphs, respectively.

$$\begin{array}{|l}
[X] \\
\hline
\Omega^{\mathit{ProgPar}}_{\mathit{ParProc}} : X \to ((X \times \mathit{ParProc}) \to \mathit{ProgPar}) \to \mathit{RCell} \nrightarrow \mathit{RCell} \\
\hline
\forall x : X;\ f : (X \times \mathit{ParProc}) \to \mathit{ProgPar};\ p : \mathit{ParProc};\ \mathit{pobs} : \mathit{PObs} \bullet \\
\quad \Omega^{\mathit{ProgPar}}_{\mathit{ParProc}}\ x\ f\ (\mathit{ParProcC}(p), \mathit{pobs}) = (\mathit{ProgC}(\langle f(x, p)\rangle), \mathit{pobs})
\end{array}$$

The structural combinator that applies to an action or process definition, $\boxed{\widehat{=}}$, is defined in the way described above. If applied to a process definition **process** $n[\mathit{gen}] \mathrel{\widehat{=}} p$ (denoted by $\mathbf{process}((n, \mathit{gen}), p)$ in our embedding of the *Circus* syntax), where $[\mathit{gen}]$ are optional type arguments for generic processes, it applies the tactic to the process body and reconstructs the process definition. Nevertheless, if the combinator is applied to an action definition $n \mathrel{\widehat{=}} a$ (denoted by $\mathit{ActDef}(n, a)$ in our syntactic embedding), it applies the tactic to the action body and reconstructs the action definition.

$$\begin{array}{|l}
\boxed{\widehat{=}} : \mathit{Tactic} \to \mathit{Tactic} \\
\hline
\forall t : \mathit{Tactic};\ n : N;\ \mathit{gen} : \operatorname{seq} N;\ a : \mathit{ParAct};\ p : \mathit{ParProc};\ \mathit{pobs} : \mathit{PObs} \bullet \\
\quad (\boxed{\widehat{=}}\ t)(\mathit{ProgC}(\langle \mathbf{process}((n, \mathit{gen}), p)\rangle), \mathit{pobs}) = \\
\qquad (\Omega^{\mathit{ProgPar}}_{\mathit{ParProc}}\ (n, \mathit{gen})\ (\mathbf{process})) * (t\ (\mathit{ParProcC}(p), \mathit{pobs})) \\
\quad \land\ (\boxed{\widehat{=}}\ t)(\mathit{ProcParC}(\langle \mathit{ActDef}(n, a)\rangle), \mathit{pobs}) = \\
\qquad (\Omega^{\mathit{ProcPar}}_{\mathit{ParAct}}\ n\ \mathit{ActDef})\ * (t\ (\mathit{ParActC}(a), \mathit{pobs}))
\end{array}$$

A similar approach has been adopted for the *Circus* binary operators of sequential composition, external choice, internal choice, parallel composition and interleaving. Nevertheless, their domains are slightly different: sequence, external and internal choice are defined as functions with a domain composed of pairs of tactics, and the remaining constructs are defined as functions with a domain composed of triples. Nevertheless, the first element of these triples have no influence in the behaviour and hence, we can have a single way to define the behaviour of the structural combinator related to all these *Circus* constructs and we have done so by defining a function *generalise* that transforms functions $((X \times X) \to X)$ into functions of type $((\mathit{NIL\_VAL} \times X \times X) \to X)$ in the obvious way.

After this, we could define Ω functions ($\Omega_{\mathit{ProcBody}_2}$ and $\Omega_{\mathit{ActBody}_2}$) that rebuild actions or process refinement cells given a binary operator and two actions or processes. Furthermore, we defined functions that applies each of the two tactics received as arguments to each of the action or process bodies also received as argument ($\mathit{applyTacs}_{\mathit{ProcBody}_2}$ and $\mathit{applyTacs}_{\mathit{ActBody}_2}$), combines both lists of results using the distributed cartesian product and rebuilds the refinement cells using the corresponding Ω functions. In order to have the same structure in the definitions of structural combinators that apply to constructs like sequential composition and in the definition of structural combinators that apply to constructs like parallel composition, this function accepts functions that represent *Circus* constructs that have triples in their domain. Constructs like sequential composition have only two arguments, which are the two actions or process that are composed in sequence. For constructs like parallel composition, however, we have one further argument: a triple (ns_1, cs, ns_2) that contains the state partitions ns_1 and ns_2, and the synchronisation channel set cs.

By way of illustration, we present the final definition of the structural combinator for sequential composition. As we know, the domain of sequential composition are pairs; in order to use the function that applies the tactics, we need to generalise the functions $;_A$ and $;_P$ (embedding of the sequential composition for actions and processes, respectively) before giving them as arguments to the functions that apply the tactics. Besides, a null value, nil, is used as the first argument of this call.

$$\begin{array}{|l}
\_\boxed{;}\_ : (Tactic \times Tactic) \rightarrow Tactic \\
\hline
\forall\, t1, t2 : Tactic;\ a1, a2 : ActBody;\ p1, p2 : ProcBody;\ pobs : PObs \bullet \\
\quad (t1 \boxed{;} t2)(ParProcC(BaseProc(;_P(p1, p2))), pobs) = \\
\qquad applyTacs_{ProcBody_2}\,(nil, generalise\,(;_P), (p1, p2), pobs, (t1, t2)) \\
\quad \wedge\ (t1 \boxed{;} t2)(ParActC(BaseAct(;_A(a1, a2))), pobs) = \\
\qquad applyTacs_{ActBody_2}\,(nil, generalise\,(;_A), (a1, a2), pobs, (t1, t2))
\end{array}$$

The same applies to the structural combinators that apply to external choice and internal choice. For parallel composition of processes and actions (denoted by $\|_P$ and $\|_A$, respectively) we have that the first element of the triples are indeed used and hence, we do not need to generalise the functions $\|_P$ and $\|_A$. We invoke $applyTacs_{ProcBody_2}$ and $applyTacs_{ActBody_2}$ using the original element of the triple and the original function as arguments.

$$\begin{array}{|l}
\_\boxed{\|}\_ : (Tactic \times Tactic) \rightarrow Tactic \\
\hline
\forall\, t1, t2 : Tactic;\ a1, a2 : ActBody;\ p1, p2 : ProcBody; \\
\quad ns1, ns2 : NSExp;\ cs : CSExp;\ pobs : PObs \bullet \\
\quad (t1 \boxed{\|} t2)(ParProcC(BaseProc(\|_P\ (cs, p1, p2))), pobs) = \\
\qquad applyTacs_{ProcBody_2}\,(cs, (\|_P), (p1, p2), pobs, (t1, t2)) \\
\quad \wedge\ (t1 \boxed{\|} t2)(ParActC(BaseAct(\|_A\ ((ns1, cs, ns2), a1, a2))), pobs) = \\
\qquad applyTacs_{ActBody_2}\,((ns1, cs, ns2), (\|_A), (a1, a2), pobs, (t1, t2))
\end{array}$$

The same applies to the interleaving structural combinator.

For the iterated operators, and for the hiding, parametrisation, and renaming operators we follow a similar approach. The only difference is the definitions of the Ω functions, which have to deal with different types of arguments.

5. A Refinement Strategy for Verification of Control System Implementations

Control systems can be specified using block diagrams, which model systems as a directed graph of blocks interconnected by wires. The wires carry signals that represent input and output and the blocks represent functions that determine how the outputs are calculated from the inputs.

Simulink is a popular tool that is part of the Matlab environment[8]; its use in the avionics and automotive sectors is very widespread. A simple example of two Simulink diagrams is presented in Figure 2; it contains a PID (Proportional Integral Derivative) controller, a generic control loop feedback mechanism that attempts to correct the error between a measured process variable and a desired set-point by calculating and then outputting a corrective action that can adjust the process accordingly.

Control systems present a cyclic behaviour. We consider discrete-time models, in which inputs and outputs are sampled at fixed intervals. The inputs and outputs are represented by rounded boxes containing numbers. In our example, there are four inputs, E, Kp, Ki, and Kd, and one output, Y.

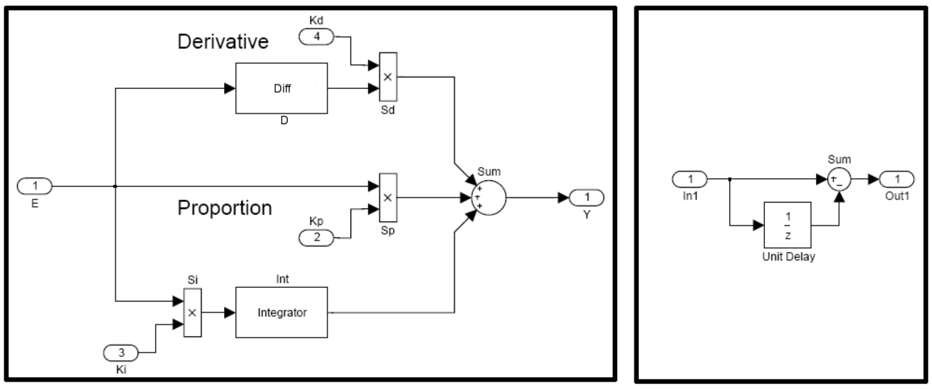

Fig. 2. A Simple PID Controller

Typically, a block takes input signals and produces outputs according to its corresponding function. For instance, the circle is a sum block and boxes with a $\times$ symbol model a product. There are libraries of blocks in Simulink, and they can also be user-defined. Boxes enclosing names are subsystems; they denote control systems defined in other diagrams. For example, the diagram that corresponds to the Diff block is also presented in Figure 2.

Blocks can have state. For instance, Unit Delay blocks store the value of the input signal, and output the value stored in the previous cycle.

In [2], we present a technique to verify SPARK Ada programs with respect to Simulink diagrams using *Circus*. The approach, illustrated in Figure 3, is based on calculating the *Circus* model of the diagram using the semantics given in [2], calculating a *Circus* model for the SPARK Ada program, and proving that the former is refined by the latter.

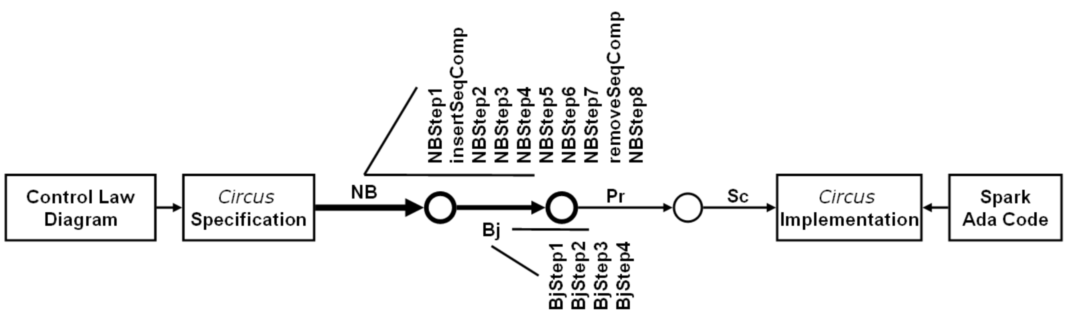

Fig. 3. The Refinement Strategy

In the model of the diagram, there is a basic *Circus* process for each block, and the diagram itself is specified by the parallel composition of these processes. For a subsystem block, the *Circus* process captures the parallel behaviour that arises if some of the outputs do not depend on the values of all the inputs. For example, if there is one output whose value does not depend on the value of all the inputs, as soon as the required inputs become available, its calculation can proceed, and the resulting value can be output. In this case, the calculation of the output is an independent flow of execution of the subsystem. In addition, for all blocks, the update of its state, if any, is an independent flow of execution.

By way of illustration, the translation of the Diff block shown in Figure 2 is the *Diff* process show in Figure 4. Informally, *Init* initialises the process state, *Calc_Diff_out* calculates the output of the differentiator at the next clock cycle, and *Calc_Diff_St* calculates the process state at the current clock cycle; all of them are defined as Z operations on the state of *Diff*.

The inputs of diagrams and blocks are modelled as components $In1?$, $In2?$, and so on. Similarly, outputs have conventional names $Out1!$, $Out2!$, and so on. Components *state*, *state*0, and *initialstate* record the value of the state at the beginning and at the end of the cycle, and at the beginning of the first cycle. The

$\mathbf{process}\ \mathit{Diff} \mathrel{\widehat=} \mathbf{begin}$

$\mathbf{state}\ \mathit{Diff\_St} \mathrel{\widehat=} [\,\mathit{pid\_Diff\_UnitDelay\_St} : \mathbb{U}\,]$

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

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

$$
\begin{array}{|l}
\hline \mathit{pid\_Diff} \\
\mathit{In}1? : \mathbb{U};\ \mathit{Out}1! : \mathbb{U} \\
\mathit{Sum} : \mathit{pid\_Diff\_Sum} \\
\mathit{UnitDelay} : \mathit{pid\_Diff\_UnitDelay} \\
\hline
\mathit{Out}1! = \mathit{Sum}.\mathit{Out}1! \\
\mathit{UnitDelay}.\mathit{In}1? = \mathit{Sum}.\mathit{In}1? \\
\mathit{Sum}.\mathit{In}1? = \mathit{In}1? \\
\mathit{Sum}.\mathit{In}2? = \mathit{UnitDelay}.\mathit{Out}1! \\
\hline
\end{array}
$$

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

$$
\begin{array}{|l}
\hline \mathit{Calculate\_Diff} \\
\Delta \mathit{Diff\_State} \\
\mathit{In}1?m,\ \mathit{Out}1! : \mathbb{U} \\
\hline
\exists\, b : \mathit{pid\_Diff} \bullet \\
\qquad b.\mathit{In}1? = \mathit{In}1? \\
\qquad \land\ b.\mathit{UnitDelay}.\mathit{state} = \mathit{pid\_Diff\_UnitDelay\_state} \\
\qquad \land\ b.\mathit{UnitDelay}.\mathit{state}' = \mathit{pid\_Diff\_UnitDelay\_state}' \\
\qquad \land\ b.\mathit{Out}1! = \mathit{Out}1! \\
\hline
\end{array}
$$

$\mathit{Calculate\_Diff\_Out} \mathrel{\widehat=} \mathit{Calculate\_Diff} \setminus (\mathit{pid\_Diff\_UnitDelay\_state}') \land \Xi \mathit{Diff\_State}$

$\mathit{Exec\_Diff\_out} \mathrel{\widehat=} \mathbf{var}\ \mathit{In}1 : \mathbb{U} \bullet E?x \to \mathit{In}1 := x;\ \mathbf{var}\ \mathit{Out}1 : \mathbb{U} \bullet \mathit{Calc\_Diff\_out};\ \mathit{Diff\_out}!\mathit{Out}1 \to \mathit{Skip}$

$\mathit{Flows} \mathrel{\widehat=} \mathit{Exec\_Diff\_out}$

$\mathit{Calculate\_Diff\_State} \mathrel{\widehat=} \mathit{Calculate\_Diff} \setminus \mathit{Out}1!$

$\mathit{Diff\_StUpdt} \mathrel{\widehat=} \mathbf{var}\ \mathit{In}1 : \mathbb{U} \bullet E?x \to \mathit{In}1 := x;\ \mathit{Calc\_Diff\_St}$

$\bullet\ \mathit{Init};\ \mu X \bullet (\mathit{Flows} \lbrack\!\lbrack\, \{\,\} \mid \{\!|E|\!\} \mid \{\,\mathit{pid\_Diff\_UnitDelay\_St}\} \,\rbrack\!\rbrack\ \mathit{Diff\_StUpdt});\ \mathit{end\_cycle} \to X$

$\mathbf{end}$

Fig. 4. *Circus* process for the block Diff

other components, if any, represent blocks; for each block in the diagram or in the diagram of a subsystem block, there is a component.

For each flow of execution f, the action *Exec_f* takes the required inputs, and then calculates and produces the outputs. The name f of the flow is determined by the unique outputs that it produces. In *Exec_Diff_out* there is one input variable $\mathit{In}1$, and one output variable $\mathit{Out}1$. The inputs are received in any order. The value x of the input is recorded in the corresponding variable *Ini*. Similarly, outputs are sent in any order. In our example, since there is only one input and one output, the interleavings are each reduced to one action: an input through E and an output through *Diff_out*.

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

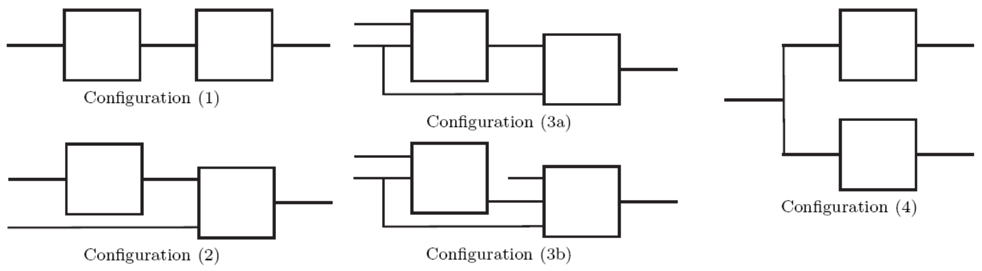

Fig. 5. Blocks Configurations

there is only one flow, so the parallelism in the action is reduced to a single action *Exec_Diff_out* that synchronises with *Diff_StUpdt* on E.

The proof of refinement uses a four-phase strategy. In the first of them, NB, we refine the *Circus* process that corresponds to each block into a recursion that iteratively performs an action that embodies the behaviour of one cycle, and signals the end of the cycle. The action should be in a form similar to that of the model of a SPARK Ada procedure: interleaving of inputs, followed by output calculations and state update, followed by interleaving of outputs.

Informally, the steps in the phase NB are described in [2] as follows: in order to normalise the model of a block we remove the parallelism between the actions that model the flows of execution and the state update, and promote the local variables of the main action to state components. If the block can be implemented sequentially, this step succeeds generating only proof obligations that can be discharged using simple syntactic checks.

After the NB phase, we have the phase BJ, in which we 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. Each of the resulting processes should be refined to put them back into the normal form described in the previous phase NB. 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.

After these two phases, two other phases, Pr, and Sc conclude the refinement. They match the structure of the diagram to the architecture of the scheduler, and prove that the individual procedures implement the block functionality correctly. Their definitions are omitted here for the sake of conciseness. Further details can be found in [2].

6. Case Study - The Tactics NB and BJ

In this section, we present the tactics NB and BJ that formally describe the refinement strategy presented in Section 5. Their application to the example presented here is also discussed; it illustrates how we can accomplish the stages NB and BJ of the refinement strategy by using refinement tactics.

6.1. *Phase* NB

In [2], we describe the NB phase for blocks whose flows share their inputs as in Configuration 4 in Figure 5. The state update is also combined in this way with the flows.

The first step of this phase is a series of applications of the refinement Law copy-rule-action to eliminate all references to action names in the main action. The tactic that accomplishes this step uses a couple of auxiliary tactics in its definition. The first one, TRY, makes a robust application of a given tactic t.

Tactic TRY$(t) \mathrel{\widehat=}\ !(t \mid \textbf{skip})$
end

The next tactic is used to repeatedly apply a given law l using the elements of a given list *args* as arguments, in sequence. It uses the tactic TRY in order to skip when it reaches the base case, an empty list of arguments.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{APPLYL}(l, args) \mathrel{\widehat=}\ \mathsf{TRY}(\ \textbf{law}\ \mathsf{l}(hd\ args);\ \mathsf{APPLYL}(tl\ s)\ )\\ \textbf{end}\end{array}$$

The functions *hd* and *tl* return the head and the tail of a given list, respectively. The former fails if applied to an empty sequence. A similar tactic, APPLYT is used to apply tactics in the same way.

The tactic below formalises the series of applications of Law copy-rule-action. It receives a list *fs* of the names of the actions *Exec_*f that execute the flows as arguments. It applies to explicit process definitions, and transforms the process using Law copy-rule-action () .

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{applyCopyRule}(fs) \mathrel{\widehat=}\\ \quad \textbf{applies to}\ \textbf{process}\ P \mathrel{\widehat=}\ \textbf{begin}\ PPars \bullet Main\ \textbf{end}\\ \quad \textbf{do}\ \boxed{\widehat=}\ \left(\begin{array}{l}\textbf{law}\ \mathsf{copy\text{-}rule\text{-}action}(\text{“}Flows\text{”});\\ \mathsf{APPLYL}(\mathsf{copy\text{-}rule\text{-}action}, fs);\\ \mathsf{TRY}(\textbf{law}\ \mathsf{copy\text{-}rule\text{-}action}(P + \text{“}\_StUpdt\text{”}))\end{array}\right)\\ \textbf{end}\end{array}$$

The tactic that corresponds to the first step of the NB phase, NBStep1, simply receives the list of the action names and invokes **tactic** applyCopyRule(*fs*).

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{NBStep1}(fs) \mathrel{\widehat=}\ \textbf{tactic}\ \mathsf{applyCopyRule}(fs)\\ \textbf{end}\end{array}$$

The application of this tactic to *Diff* changes its main action to the action below in which the references to *Flows*, and then *Exec_Diff_out* (the unique flow) and *Diff_StUpdt* are replaced with their definitions. For that, we give as parameters to NBStep1 the singleton list $\langle Exec\_Diff\_out \rangle$.

$$Init;\ \mu X \bullet \left(\boxed{\begin{array}{l}\left(\begin{array}{l}\textbf{var}\ In1 : \mathbb{U} \bullet\\ \qquad E?x \rightarrow In1 := x;\\ \qquad \textbf{var}\ Out1 : \mathbb{U} \bullet Calc\_Diff\_out;\ Diff\_out!Out1 \rightarrow Skip\end{array}\right)\\ \lbrack\!\lbrack \{\,\} \mid \{\!| E |\!\} \mid \{\, pid\_Diff\_UnitDelay\_St \} \rbrack\!\rbrack\\ (\textbf{var}\ In1 : \mathbb{U} \bullet E?x \rightarrow In1 := x;\ Calc\_Diff\_St)\end{array}}\right);\ end\_cycle \rightarrow X$$

Throughout this paper, we box the target of the next refinement step.

6.1.1. *Synchronise inputs*

All flows in the main action require all inputs, and so does the state update. For this reason, all parallel actions in the body of the recursion declare local variables d_{In} to hold each of the input values, and take all of them in interleaving in A_{In}. In our example, an interleaving is not needed because we have a single input. In this step, we extract from the parallelism the declarations d_{In} using Law 35 (var-exp-par-2) and the interleaving A_{In}, using a law that distributes an action over a parallel composition, Law 27 (par-seq-step-2).

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{syncInput}(\,) \mathrel{\widehat=}\\ \quad \textbf{applies to}\ (\textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In};\ A_{Out})\ \lbrack\!\lbrack\, ns_1 \mid cs \mid ns_2 \,\rbrack\!\rbrack\ (\textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In};\ A_{St})\\ \quad \textbf{do}\ \textbf{law}\ \mathsf{var\text{-}exp\text{-}par\text{-}2}();\ \boxed{\textbf{var}}\ \textbf{law}\ \mathsf{par\text{-}seq\text{-}step\text{-}2}()\ \boxed{\rbrack\!\rbrack}\\ \quad \textbf{generates}\ \textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In};\ (A_{Out}\ \lbrack\!\lbrack\, ns_1 \mid cs \mid ns_2 \,\rbrack\!\rbrack\ A_{St})\\ \quad \textbf{proof obligations}\ \ usedC(A_{In}) \subseteq cs, wrtV(A_{In}) \subseteq \{d_{In}\}\\ \textbf{end}\end{array}$$

This tactic generates a program that declares the input variables, takes the inputs and behaves like a parallel composition.

In our example we have a single flow; nevertheless, we aim at the definition of a tactic that supports multiple flows. In the general case, we have a parallel composition as the one presented below in which the right-hand side is the state update, and the left-hand side is the parallel composition of all the flows.

$$I;\ \mu X \bullet \Big(((\mathbf{var}\ d \bullet A_{In};\ A_{Out_0}) \parallel (\ldots \parallel (\mathbf{var}\ d \bullet A_{In};\ A_{Out_n}))) \parallel (\mathbf{var}\ d \bullet A_{In};\ A_{St}) \Big);\ EC$$

Our strategy is to remove the declarations d and interleaving A_{In} from the parallel composition of all the flows by recursively applying syncInput. Only then, we remove d and A_{In} from the outermost parallel composition. The auxiliary tactic $\mathsf{foldl}_{\parallel}$ recursively applies a given tactic t, from the innermost to the outermost parallel composition of an action $A_1 \parallel (\ldots \parallel A_n)$.

> **Tactic** $\mathsf{foldl}_{\parallel}(t) \mathrel{\widehat{=}} \mu X \bullet$ **tactic** $\mathsf{TRY}((\mathbf{skip} \boxed{\parallel} X);\ t)$
> **end**

For example, the application of **tactic** $\mathsf{foldl}_{\parallel}$ (**tactic** syncInput()) to an instantiation of the generic case in which there are three flows is presented below. The tactic recurs until the point in which the application of the structural combinator $\boxed{\parallel}$ fails (lines 1 to 6), in which case, since we are in a TRY tactic, the tactic skips and returns ($\mathbf{var}\ d \bullet A_{In};\ A_{Out_2}$) (line 7). Then, the tactic applies **tactic** syncInput() to each result of the the recursive invocation: first, it synchronises the inputs of the branches 1 and 2 (lines 8 and 9), and finally, it synchronises all the inputs (lines 10 and 11).

$$
\begin{array}{ll}
(\mathbf{var}\ d \bullet A_i;\ A_{o_0}) \parallel ((\mathbf{var}\ d \bullet A_i;\ A_{o_1}) \parallel (\mathbf{var}\ d \bullet A_i;\ A_{o_2})) & (1)\\
= [\,\mathbf{tactic}\ \mathsf{TRY}\,((\mathbf{skip}\boxed{\parallel}(\mathbf{tactic}\ \mathsf{foldl}_{\parallel}\,(\mathbf{tactic}\ \mathsf{syncInput}())));\ \ldots)\,] & (2)\\
\quad (\mathbf{var}\ d \bullet A_i;\ A_{o_1}) \parallel (\mathbf{var}\ d \bullet A_i;\ A_{o_2}) & (3)\\
\quad = [\,\mathbf{tactic}\ \mathsf{TRY}\,((\mathbf{skip}\boxed{\parallel}(\mathbf{tactic}\ \mathsf{foldl}_{\parallel}\,(\mathbf{tactic}\ \mathsf{syncInput}())));\ \ldots)\,] & (4)\\
\qquad (\mathbf{var}\ d \bullet A_i;\ A_{o_2}) & (5)\\
\qquad = [\,\mathbf{tactic}\ \mathsf{TRY}\,((\mathbf{skip}\boxed{\parallel}(\mathbf{tactic}\ \mathsf{foldl}_{\parallel}\,(\mathbf{tactic}\ \mathsf{syncInput}())));\ \ldots)\,] & (6)\\
\qquad (\mathbf{var}\ d \bullet A_i;\ A_{o_2}) & (7)\\
\quad = [\,\mathbf{tactic}\ \mathsf{TRY}\,(\ldots;\ \mathbf{tactic}\ \mathsf{syncInput}())\,] & (8)\\
\quad (\mathbf{var}\ d \bullet A_i;\ (A_{o_1} \parallel A_{o_2})) & (9)\\
= [\,\mathbf{tactic}\ \mathsf{TRY}\,(\ldots;\ \mathbf{tactic}\ \mathsf{syncInput}())\,] & (10)\\
\mathbf{var}\ d \bullet A_i;\ (A_{o_0} \parallel (A_{o_1} \parallel A_{o_2})) & (11)
\end{array}
$$

In the same way, we may use $\mathsf{foldl}_{\parallel}$ in the n-ary case to join all the variables declarations d and interleaving A_i in the left-hand action of the outermost parallel composition. This is captured by the tactic that follows.

> **Tactic** joinFlowsInput $\mathrel{\widehat{=}}$ **tactic** $\mathsf{foldl}_{\parallel}$ (**tactic** syncInput())
> **end**

The process to which we are applying this step may have state or not: the main action of a stateful process is a parallel composition of the flows with the state update. For this case, we define the following tactic, which synchronises the inputs of the flows, and then, it synchronises the inputs of the whole action.

> **Tactic** NBStep2_f( ) $\mathrel{\widehat{=}}$ (**tactic** joinFlowsInput()$\boxed{\parallel}$**skip**); **tactic** syncInput()
> **end**

Nevertheless, stateless processes do not have a parallel composition with a state update; the application of the tactic above fails. Hence, we define another tactic that synchronises the input of the flows, and then, introduces a parallel composition of the flows output with *Skip*. This unifies the structure of the actions that result from the application of this step to both stateful and stateless processes, allowing the remaining tactics to be used in both of them.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{NBStep2\_l}(\,) \mathrel{\widehat=} \ \textbf{tactic}\ \mathsf{joinFlowsInput}();\ \boxed{\textbf{var}}\,(\textbf{skip}\,\boxed{;}\ \textbf{tactic}\ \mathsf{createPar}())\,\boxed{]\!]}\\ \textbf{end}\end{array}$$

The tactic createPar creates a parallel composition using Laws 16 (inter-unit) and 23 (par-inter-2) in sequence.

Finally, we may define the tactic that corresponds to second step of the NB phase, NBStep2: it is either the application of the stateful version or the application of the stateless version of the second step.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{NBStep2}(\,) \mathrel{\widehat=} \ \textbf{tactic}\ \mathsf{NBStep2\_f}() \mid \textbf{tactic}\ \mathsf{NBStep2\_l}()\\ \textbf{end}\end{array}$$

Our example has one flow; hence, the application of joinFlowsInput immediately skips. Afterwards, the application of syncInput returns the action below.

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

The next step expands the scope of the output variable blocks.

6.1.2. *Expanding the scope of the output variables*

Since there are no repeated declarations of output variables and each output is handled by a single flow, we can expand the scope of the output variable blocks, and join the resulting nested blocks. This can be achieved using Laws 34 (var-exp-par), 37 (var-exp-seq) and 20 (join-blocks).

As for the previous step, we need to define a tactic that supports multiple flows. At this point, the general structure of the main action has a parallel composition as the one presented below in which the left-hand side is the parallel composition of variable blocks that declare different output variables.

$$I;\ \mu\, X \bullet (\,\textbf{var}\ d \bullet A_{In};\ (((\textbf{var}\ d_0 \bullet A_0) \parallel (\ldots \parallel (\textbf{var}\ d_n \bullet A_n))) \parallel A_{St})\,);\ EC$$

The strategy to define the tactic that corresponds to this step is similar to the one used in the previous step: we define a tactic, expDisjVarPar, which extracts both variable blocks from a parallel composition of two variable blocks, and joins them; we use $\mathsf{foldl}_{\parallel}$ to join all the variables blocks in the left-hand action of the outermost parallel composition; and finally, we define a tactic that expands the scope of the output variable blocks to outside the parallel composition and A_{In}, and join the variable blocks.

The tactic expDisjVarPar presented below applies to a parallel composition of two variables block whose sets of declared variables are disjoint. It applies Law var-exp-par to expand the scope of the variable block in the left-hand action to outside the parallelism. Next, it commutes the parallel composition and uses the Law var-exp-par again to expand the scope of the other variable block to outside the parallel composition. Finally, it commutes the parallel composition once again and joins the variable blocks.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{expDisjVarPar}(\,) \mathrel{\widehat=}\\ \quad \textbf{applies to}\ (\textbf{var}\ d_0 \bullet A_0)\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ (\textbf{var}\ d_1 \bullet A_1)\\ \quad \textbf{do}\ \ \textbf{law}\ \mathsf{var\text{-}exp\text{-}par}();\ \boxed{\textbf{var}}\ \textbf{law}\ \mathsf{par\text{-}comm}();\ \textbf{law}\ \mathsf{var\text{-}exp\text{-}par}();\ \boxed{\textbf{var}}\ \ \textbf{law}\ \mathsf{par\text{-}comm}()\,\boxed{]\!]}\,\boxed{]\!]};\\ \qquad \textbf{law}\ \mathsf{join\text{-}blocks}()\\ \quad \textbf{generates}\ \ \textbf{var}\ d_0;\ d_1 \bullet (A_0\ [\![\, ns_1 \mid cs \mid ns_2 \,]\!]\ A_1)\\ \quad \textbf{proof obligations}\ \ \{d_0, d_0'\} \cap FV(A_1) = \emptyset, \{d_1, d_1'\} \cap FV(A_0) = \emptyset\\ \textbf{end}\end{array}$$

Using this tactic, we may join all the variables declarations d_i in the left-hand action of the outermost parallel composition. This is captured by the tactic joinFlowsOutVarScope declared below.

$$\begin{array}{l}\textbf{Tactic}\ \textsf{joinFlowsOutVarScope} \mathrel{\widehat{=}} (\textbf{tactic}\ \textsf{foldl}_{\|}\ (\textbf{tactic}\ \textsf{expDisjVarPar}()))\boxed{\|}\,\textbf{skip}\\ \textbf{end}\end{array}$$

Finally, we define the tactic expOutVarScope, which applies to actions that declare the input variables, receives their values, and then, declares the output variables, and calculates and produces the outputs in parallel with the state update. First, using Law 34 (var-exp-par), we expand the scope of the variable blocks to outside the parallelism. Next, the tactic introduces a *Skip* to obtain an action in the format accepted by Law var-exp-seq, which is then applied to move the variable declaration to include A_{In} in its scope. Finally, the tactics remove the *Skip* that was introduced and joins both variable blocks. The invocation of equality laws superscripted with b (from $b$ackwards) indicates that the law shall be applied from right to left.

$$\begin{array}{l}\textbf{Tactic}\ \textsf{expOutVarScope}(\,) \mathrel{\widehat{=}}\\ \quad \textbf{applies to}\ \ \textbf{var}\ d \bullet A_{In};\ ((\textbf{var}\ d_O \bullet A_O) \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_{St})\\ \quad \textbf{do}\ \ \boxed{\textbf{var}}\,(\,\textbf{skip}\,\boxed{;}\,(\textbf{law}\ \textsf{var-exp-par}();\ \ \textbf{law}\ \textsf{seq-right-unit}())\,);\ \ \textbf{law}\ \textsf{var-exp-seq}();\\ \qquad\qquad \boxed{\textbf{var}}\ \ \textbf{skip}\,\boxed{;}\ \textbf{law}\ \textsf{seq-right-unit}^{b}(\,)\,\boxed{\rrbracket}\,\boxed{\rrbracket};\\ \qquad \textbf{law}\ \textsf{join-blocks}()\\ \quad \textbf{generates}\ \ \textbf{var}\ d;\ d_O \bullet A_{In};\ (A_O \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\, A_{St})\\ \quad \textbf{proof obligations}\ \ \{d_{Out}, d'_{Out}\} \cap FV(A_{St}) = \emptyset, \{d_{Out}, d'_{Out}\} \cap FV(A_{In}) = \emptyset\\ \textbf{end}\end{array}$$

The result is a single variable block that declares input and output variables. The tactic that corresponds to the third step of the NB phase, NBStep3, first joins all the variables blocks in the left-hand action of the outermost parallel composition. Finally, it invokes **tactic** expOutVarScope() in order to expand the scope of the block that introduces the output variables, and joins the resulting nested blocks.

$$\begin{array}{l}\textbf{Tactic}\ \textsf{NBStep3}(\,) \mathrel{\widehat{=}}\ (\boxed{\textbf{var}}\,\textbf{skip}\,\boxed{;}\ \textbf{tactic}\ \textsf{joinFlowsOutVarScope}()\,\boxed{\rrbracket});\ \ \textbf{tactic}\ \textsf{expOutVarScope}()\\ \textbf{end}\end{array}$$

As for the previous step, the application of the tactic joinFlowsOutVarScope immediately skips in our example because it contains only one flow. The application of the tactic expOutVarScope yields the following action.

$$Init;\ \mu X \bullet \left(\begin{array}{l}\textbf{var}\ In1 : \mathbb{U};\ Out1 : \mathbb{U} \bullet\\ \qquad E?x \rightarrow In1 := x;\\ \qquad \boxed{\begin{array}{l}(Calc\_Diff\_out;\ Diff\_out!Out1 \rightarrow Skip)\\ \llbracket\, \{\,\} \mid \{\!|E|\!\} \mid \{\, pid\_Diff\_UnitDelay\_St\} \,\rrbracket\\ Calc\_Diff\_St\end{array}}\end{array}\right);\ end\_cycle \rightarrow X$$

The next step removes all schemas that calculates the outputs and updates the state from the parallel composition.

6.1.3. *Isolating the input processing*

The fourth step aims at isolating the communication of the output values. In the most general case, at this stage, we have a parallel composition as the one presented below, in which the right-hand action is the state update and the left-hand action is the parallel composition of the flows: each flow calculates the output values and communicates them.

$$I;\ \mu X \bullet (\textbf{var}\ d;\ d_O \bullet A_{In};\ (((A_{C_0};\ A_{O_0}) \parallel (\ldots \parallel (A_{C_n};\ A_{O_n}))) \parallel A_{St}));EC$$

As before, the strategy is to define a tactic that isolates the output communications in a parallel composition of two flows, use $\textsf{foldl}_{\|}$ to isolate all the output communications in the left-hand action of the outermost parallel composition, and finally, define a tactic that isolates the output communications in the outermost parallel composition.

The tactic isolateSeqActions presented below applies to a parallel composition $(A_{C_0}; A_{O_0}) \parallel (A_{C_1}; A_{O_1})$. It applies Law 26 (par-seq-step) to remove the schema A_{C_0} from the parallel composition resulting in a sequential composition. Next, it commutes the remaining parallel composition and uses the Law par-seq-step again to remove the schema A_{C_1} from the parallel composition. Finally, it commutes the parallel composition once again and applies the associativity law for parallel composition in order to aggregate A_{C_0} and A_{C_1}.

```
Tactic isolateSeqActions( ) $\widehat{=}$
    applies to $(A_{C_0}; A_{O_0}) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (A_{C_1}; A_{O_1}))$
    do law par-seq-step();
        (skip $\boxed{;}$ (law par-comm(); law par-seq-step(); (skip $\boxed{;}$ law par-comm())));
        law seq-assoc()
    generates $(A_{C_0}; A_{C_1}); (A_{O_0} \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{O_1})$
    proof obligations $usedC(A_{C_0}) = \emptyset, usedC(A_{C_1}) = \emptyset,$
                      $wrtV(A_{C_0}) \subseteq ns_1 \cap ns_1', wrtV(A_{C_1}) \subseteq ns_2 \cap ns_2'$
                      $usedV(A_{C_1}; A_{O_1}) \cap wrtV(A_{C_0}) = \emptyset, usedV(A_{C_0}) \cap wrtV(A_{C_1}) = \emptyset$
end
```

The proof obligations are originated from the applications of Law par-seq-step. Using this tactic, we may isolate all the output communications A_{O_i} in the left-hand action of the outermost parallel composition. This is captured by the tactic joinFlowsCalc declared below.

```
Tactic joinFlowsCalc $\widehat{=}$ (tactic foldl$_{\parallel}$ (tactic isolateSeqActions())) $\boxed{\parallel}$ skip
end
```

Finally, we can define the tactic isolateIn, which introduces a *Skip* into the right branch of the parallel composition and then uses Law par-seq-step to remove the schemas A_{C_i} that calculate the outputs from the parallel composition resulting in a sequential composition. Then, it works on the second part of this sequential composition: it commutes the parallel composition and then it applies once again Law par-seq-step in order to remove the schemas A_{St} that calculates the state. Once again, it commutes the remaining parallel composition. Finally, it applies the Law 31 (seq-assoc) to the whole sequential composition; this aggregates the output calculation and the state update.

```
Tactic isolateIn( ) $\widehat{=}$
    applies to $(A_C; A_O) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_{St}$
    do ( skip $\boxed{\parallel}$ (law seq-right-unit()) ); law par-seq-step();
        ( skip $\boxed{;}$ (law par-com(); law par-seq-step(); (skip $\boxed{;}$ law par-com())) );
        law seq-assoc()
    generates $(A_C; A_{St}); (A_O \llbracket ns_1 \mid cs \mid ns_2 \rrbracket Skip)$
    proof obligations $usedV(A_{St}) \cap wrtV(A_{Calc}) = \emptyset,$
                      $usedV(A_{Comm}) \cap wrtV(A_{St}) = \emptyset,$
                      $wrtV(A_{Calc}) \subseteq ns_1 \cup ns_1', wrtV(A_{St}) \subseteq ns_2 \cup ns_2'$
end
```

This step is applied to the result of step three, which is a sequential composition $A_{In}; (A_{Out} \parallel A_{St})$. Its objective is to apply isolateIn to the parallel composition. Nevertheless, the system may have many flows;

hence, we first need to isolate all the output communications in A_{Out}. Afterwards, we are able to apply isolateIn to the parallel composition. Finally, Law seq-assoc isolates the parallel composition as the second part of a sequential composition.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{NBStep4}(\,) \mathrel{\widehat=} (\,\textbf{skip}\ \boxed{;}\ (\,\textbf{tactic}\ \mathsf{joinFlowsCalc}();\ \textbf{tactic}\ \mathsf{isolateIn}()\,)\,);\ \textbf{law}\ \mathsf{seq\text{-}assoc}()\\ \textbf{end}\end{array}$$

In our example, the application of the tactic joinFlowsCalc immediately skips. The application of the tactic isolateIn yields the following action.

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

Finally, the next step removes the parallel composition from the main action.

6.1.4. *Introducing and simplifying interleaving of outputs*

None of the input variables occur in the parallelism resulting from the last step. Hence, we can use the tactic interIntroAndSimpl presented in Section 3.3 to simplify this parallel composition. The result of the previous step is a sequence: the first part of the sequence processes inputs and calculates the outputs and the state, and the second part of the sequence is the parallel composition; we apply interIntroAndSimpl only to the second part.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{NBSteps5\_6}(\,) \mathrel{\widehat=}\ \textbf{skip}\ \boxed{;}\ \textbf{tactic}\ \mathsf{interIntroAndSimpl}()\\ \textbf{end}\end{array}$$

In our example, the application of this tactic yields the following action.

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

Next, we extend the scope of the variables blocks to the whole main action.

6.1.5. *Extend scope of the variable declarations to the outer level*

At this stage, the main action's format is A_{In}; $(\mu X \bullet (\textbf{var}\ d \bullet A_{OutSt});\ EC)$. We expand the scope of d to the outer level using the unit laws for sequence, and Laws 36 (var-exp-rec) and 37 (var-exp-seq) as follows. First, we introduce a *Skip* to the left of the sequential composition in the body of the recursion. Next, we expand the scope of d to the whole sequential composition in the body of the recursion (Law var-exp-seq), remove the *Skip* that was introduced, and expand the scope of d over the recursion (Law var-exp-rec). Finally, we introduce a *Skip* to the sequential composition in the main action, expand the scope of d to the whole sequential composition (Law var-exp-seq), and remove the *Skip* that was introduced. At the end, we have

$\mathbf{var}\ d \bullet A_{In};\ (\mu X \bullet (A_{OutSt};\ EC))$ as the main action.

$$
\begin{array}{l}
\textbf{Tactic}\ \mathsf{extendVarScope}(\,) \mathrel{\widehat=} \\
\quad \textbf{applies to}\ \ A_{In};\ (\mu X \bullet (\mathbf{var}\ d \bullet A_{OutSt});\ EC) \\
\quad \textbf{do}\ \left(\textbf{skip} \boxed{;} \left(\begin{array}{l} (\boxed{\mu}\ (\textbf{law}\ \mathsf{seq\text{-}left\text{-}unit}();\ \textbf{law}\ \mathsf{var\text{-}exp\text{-}seq}();\ \boxed{\textbf{var}}\ \textbf{law}\ \mathsf{seq\text{-}left\text{-}unit}^{b}()\,\boxed{]\!]}));\\ \textbf{law}\ \mathsf{var\text{-}exp\text{-}rec}();\ \textbf{law}\ \mathsf{seq\text{-}right\text{-}unit}() \end{array} \right) \right); \\
\qquad \textbf{law}\ \mathsf{var\text{-}exp\text{-}seq}();\ \boxed{\textbf{var}}\ (\textbf{skip} \boxed{;}\ \textbf{law}\ \mathsf{seq\text{-}right\text{-}unit}^{b}())\,\boxed{]\!]} \\
\quad \textbf{generates}\ \ \mathbf{var}\ d \bullet A_{In};\ (\mu X \bullet (A_{OutSt};\ EC)) \\
\quad \textbf{proof obligations}\ \{d, d'\} \cap (FV(A_{In}) \cup FV(EC)) = \emptyset,\ d \text{ are initialised before use in } A_{OutSt} \\
\textbf{end}
\end{array}
$$

The proof obligations are those originated from the application of the expansion laws. The simple application of extendVarScope represents the seventh step of the phase **NB**.

$$
\begin{array}{l}
\textbf{Tactic}\ \mathsf{NBStep7}(\,) \mathrel{\widehat=}\ \textbf{tactic}\ \mathsf{extendVarScope}() \\
\textbf{end}
\end{array}
$$

The result of its application to our example yields the following main action.

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

This concludes the transformation in the main action of the process.

6.1.6. *Promote local variables to state components*

In the last step, the tactic NBStep8 simply invokes the tactic promoteVars in order to turn the input and output variables into state components. This concludes the application of the refinement strategy, which, in our example, results in the following process.

$$
\begin{array}{l}
\mathbf{process}\ Diff \mathrel{\widehat=}\ \mathbf{begin} \\
\mathbf{state}\ Diff\_St \mathrel{\widehat=} [\,pid\_Diff\_UnitDelay\_St : \mathbb{U};\ In1 : \mathbb{U};\ Out1 : \mathbb{U}\,] \\
\quad \ldots \\
\quad \bullet\ Init;\ \mu X \bullet \left(\begin{array}{l} (\,(E?x \rightarrow In1 := x);\ (Calc\_Diff\_out;\ Calc\_Diff\_St)\,); \\ (Diff\_out!Out1 \rightarrow Skip) \end{array} \right);\ end\_cycle \rightarrow X \\
\mathbf{end}
\end{array}
$$

There is one tactic NBStep$i$, for each of the steps i of the refinement strategy. We compose most of these tactics in the tactic NBMain. Furthermore, two auxiliary tactics are used in NBMain. As previously discussed, the process we are dealing with may have a state or not. The example presented here falls in the first case: its main action is a sequential composition of a schema that initialises the state and a recursion. In the second case, however, since there is no state to initialise, the main action is just a recursion. In order to have the same structure (a sequential composition) in both cases, we use two auxiliary tactics, insertSeqComp and removeSeqComp. In the absence of a sequential composition, the tactic insertSeqComp introduces one, using law seq-left-unit; otherwise, it skips.

$$
\begin{array}{l}
\textbf{Tactic}\ \mathsf{insertSeqComp}(\,) \mathrel{\widehat=} \mathsf{TRY}(\textbf{fails}(\textbf{skip} \boxed{;} \textbf{skip});\ (\textbf{law}\ \mathsf{seq\text{-}left\text{-}unit}())) \\
\textbf{end}
\end{array}
$$

The second one, tactic removeSeqComp removes any sequential composition with *Skip* using Laws 32 (seq-left-unit) and 33 (seq-right-unit).

$$\textbf{Tactic}\ \mathsf{removeSeqComp}(\,) \mathrel{\widehat=} \mathsf{TRY}(\,\textbf{law}\ \mathsf{seq\text{-}left\text{-}unit}^b()) \ ;\ \mathsf{TRY}(\,\textbf{law}\ \mathsf{seq\text{-}right\text{-}unit}^b())$$
$$\textbf{end}$$

The tactic NBMain is applied to the main action of the processes. After introducing a sequential composition, if needed, it works on the body of the recursion. This body is a sequential composition in which the second part ends the cycle and is not changed. Hence, the tactic only changes its first action: it applies NBStep2 (creating a parallel composition with *Skip* if needed), NBStep3, NBStep4, and NBSteps5_6. Finally, we apply the seventh step and remove any sequential composition with *Skip* in the variable block.

$$\begin{array}{l} \textbf{Tactic}\ \mathsf{NBMain}(\,) \mathrel{\widehat=} \textbf{tactic}\ \mathsf{insertSeqComp}(); \\ \qquad \left(\textbf{skip}\ \boxed{;}\ \boxed{\mu} \left(\left(\begin{array}{l} \textbf{tactic}\ \mathsf{NBStep2}();\ \textbf{tactic}\ \mathsf{NBStep3}(); \\ \boxed{\textbf{var}}\ \textbf{tactic}\ \mathsf{NBStep4}(); \textbf{tactic}\ \mathsf{NBSteps5\_6}()\ \boxed{]\!]} \end{array} \right) \boxed{;} \textbf{skip} \right) \right); \\ \qquad \textbf{tactic}\ \mathsf{NBStep7}(); \boxed{\textbf{var}}\ \textbf{tactic}\ \mathsf{removeSeqComp}()\ \boxed{]\!]} \\ \textbf{end} \end{array}$$

The tactic NBProc presented below can be applied to normalise the process that corresponds to a individual block: it receives a process name and normalises the corresponding *Circus* process. First, it applies the tactic NBStep1 using a list that contains the names of the actions of the process that execute its flows that is returned by the function FNames. Then, it applies the tactic NBMain to the main action of the process. Finally, it promotes the variables declared in the beginning of the resulting main action to state components (NBStep8).

$$\begin{array}{l} \textbf{Tactic}\ \mathsf{NBProc}(pname) \mathrel{\widehat=} \textbf{program} \left\langle \left(\begin{array}{l} pname, \textbf{tactic}\ \mathsf{NBStep1}(\mathsf{FNames}(pname)); \\ \qquad \boxed{\widehat=} \left(\begin{array}{l} \boxed{\textbf{beginend}}\ (\langle\rangle, \textbf{tactic}\ \mathsf{NBMain}()); \\ \textbf{tactic}\ \mathsf{NBStep8}() \end{array} \right) \end{array} \right) \right\rangle \\ \textbf{end} \end{array}$$

This tactic refines 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 interleaving of outputs, and synchronisation on *end_cycle*.

Using this tactic, we may also refine the remaining components shown in Figure 2; the refinement of Int, Si, Sd, Sp, and Sum can be accomplished with simple applications of tactic NBProc. We achieve this by applying the following tactic to the *Circus* program that contains their specifications.

$$\textbf{Tactic}\ \mathsf{NB}(ind\_blocks) \mathrel{\widehat=} \mathsf{APPLYT}(\mathsf{NBProc}, ind\_blocks)$$
$$\textbf{end}$$

This tactic receives an argument that is a list of block names. In our example, we have that the list $\langle Diff, Sd, Int, Si, Sp, Sum \rangle$ can be used to apply the phase NB to the whole *Circus* program.

Although not presented in this paper, Si, Sd, Sp, and Sum do not have state and, as a direct consequence, do not have a parallel composition in the main action because they do not need to have any state update. The first three of them, Si, Sd, and Sp, take two input values and produce one output value; the last one of them Sum takes three input values and produces one output value. Regardless of the difference in the internal structure of these processes, however, the tactic NB can be applied with success.

6.2. *Phase* BJ

In the phase BJ, we use the 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, we identify a procedure `Calc_Derivative` that implements the functionality of the blocks Diff and Sd. Similarly, we can also also find a procedure `Calc_Integral` that implements the blocks Si and Int. Finally, the main program has procedures `Calc_Proportion`, which implements the block Sp, and `Calc_Output`, which implements the block Sum.

We consider each of these procedures, or more precisely, those that implement more than one block. For each of them, we remove, in the process that defines the diagram, the parallelism between the processes that model the blocks that they implement. 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. Afterwards, with the collection of processes now in correspondence with the procedures of the implementation, we then group the processes that correspond to procedures scheduled by a single task. In our case, the procedures `Calc_Proportion` and `Calc_Output`, which implement the blocks Sp and Sum, are scheduled by the same program. Therefore, in this phase, we also join the processes *Sp* and *Sum* to produce a process *Sp_Sum*.

To illustrate the steps of this phase, we join the processes *Diff* and *Sd*, which model Diff and Sd. For *Int* and *Si*, and *Sp* and *Sum*, of course, we proceed in a similar way.

6.2.1. *Create a single process*

The first step of the phase BJ joins the processes that are implemented in a same procedure or in different procedures that are scheduled by the same task in the Ada code together. For that, it receives as argument a sequence that contains sequences of blocks that are to be joined. In our example, we use $\langle\langle Diff, Sd\rangle, \langle Int, Si\rangle, \langle Sp, Sum\rangle\rangle$ (henceforth called *pid_blocks*) as argument. That means that, for instance, that processes *Diff* and *Sd* are to be joined. In fact, the argument *ind_blocks* given to the tactic NB is the distributed concatenation of *pid_blocks*.

The tactic below uses Law 5 (join-proc-par) to join the processes. This law is based directly on the definition of process parallelism [23]. 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. For simplicity, we assume that 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. The overall program is changed to refer to the newly created process instead of the parallel composition; furthermore, the individual processes are also removed from the specification.

The processes are joined two at a time. For this reason, we use an auxiliary function *join_all* that receives a list of list of process names, in which each member is a list of process that must be joined. If the list cardinality is less than two, it ignores the list, otherwise, it creates a list of arguments for Law join-proc-par. This function is particularly useful if we have procedures implementing more than two blocks. For instance, we have that if we apply *join_all* to $\langle\langle P_1, P_2, P_3\rangle\rangle$, we get $\langle(P_1, P_2), (P_1\_P_2, P_3)\rangle$ as result. That means that we first join P_1 and P_2 and then, we join the resulting process and P_3.

In our example, we get the list $\langle(Diff, Sd), (Int, Si), (Sp, Sum)\rangle$, which is interactively used, as presented below, as argument for Law join-proc-par.

> **Tactic** createSingleProcesses(*blocks*) $\widehat{=}$ APPLYL(join-proc-pars, *join_all*(*blocks*))
> **end**

The simple invocation of this tactic formalises the first step of the refinement strategy.

> **Tactic** BJSt1(*blocks*) $\widehat{=}$ createSingleProcesses(*blocks*)
> **end**

In our example, the application of this tactic, using the list *pid_blocks* as argument, we get the *Circus* program sketched in Figure 6. The *Ini* and *Outj* variables in the state are renamed when the processes are joined to avoid clashes as explained above. 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.

The next steps aim at normalising the main action of the joined processes.

process $Diff\_Sd \mathrel{\widehat=}$ **begin state** $Diff\_State \land Sd\_State$

$Diff\_State \mathrel{\widehat=} [\, pid\_Diff\_UnitDelay\_state : \mathbb{U};\ pid\_Diff\_In1 : \mathbb{U};\ pid\_Diff\_Out1 : \mathbb{U} \,]$
$Sd\_State \mathrel{\widehat=} [\, pid\_Sd\_In1, pid\_Sd\_In2 : \mathbb{U};\ pid\_Sd\_Out1 : \mathbb{U} \,]$

$\ldots$

$$\bullet\ \boxed{\begin{array}{l}
\left(\begin{array}{l}
pid\_Diff\_Init; \\
\mu\, X \bullet \left(\left(\begin{array}{l}
\left(\begin{array}{l}
E?x \rightarrow \\
\left(\begin{array}{l}
(pid\_Diff\_In1 := x;\ Calculate\_Diff\_out); \\
Calculate\_Diff\_State
\end{array}\right)
\end{array}\right); \\
(Diff\_out!pid\_Diff\_Out1 \rightarrow Skip)
\end{array}\right); endCycle \rightarrow X\right)
\end{array}\right) \\
\llbracket\ \{pid\_Diff\_UnitDelay\_state, pid\_Diff\_In1, pid\_Diff\_Out1\} \\
\quad \mid \{\!|\ Diff\_out, end\_cycle\ |\!\} \mid \\
\quad \{pid\_Sd\_In1, pid\_Sd\_In2, pid\_Sd\_Out1\}\ \rrbracket \\
\left(\mu\, X \bullet \left(\left(\begin{array}{l}
\left(\begin{array}{l}
Diff\_out?x \rightarrow pid\_Sd\_In2 := x \\
\interleave\!\llbracket\, \{\, pid\_Sd\_In2 \,\} \mid \{\, pid\_Sd\_In1 \,\} \,\rrbracket\!\interleave \\
Kd?x \rightarrow pid\_Sd\_In1 := x
\end{array}\right); \\
(pid\_Sd;\ Sd\_out!pid\_Sd\_Out1 \rightarrow Skip)
\end{array}\right); endCycle \rightarrow X\right)\right)
\end{array}}$$

end

process $Si\_Int \mathrel{\widehat=}\ \ldots$ **end**

process $Sp\_Sum \mathrel{\widehat=}\ \ldots$ **end**

process $PID \mathrel{\widehat=}\ \ldots$ **end**

Fig. 6. Result of applying BJStep1

6.2.2. *Extract Initialisations*

This step of the refinement strategy removes the initialisation from the parallelism. At this stage, the main action of the processes that have been joined have the following structure.

$$(((I_0;\ R_0) \parallel (I_1;\ R_1)) \parallel \ldots) \parallel (I_n;\ R_n)$$

Where I_i are the state initialisation, if any, and R_i are the recursive behaviours that synchronise on *end_cycle* at the end of each cycle.

The tactic isolateSeqActions from Section 6.1.3 applies to a parallel composition $(A_{C_0};\ A_{O_0}) \parallel (A_{C_1};\ A_{O_1})$ and removes A_{C_0} and A_{C_1} from the parallel composition; hence, in the binary case, it can also be used to isolate the initialisations. However, since the original blocks may have state initialisation or not, before invoking this tactic, we must guarantee that there will be a sequential composition. For that, we use the tactic insertSeqComp on both sides of the parallel composition before invoking isolateSeqActions. Finally, we remove any sequential composition with *Skip* that might have been included earlier.

Tactic isolateInitBin() $\mathrel{\widehat=}$ (**tactic** insertSeqComp()$\boxed{\parallel}$ **tactic** insertSeqComp()) ;
tactic isolateSeqActions() ; (**tactic** removeSeqComp()$\boxed{;}$**skip**)
end

For a given process name, the tactic BJSt2Proc extracts the initialisations from the parallel composition in the main action of the process. In our example we have joined only two processes. However, in the general case, we may have joined more than two processes together, in which case we have a nested parallel composition (associated to the left). Our strategy is to remove the initialisations from the parallel composition of all the parallel branches by recursively applying isolateSeqActions. For this reason, we use the tactic $\mathsf{foldr}_{\parallel}$, which is very similar to the previously presented $\mathsf{foldl}_{\parallel}$, but applies to left-associated parallel compositions $((A_1 \parallel A_2) \parallel \ldots) \parallel A_n$ (moving *r*ight).

$$\begin{array}{l}
\textbf{Tactic } \mathsf{BJSt2Proc}(\mathit{pname}) \mathrel{\widehat=} \\
\quad \textbf{program } \langle\!(\mathit{pname}, \boxed{\widehat=} \, (\boxed{\textbf{beginend}} \, (\langle\rangle, \mathsf{foldr}_{\parallel} \, (\textbf{tactic } \mathsf{isolateInitBin}())))) \,)\!\rangle \\
\textbf{end}
\end{array}$$

Using this tactic, we may refine all processes that resulted from joining processes in parallel in the previous step (*Diff_Sd*, *Int_Si*, and *Sp_Sum*). This can be achieved by applying the following tactic to the *Circus* program.

$$\begin{array}{l}
\textbf{Tactic } \mathsf{BJSt2}(\mathit{blocks}) \mathrel{\widehat=} \ \mathsf{APPLYT}(\mathsf{BJSt2Proc}, \mathit{join\_names}(\mathit{blocks})) \\
\textbf{end}
\end{array}$$

This tactic receives a sequence that contains sequences of blocks that are to be joined (for instance, *pid_blocks*) as arguments. It uses the function *join_names* that returns the names of the final processes that resulted from each individual join. The resulting list is used as argument for the application of tactic BJSt2Proc. In our example, the tactic BJSt2Proc is applied three times: one for each of the processes *Diff_Sd*, *Int_Si*, and *Sp_Sum*. For the first one, we have the resulting main action presented below.

$$\begin{array}{l}
\mathit{pid\_Diff\_Init}; \\
\boxed{\left(\begin{array}{l}
\left(\mu X \bullet \left(\left(\begin{array}{l}
\left(\begin{array}{l} E?x \rightarrow \\ \left(\begin{array}{l} (\mathit{pid\_Diff\_In}1 := x;\ \mathit{Calculate\_Diff\_out}); \\ \mathit{Calculate\_Diff\_State} \end{array}\right) \end{array}\right); \\
(\mathit{Diff\_out}!\mathit{pid\_Diff\_Out}1 \rightarrow \mathit{Skip})
\end{array}\right); \mathit{endCycle} \rightarrow X\right)\right) \\
\lbrack\!\lbrack\ \{\mathit{pid\_Diff\_UnitDelay\_state}, \mathit{pid\_Diff\_In}1, \mathit{pid\_Diff\_Out}1\} \\
\quad \mid \{\!\mid \mathit{Diff\_out}, \mathit{end\_cycle} \mid\!\} \mid \\
\quad \{\mathit{pid\_Sd\_In}1, \mathit{pid\_Sd\_In}2, \mathit{pid\_Sd\_Out}1\}\ \rbrack\!\rbrack \\
\left(\mu X \bullet \left(\left(\begin{array}{l}
\left(\begin{array}{l} \mathit{Diff\_out}?x \rightarrow \mathit{pid\_Sd\_In}2 := x \\ \lbrack\!\lbrack\!\lbrack \{\, \mathit{pid\_Sd\_In}2 \,\} \mid \{\, \mathit{pid\_Sd\_In}1 \,\} \rbrack\!\rbrack\!\rbrack \\ \mathit{Kd}?x \rightarrow \mathit{pid\_Sd\_In}1 := x \end{array}\right); \\
(\mathit{pid\_Sd};\ \mathit{Sd\_out}!\mathit{pid\_Sd\_Out}1 \rightarrow \mathit{Skip})
\end{array}\right); \mathit{endCycle} \rightarrow X\right)\right)
\end{array}\right)}
\end{array}$$

We are left with the initialisation of the state components related to the original *Diff* process followed by a parallel composition of two recursive actions, which we intend to transform into a single recursive action in the step that follows.

6.2.3. *Extract the synchronisation on end_cycle*

In this step of the BJ phase of the refinement strategy, we extract the synchronisation on *end_cycle*. For that, we use the fixed-point Law 2 (rec-sync).

Law 2 (rec-sync)

$$(\mu X \bullet A_1;\ c \to X) \llbracket ns_1 \mid \{\!| c |\!\} \cup cs \mid ns_2 \rrbracket (\mu X \bullet A_2;\ c \to X)$$
$$=$$
$$\mu X \bullet (A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2);\ c \to X$$

provided
- $c \notin cs \cup usedC(A_1) \cup usedC(A_2)$
- $wrtV(A_1) \cap usedV(A_2) = \emptyset$
- $wrtV(A_2) \cap usedV(A_1) = \emptyset$

The first proviso ensures that in the parallelism of recursive actions, the channel c is only used at the end of the bodies $A_1;\ c \to X$ and $A_2;\ c \to X$ of each recursion. The set $usedC(A)$ contains the channels used by an action A. The synchronisation on c ensures that the recursions proceed in lock-step. This law states that we can establish a lock-step by considering a single recursive action in which A_1 and A_2 are executed in parallel in each iteration. We use $wrtV(A)$ to refer to the set of variables whose values can potentially be changed by the action A, and $usedV(A)$ to the set of variables that are used by A.

The tactic isolateEC extracts the synchronisation on *end_cycle* from the main action of the processes that implement the groups of blocks. As previously described, these actions may be a sequential composition of a state initialisation followed by a recursive behaviour. However, the state initialisation is not always present. For uniformity, as we did in the tactics NB and isolateInitBin, we include a sequential composition, if needed, using insertSeqComp.

Tactic isolateEC() $\widehat{=}$ **tactic** insertSeqComp() ; (**skip** $\boxed{;}$ (foldr$_{\|}$ (**law** rec-sync())))
end

For the same reason as in the previous step, we may have a nested parallel composition (associated to the left). Hence, we take the same approach as in the previous step: our strategy is to remove the synchronisations on *end_cycle* from the parallel composition of all the parallel branches by recursively using Law 2. For this reason, we use the tactic foldr$_{\|}$. For a given process name, the tactic BJSt3Proc extracts the synchronisation on *end_cycle*. It works on the whole *Circus* programs but only changes, using isolateEC, the main action of the specified process.

Tactic BJSt3Proc($pname$) $\widehat{=}$ **program** $\langle(pname, \boxed{\widehat{=}}$ ($\boxed{\textbf{beginend}}$ $(\langle\rangle$, **tactic** isolateEC()))))$\rangle$
end

The tactic that refines all processes that resulted from joining processes in parallel (*Diff_Sd*, *Int_Si*, and *Sp_Sum*) uses the tactic BJSt3Proc and the names of the blocks that have been joined.

Tactic BjSt3($blocks$) $\widehat{=}$ APPLYT(BJSt3Proc, $join\_names(blocks)$)
end

In our example, the tactic BJSt3Proc is applied to *Diff_Sd*, *Int_Si*, and *Sp_Sum*. For the first one, this application yields the following recursive behaviour after the initialisation.

$$\mu X \bullet \left(\boxed{\begin{array}{l} \left(\begin{array}{l} (\,E?x \to ((pid\_Diff\_In1 := x;\ Calculate\_Diff\_out);\ Calculate\_Diff\_State)\,); \\ (Diff\_out!pid\_Diff\_Out1 \to Skip) \end{array}\right) \\ \llbracket \{pid\_Diff\_UnitDelay\_state, pid\_Diff\_In1, pid\_Diff\_Out1\} \\ \mid \{\!| Diff\_out |\!\} \mid \{pid\_Sd\_In1, pid\_Sd\_In2, pid\_Sd\_Out1\} \rrbracket \\ \left(\left(\begin{array}{l} Diff\_out?x \to pid\_Sd\_In2 := x \\ \llbracket \{\, pid\_Sd\_In2 \,\} \mid \{\, pid\_Sd\_In1 \,\} \rrbracket \\ Kd?x \to pid\_Sd\_In1 := x \end{array}\right);\ (pid\_Sd;\ Sd\_out!pid\_Sd\_Out1 \to Skip)\right) \end{array}}\right);$$
$$end\_cycle \to X$$

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.

We are now left with the task of removing the remaining parallel composition. This is accomplished with the next step.

6.2.4. *Remove Parallelism*

The particular steps required to remove a parallelism depend on the way in which the parallel blocks are arranged. Also, joining parallelism is not always possible: we combine blocks connected in sequence. If we have more than two blocks to combine, we join two at a time. For this reason, the tactic that formalises this step is defined as an alternation of different possibilities, one for each configuration.

```
Tactic BJSt4A_all_config() $\widehat{=}$
    tactic BJSt4A-config1() | tactic BJSt4A-config2()
    | tactic BJSt4A-config3A() | tactic BJSt4A-config3B() | tactic BJSt4A-config4()
end
```

The first one that succeeds leads to the success of the whole tactic.

The configurations presented in Figure 5 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. For these configurations, we proceed as shown below. Configuration (4) involves no internal channels and, therefore, the removal of the parallelism is simpler. For conciseness, we describe below the tactic that formalises this step for our example, configuration 2. The formalisation of the remaining configurations can be found elsewhere [19].

The fourth step of the BJ phase is accomplished in four stages.

A. Evaluate the synchronisation entailed by the internal communications
B. Remove internal communications
C. Sequentialise assignments
D. Introduce interleaving of inputs

In what follows, we describe each one of them separately. Finally, we define the tactic BJSt4 that formalises the whole step 4.

6.2.4.1. *Evaluate the synchronisation entailed by the internal communications.*

In this stage, we use highly specialised, but similar, refinement laws. For our example (that is, configuration 2), we have the following structure in the body of the recursion.

$$(\mathit{TakeInAndCalc};\ \mathit{OutInternalComm}) \parallel ((\mathit{InpInternalComm} \interleave \mathit{InpExternal});\ \mathit{CalcOutAndComm})$$

The main behaviour of the body of the recursion has two parallel branches. The left-hand side action takes the inputs and calculates the new state and outputs (*TakeInAndCalc*) and then it communicates the outputs (*OutInternalComm*). The action on the right-hand side takes the inputs communicated from the other action (*InpInternalComm*) and the external inputs (*InpExternal*) in interleaving. Afterwards, it calculates the state and outputs the values that were calculated.

In our example, we have that both *OutInternalComm* and *InpInternalComm* are simple prefixed actions (on *Diff_out*). That means that the first block has one output (*Diff_out*) and synchronises with *InpInternalComm* on this channel. There may be, however, the case in which there is more than one internal communication. In these cases, both *OutInternalComm* and *InpInternalComm* are an interleaving of prefixed actions. To deal with these cases, we use Law 15 (inter-index) that is based on the definition of indexed interleaving and transforms an explicit interleaving into a indexed one. Simple prefixed actions like in our example are also transformed into an indexed interleaving in which the range of the indexing variable has cardinality one. We apply Law inter-index to *OutInternalComm* and *InpInternalComm*. Only then, we are able to use a generalisation of the Law 24 (par-out-inp-inter-exchange) from [20], which is useful when the first block has one output that synchronises with one of the two interleaved inputs of the second block. The generalised Law 25 (par-out-inp-inter-exchange-n) follows the same principles, but evaluates the multiple synchronisations.

The tactic BJSt4A-config2 transforms both *OutInternal* and *InpInternalComm* into indexed interleaving using Law inter-index, evaluates the synchronisation entailed by the internal communications for configuration (2) using Law par-out-inp-inter-exchange-n, and uses Law inter-index once again to expand the indexed interleaving that resulted from the evaluation.

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{BJSt4A-config2}() \mathrel{\widehat=} \\
\quad ((\textbf{skip} \boxed{;}\ \textbf{law}\ \mathsf{inter\text{-}index}^b()) \boxed{|||} ((\textbf{law}\ \mathsf{inter\text{-}index}^b() \boxed{|||} \textbf{skip}) \boxed{;} \textbf{skip}))\ ; \\
\quad \textbf{law}\ \mathsf{par\text{-}out\text{-}inp\text{-}inter\text{-}exchange\text{-}n}()\ ; \\
\quad ((\textbf{skip} \boxed{;} (\textbf{law}\ \mathsf{inter\text{-}index}() \boxed{|||} \textbf{skip})) \boxed{;} \textbf{skip}) \\
\textbf{end}
\end{array}$$

In our example, the result of applying the tactic BJSt4A-config2 to the first part of the body of the recursion is presented below.

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

As for the previous steps, we have a tactic that formalises the application of the stage A for a specific process. It applies to the overall *Circus* program, but works specifically on the main action of the given process. More precisely, it works on the first action of the sequential composition within the body of the recursion in the main action.

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{BJSt4AProc}(pname) \mathrel{\widehat=} \\
\quad \textbf{program}\ \langle\, (pname, \boxed{\widehat=}\ \boxed{\textbf{beginend}}\ (\,\langle\rangle, \textbf{skip} \boxed{;} \boxed{\mu} (\,\textbf{tactic}\ \mathsf{BJSt4A\_all\_config}() \boxed{;} \textbf{skip}\,)\,))\,\rangle \\
\textbf{end}
\end{array}$$

The tactic that formalises the application of the first stage of this step to the whole processes is presented below. It follows the same patterns as the tactics previously presented.

$$\begin{array}{l}
\textbf{Tactic}\ \mathsf{BjSt4A}(blocks) \mathrel{\widehat=}\ \mathsf{APPLYT}(\mathsf{BJSt4AProc}, join\_names(blocks)) \\
\textbf{end}
\end{array}$$

The next step removes the communications like *Diff_out* that happen internally between blocks.

6.2.5. *Remove Internal Communications*

Before working on each individual process, we need to work on the overall specification to distribute the hiding over the process that describes the overall system specification, in our example, *PID*.

The tactic BjSt4_HidPrep presented below applies to the whole *Circus* program. It receives the name *main* of the main process of the system and works on the body of its definition. First, it applies the Law 22 (par-hid-dist) that applies to processes $((P_1\,\alpha_1) \parallel \ldots \parallel (P_n\,\alpha_n)) \setminus cs$ and distributes the hiding over the parallel composition. The law guarantees that for each one of the parallel processes we hide only the events that are in cs but are not in the interface of the other parallel processes. We remove from cs these events; hence, cs is left only with the events that are shared between parallel processes.

Next, using the tactic $\mathsf{mapr}_{\parallel}$, BjSt4_HidPrep applies Law 7 (hid-contract) to each parallel process. This law reduces the set of hidden events to only those that are actually in the interface of the process.

The tactical $\mathsf{mapr}_{\|}$ applies a tactic to all elements in a nested right-associated parallel composition. It is defined as follows.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{mapr}_{\|}(t) \mathrel{\widehat=}\ \mu X \bullet (t \boxed{\|} X) \mid t\\ \textbf{end}\end{array}$$

If it finds a parallel composition, it applies t to the left branch and recurs its application to the right branch. Finally, at the last branch to the left it will apply t; this is done by the alternative in the tactic.

Finally, the tactic BjSt4_HidPrep applies the Law 4 (join-proc-hid) to the overall *Circus* program. This law states that if a process P is only referenced in the overall program by hiding some of its events $((P\ \alpha_P) \setminus cs)$, then we replace P by a new process P_1 in the *Circus* specification. The new process P_1 is very similar to P, but hide these events in its main action. In the overall *Circus* specification, we may now reference $(P_1\ (\alpha_P \setminus cs))$; the new interface removes cs from the original one.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{BjSt4\_HidPrep}(main) \mathrel{\widehat=}\\ \quad \textbf{program}\ \langle\, (main, \boxed{\widehat=}\ (\textbf{law}\ \mathsf{par\text{-}hid\text{-}dist}()\ ;\ \mathsf{mapr}_{\|}\ (\textbf{law}\ \mathsf{hid\text{-}contract}())))\,\rangle\ ;\\ \quad \mathsf{APPLYL}(\mathsf{join\text{-}proc\text{-}hid}, join\_names(blocks))\\ \textbf{end}\end{array}$$

In our example, we have that the *PID* is the main action and its definition, before the application of the tactic BjSt4_HidPrep is as follows.

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

The application of the tactic BjSt4_HidPrep distributes the hiding over the parallel composition and changes the overall *Circus* program including the definition of *PID*. A sketch of the resulting *Circus* program is presented below.

$$\ldots$$

$$Diff\_Sd_1 \mathrel{\widehat=}\ \ldots \bullet \ldots \setminus \{\!|\ Diff\_out\ |\!\}\ \textbf{end}$$

$$PID \mathrel{\widehat=} \left(\begin{array}{l} Diff\_Sd_1\ \{\!|\ E, Kd, Sd\_out, end\_cycle\ |\!\}\\ \|\ Int\_Si_1\ \{\!|\ E, Ki, Int\_out, end\_cycle\ |\!\}\\ \|\ Sp\_Sum_1\ \{\!|\ E, Kp, Sd\_out, Int\_out, Y, end\_cycle\ |\!\}\end{array}\right) \setminus \{\!|\ Int\_out, Sd\_out\ |\!\}$$

The new definition of $Diff\_Sd$, $Diff\_Sd_1$, hides the internal communication between *Diff* and *Sd* in its main action. Only now, we are able to work on the main action of each process resulting from a join, like $Diff\_Sd$, as informally described in [2].

For each process, we use the distribution laws of hiding to localise the hiding of the channel involved around the prefixing, and apply Law 13 (hid-step) to remove the communication. This distribution is achieved by the tactic HidDistStep, which exhausts the application of the hiding distribution laws over the *Circus* operators and, when it is no longer possible to distribute the hiding, it applies the step Law 13. Finally, it uses

Law 8 (hid-idem) to remove the hiding.

$$\textbf{Tactic}\ \mathsf{HidDistStep}() \mathrel{\widehat=}\ \mu\, X \bullet \mathsf{TRY}\left(\begin{array}{l} (\textbf{law}\ \mathsf{hid\text{-}seq\text{-}dist}()\ ;\ (X\boxed{;}X)) \\ \mid (\textbf{law}\ \mathsf{hid\text{-}par\text{-}dist}()\ ;\ (X\boxed{\parallel}X)) \\ \mid (\textbf{law}\ \mathsf{hid\text{-}inter\text{-}dist}()\ ;\ (X\boxed{|||}X)) \\ \mid (\textbf{law}\ \mathsf{hid\text{-}rec\text{-}dist}()\ ;\ (\boxed{\mu}X)) \\ \mid (\textbf{law}\ \mathsf{hid\text{-}step}()\ ;\ X) \\ \mid (\textbf{law}\ \mathsf{hid\text{-}idem}()) \end{array}\right)$$
$\textbf{end}$

For a given process name, the tactic BJSt4Proc removes the internal communications. It works on the whole *Circus* programs but only changes, using HidDistStep, the main action of the specified process.

$\textbf{Tactic}\ \mathsf{BJSt4BProc}(pname) \mathrel{\widehat=} \textbf{program}\ \langle (pname, \boxed{\widehat=}\,(\boxed{\textbf{beginend}}\,(\langle\rangle, \textbf{tactic}\ \mathsf{HidDistStep}()))))\rangle$
$\textbf{end}$

The tactic that refines all process that resulted from joining two process in parallel ($Diff\_Sd$, Int_Si, and Sp_Sum) uses the tactic BJSt4Proc and the names of the blocks that have been joined.

$\textbf{Tactic}\ \mathsf{BjSt4B}(blocks) \mathrel{\widehat=}\ \mathsf{APPLYT}(\mathsf{BJSt4BProc}, join\_names_1(blocks))$
$\textbf{end}$

The function $join\_names_1$ is very similar to $join\_names$, but sufixes all names with $_1$.

In our example, the application of the tactic BJSt4B removes the communication through $Diff\_out$ in the main action of process $Diff\_Sd$ yielding the main action presented below.

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

If there were several internal communications, then we are left with an interleaving of assignments. The next step aims at sequentialising these assignments.

6.2.6. *Sequentialise Assignments*

We transform the interleaving into a sequence of assignments using the Law 17 (inter-seq-assig). However, this law only needs to be applied to the interleaving of assignments. For this reason, the tactic SeqAssig presented below, recursively searches for such structures and, once it finds an interleaving, it searches for further interleaved assignments, and then, it tries to apply the Law inter-seq-assig.

$\textbf{Tactic}\ \mathsf{SeqAssig}() \mathrel{\widehat=}$
$\quad \mu\, X \bullet \mathsf{TRY}(\,(X\boxed{;}X) \mid (X\boxed{\parallel}X) \mid (\boxed{\rightarrow}X) \mid (\boxed{\mu}X) \mid ((X\boxed{|||}X)\ ;\ \mathsf{TRY}(\textbf{law}\ \mathsf{inter\text{-}seq\text{-}assig}()))\,)$
$\textbf{end}$

The definition of the tactic that implements the application of this step of the refinement strategy follows the same structures as the tactics corresponding to the previous steps. First, we define a tactic BJSt4CProc that applies this step to the main action of a specific process, given its name. Finally, we define a tactic BjSt4C that applies this tactic to each process that resulted from joining two process in parallel. In *Diff_Sd*, there is only one internal communication; the application of BjSt4C leaves the main action unchanged. We are left with the last stage of this step, which introduces an interleaving of inputs.

6.2.7. *Introduce Interleaving of Inputs*

As illustrated by our example, at this stage we are left with an interleaving that may include more than just the inputs; we need to simplify it using the tactic that we present in the sequel. First, let us recall the shape of the main action before this stage. In our example, which falls in the configuration 2, the main action is as follows.

$$I;\ \mu X \bullet \boxed{(((((in_1?x \rightarrow ((v_1 := x;\ A_1);\ A_2));\ A_3) \llbracket ns_1 \mid ns_2 \rrbracket (in_2?x \rightarrow v_2 := x));\ A_4)};\ end\_cycle \rightarrow X$$

This stage focus on the boxed part of the main action.

In the first part, the tactic BjSt4D-config2 works on the action on the left-hand side of the interleaving.

$$(in_1?x \rightarrow ((v_1 := x;\ A_1);\ A_2));\ A_3$$

First, it applies the associativity law to the action prefixed by the input on in_1 to isolate the assignment.

$$(in_1?x \rightarrow (v_1 := x;\ (A_1;\ A_2)));\ A_3$$

Then, it applies the associativity law for prefixing, Law 28 (prefix-seq-assoc), to isolate the prefixed action $in_1?x \rightarrow st_1 := x$.

$$((in_1?x \rightarrow v_1 := x);\ (A_1;\ A_2));\ A_3$$

Next, it applies once again the associativity law for sequence in order to make the isolated prefixed action the left-hand side of a sequential composition.

$$(in_1?x \rightarrow v_1 := x);\ ((A_1;\ A_2);\ A_3)$$

Only then, we are able to use the Law 18 (inter-seq-extract-snd) that removes the processing of inputs from the interleaving, leaving just prefixings of assignments to input variables.

$$(((in_1?x \rightarrow v_1 := x) \llbracket ns_1 \mid ns_2 \rrbracket (in_2?x \rightarrow v_2 := x));\ ((A_1;\ A_2);\ A_3));\ A_4$$

The next part of the tactic works specifically on the interleaving of inputs. It uses the Laws 19 (inter-unused-name) and 14 (inter-comm) to leave in the name sets only the input variables. In our example, we remove *pid_Diff_UnitDelay_state*, *pid_Diff_Out*1, and *pid_Sd_In*2 from the first name set. The second name set already contains only the right input variable. Finally, we isolate the interleaving of inputs as the left-hand side of a sequential composition by applying, once again, the associativity law for sequences.

```
**Tactic** BjSt4D − config2() $\widehat{=}$
    **applies to** $(((in_1?x \rightarrow ((st_1 := x;\ A_1);\ A_2));\ A_3) \llbracket ns_1 \mid ns_2 \rrbracket\ (in_2?x \rightarrow st_2 := x));\ A_4$
    **do**
        $\left(\left(\begin{array}{l}\left(\left(\begin{array}{l}((\boxed{\rightarrow}\ \textbf{law}\ \text{seq-assoc}^b()\ ;\ \textbf{law}\ \text{prefix-seq-assoc}())\ \boxed{;}\ \textbf{skip})\ ;\\ \textbf{law}\ \text{seq-assoc}^b()\end{array}\right)\ \boxed{|||}\ \textbf{skip}\right)\ ;\\ \textbf{law}\ \text{inter-seq-extract-snd}()\ ;\\ \left(\left(\begin{array}{l}\textbf{law}\ \text{inter-unused-name}(\{st_2\})\ ;\ \textbf{law}\ \text{inter-comm}()\\ \textbf{law}\ \text{inter-unused-name}(\{st_1\})\ ;\ \textbf{law}\ \text{inter-comm}()\end{array}\right)\ \boxed{;}\ \textbf{skip}\right)\end{array}\right)\ \boxed{;}\ \textbf{skip}\right)\ ;$
        **law** seq-assoc()
**end**
```

For a given process name, the tactic BJSt4DProc introduces the interleaving of inputs in the main action. It works on the whole *Circus* programs but only changes, using BjSt4D-config2, the main action of the specified process. It, however, works specifically on the action in the left-hand side of the sequential composition within the recursion body.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{BJSt4DProc}(pname) \mathrel{\widehat=} \\ \quad \textbf{program} \left\langle \left(\begin{array}{l} pname, \\ \boxed{\widehat=}\,(\boxed{\textbf{beginend}}\,(\,\langle\rangle, \textbf{skip}\,\boxed{;}\,\boxed{\mu}\,(\,\mathsf{TRY}(\textbf{tactic}\ \mathsf{BjSt4D\text{-}config2}())\,\boxed{;}\,\textbf{skip})\,)) \end{array} \right) \right\rangle \\ \textbf{end}\end{array}$$

Since this step is only required for configuration 2, it uses the tactical TRY. This allows this tactic to be applied in the general case. Its application, however, only changes the main action of blocks that follows the configuration 2, like our example. This application results in the main action presented below.

$$pid\_Diff\_Init;\ \mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (E?x \rightarrow pid\_Diff\_In1 := x) \\ \lbrack\!\lbrack\!\lbrack \{pid\_Diff\_In1\} \mid \{pid\_Sd\_In1\} \rbrack\!\rbrack\!\rbrack \\ (Kd?x \rightarrow pid\_Sd\_In1 := x) \end{array} \right); \\ \left(\begin{array}{l} \left(\begin{array}{l} (Calculate\_Diff\_out;\ Calculate\_Diff\_State); \\ (pid\_Sd\_In2 := pid\_Diff\_Out1) \end{array} \right); \\ (pid\_Sd;\ Sd\_out!pid\_Sd\_Out1 \rightarrow Skip) \end{array} \right) \end{array} \right);\ endCycle \rightarrow X$$

The communication over the channel *Diff_out* has become internal to this process, so it is removed, and replaced with a direct assignment. 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*.

The tactic that refines all process that resulted from joining two process in parallel (*Diff_Sd*, *Int_Si*, and *Sp_Sum*) uses the tactic BJSt4DProc and the names of the blocks that have been joined.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{BjSt4D}(blocks) \mathrel{\widehat=}\ \mathsf{APPLYT}(\mathsf{BJSt4DProc}, join\_names_1(blocks)) \\ \textbf{end}\end{array}$$

We combine all the stages of this step in the tactic presented below that corresponds to the fourth, and final, step of the BJ phase of the refinement strategy.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{BJSt4}(main, blocks) \mathrel{\widehat=} \\ \quad \textbf{tactic}\ \mathsf{BjSt4A}(blocks)\ ;\ \textbf{tactic}\ \mathsf{BjSt4\_HidPrep}(main)\ ; \\ \quad \textbf{tactic}\ \mathsf{BjSt4B}(blocks)\ ;\ \textbf{tactic}\ \mathsf{BjSt4C}(blocks)\ ;\ \textbf{tactic}\ \mathsf{BjSt4D}(blocks) \\ \textbf{end}\end{array}$$

This tactic receives as arguments the name of the *main* process (i.e *PID*), and the sequence that contains sequences of *blocks* that are to be joined. It simply invokes the tactics that corresponds to each one of the stages in sequence.

Finally, we have the tactic that corresponds to the stage BJ of the refinement strategy.

$$\begin{array}{l}\textbf{Tactic}\ \mathsf{BJ}(main, blocks) \mathrel{\widehat=} \\ \quad \textbf{tactic}\ \mathsf{BJSt1}(blocks)\ ;\ \textbf{tactic}\ \mathsf{BJSt2}(blocks)\ ;\ \textbf{tactic}\ \mathsf{BJSt3}(blocks)\ ;\ \textbf{tactic}\ \mathsf{BJSt4}(main, blocks) \\ \textbf{end}\end{array}$$

It also receives the name of the *main* process and the *blocks* sequence as arguments. As in the previous tactic, it simply invokes the tactics that corresponds to each one of the steps in sequence.

In this step, using the tactic BJ above, we obtain the process *Diff_Sd*. Furthermore, we also join the *Circus* processes *Si* and *Int* to produce a process *Si_Int*, and the processes *Sp* and *Sum* to produce a process *Sp_Sum*. Regardless of the difference in the internal structure of these processes, however, the tactic BJ, together with tactic NB, can be applied with success reducing considerably the amount of effort used in the correctness proof of the PID controller.

The remaining of the refinement strategy can be formalised in the same way and is left as future work. This will foster its automatic application using tools like [24].

7. Conclusions

In this paper, we presented ArcAngel*C*, a refinement-tactic language that extends ArcAngel and can be used in the formalisation of refinement strategies for concurrent state-rich programs in *Circus*. Tactics can be used as single transformation rules, and hence, shorten developments. We formalised the first two phases of a refinement strategy proposed in [2] that is used to verify SPARK Ada programs with respect to Simulink diagrams using *Circus*. The approach is based on calculating the *Circus* model of the diagram using the semantics given in [2], calculating a *Circus* model for the SPARK Ada program, and proving that the former is refined by the latter. In this paper, we described the first phases as tactics NB and BJ and used them in the development of a simple PID-controller. The tactics, however, are general enough to apply to the large examples that we find in industrial practice. The formalisation of the verification strategy as tactics of refinement gives clear route to automation.

We also defined the semantics of ArcAngel*C* based on the ArcAngel semantics. As in ArcAngel, a goal is called an *RCell*, which is a pair with a program as its first element and a set of proof obligations as its second element. The application of a tactic to an *RCell* returns a list of *RCell*s containing the possible output programs with their corresponding set of proof obligations. In ArcAngel*C*, however, we need a more general definition since there are different sorts of programs to which transformation rules (refinement laws) can be applied. For instance, we have refinement laws that transform actions, processes, and even *Circus* programs.

In [16], we have shown the soundness of algebraic laws for reasoning about ArcAngel tactics. We covered most of the laws that have been proposed for Angel. For the vast majority of them, proofs are not available in the literature and are provided in [16] in the context of ArcAngel. Since these laws are valid, the strategy proposed to reduce finite Angel tactics to a normal form can be applied to ArcAngel tactics. As future work, we intend to prove that these laws are also valid in the context of ArcAngel*C*. Since it is a natural extension of ArcAngel, it is possible that these proofs will be relatively simple. They remain, however, still to be done.

The formalisation of the semantics presented here used Z as meta-language. For this reason, it will be possible to encode it in a theorem prover like ProofPower-Z and mechanically prove the algebraic laws. Furthermore, the encoding of the semantics can be done in the context of the work presented in [22], where we present the mechanisation of *Circus* in ProofPower-Z. This will allow us to use tactics in the development of *Circus* programs within the theories for *Circus* processes we have developed in ProofPower-Z.

We are currently developing a tool based on the work presented in [32, 25] to provide automated support for the application of the *Circus* refinement calculus [6, 24]. In the near future, we intend to include support for tactics written in ArcAngel*C*; using this extension, one may then specify refinement tactics like those presented in this paper, and apply them just like refinement laws.

Finally, we will complete the formalisation of the refinement strategy for Ada programs. ArcAngel*C* and the tools that we will develop will provide a route for its automated application in industry.

Appendix A. Infinite Lists

We present the model for infinite lists adopted here from [12]. The set of the finite and partial sequences of members of X is defined as

$$PF ::= \mathsf{partial} \mid \mathsf{finite}$$
$$\mathrm{pfseq}[X] == PF \times \mathrm{seq}\, X$$

We define an order $\leq$ on these pairs such that for $a, b : \mathrm{pfseq}\, X$, if a is finite, then $a \leq b$ if, and only if, b is also finite and equal to a. If a is partial, then $a \leq b$ if, and only if, a is a prefix of b.

$$\begin{array}{l}
[X] \\
\hline
\_ \leq \_ : \mathrm{pfseq}[X] \leftrightarrow \mathrm{pfseq}[X] \\
\hline
\forall\, gs, hs : \mathrm{seq}\, X \bullet \\
\quad (\mathsf{finite}, gs) \leq (\mathsf{finite}, hs) \Leftrightarrow gs = hs \\
\quad \land\ (\mathsf{finite}, gs) \leq (\mathsf{partial}, hs) \Leftrightarrow \mathit{false} \\
\quad \land\ (\mathsf{partial}, gs) \leq (\mathsf{finite}, hs) \Leftrightarrow gs\ \mathsf{prefix}\ hs \\
\quad \land\ (\mathsf{partial}, gs) \leq (\mathsf{partial}, hs) \Leftrightarrow gs\ \mathsf{prefix}\ hs
\end{array}$$

A chain of sequences is a set whose elements are pairwise related.

$$\begin{array}{l}
[X] \\
\hline
\mathsf{chain} : \mathbb{P}(\mathbb{P}(\mathrm{pfseq}[X])) \\
\hline
\forall\, c : \mathbb{P}(\mathrm{pfseq}[X]) \bullet \\
\quad c \in \mathsf{chain} \Leftrightarrow (\forall\, x, y : c \bullet x \leq y \lor y \leq x)
\end{array}$$

The set pchain contains all downward closed chains.

$$\begin{array}{l}
[X] \\
\hline
\mathsf{pchain} : \mathbb{P}(\mathsf{chain}[X]) \\
\hline
\forall\, c : \mathsf{chain}[X] \bullet \\
\quad c \in \mathsf{pchain} \Leftrightarrow (\forall\, x : c;\ y : \mathrm{pfseq}[X] \mid y \leq x \bullet y \in c)
\end{array}$$

The set pfiseq contains partial, finite, and infinite list of elements of X, which are prefixed-closed chains of elements in pfseq X.

$$\mathrm{pfiseq}[X] == \mathsf{pchain}[X]$$

The idea is that $\bot = \{(\mathsf{partial}, \langle\,\rangle)\}$, the empty list $[_\infty\ ]_\infty = \{(\mathsf{finite}, \langle\,\rangle)\}$, the finite list $[e_1, e_2, \ldots, e_n]$ is represented by the set containing $\{(\mathsf{finite}, \langle e_1, e_2, \ldots, e_n\rangle)\}$ and all approximations to it. An infinite list is represented by an infinite set of partial approximations to it. The infinite list itself is the least upper bound of such a set.

The definitions of the functions used in this paper are as follows.

(i) The map function maps a function f to each element of a possibly infinite list.

$$\begin{array}{l}
[X, Y] \\
\hline
\_\mathsf{pfmap}\_ : ((X \rightarrow Y) \times \mathrm{pfseq}[X]) \rightarrow \mathrm{pfseq}[Y] \\
\hline
\forall\, xs : \mathrm{seq}\, X;\ f : X \rightarrow Y;\ pf : PF \bullet f\ \mathsf{pfmap}\,(pf, xs) = (pf, (f \circ xs))
\end{array}$$

$$\begin{array}{l}
[X, Y] \\
\hline
\_ * \_ : ((X \rightarrow Y) \times \mathrm{pfiseq}[X]) \rightarrow \mathrm{pfiseq}[Y] \\
\hline
\forall\, c : \mathrm{pfiseq}[X];\ g : X \rightarrow Y \bullet g * c = \{ x : c \bullet g\ \mathsf{pfmap}\, x \}
\end{array}$$

The function *pfmap* maps the function f to the second element of x.

(ii) The distributed concatenation returns the concatenation of all the elements of a possibly infinite list of possibly infinite lists.

$$\begin{array}{l}
[X] \\
\hline
\overset{\infty}{\frown}/ : \mathrm{pfiseq}[\mathrm{pfiseq}[X]] \rightarrow \mathrm{pfiseq}[X] \\
\hline
\forall\, s : \mathrm{pfiseq}[\mathrm{pfiseq}[X]] \bullet \overset{\infty}{\frown}/\, s = \bigsqcup_\infty \{ c : s \bullet \overset{\infty}{\wedge}/\, c \}
\end{array}$$

It uses the function $\overset{\infty}{\wedge}/$, which is the distributed concatenation for pfseq(pfiseq X). The function *cat* is the standard concatenation function for X^*.

$$
\begin{array}{|l}
[X] \\
\overset{\infty}{\frown}/ : \mathrm{pfiseq}[\mathrm{pfiseq}[X]] \rightarrow \mathrm{pfiseq}[X] \\
\hline
\overset{\infty}{\frown}/(\mathsf{finite}, \langle\rangle) = \{\}; \\
\overset{\infty}{\frown}/(\mathsf{partial}, \langle\rangle) = \{(\mathsf{partial}, \langle\rangle)\}; \\
\forall\, g : \mathrm{pfiseq}[X] \bullet \overset{\infty}{\frown}/(\mathsf{finite}, \langle g\rangle) = g \wedge \overset{\infty}{\frown}/(\mathsf{partial}, \langle g\rangle) = g \overset{\infty}{\frown} \{(\mathsf{partial}, \langle\rangle)\}; \\
\forall\, gs, hs : \mathrm{seq}(\mathrm{pfiseq}[X]) \bullet \\
\qquad \overset{\infty}{\frown}/(\mathsf{finite}, gs \frown hs) = (\overset{\infty}{\frown}/(\mathsf{finite}, gs)) \overset{\infty}{\frown} (\overset{\infty}{\frown}/(\mathsf{finite}, hs)) \\
\qquad \wedge\ \overset{\infty}{\frown}/(\mathsf{finite}, gs \frown hs) = (\overset{\infty}{\frown}/(\mathsf{finite}, gs)) \overset{\infty}{\frown} (\overset{\infty}{\frown}/(\mathsf{partial}, hs))
\end{array}
$$

The function $\overset{\infty}{\frown}$ is the concatenation function for possibly infinite lists. Its definition is

$$
\begin{array}{|l}
[X] \\
\_ \overset{\infty}{\frown} \_ : (\mathrm{pfiseq}[X] \times \mathrm{pfiseq}[X]) \rightarrow \mathrm{pfiseq}[X] \\
\hline
\forall\, a, b : \mathrm{pfiseq}[X] \bullet \\
\qquad a \overset{\infty}{\frown} b = \{\, x : a;\ y : b \bullet x \,{}^\wedge\, y \,\}
\end{array}
$$

where the function ${}^\wedge$ is the concatenation function for pfseq X defined as

$$
\begin{array}{|l}
[X] \\
\_ \,{}^\wedge\, \_ : (\mathrm{pfseq}[X] \times \mathrm{pfseq}[X]) \rightarrow \mathrm{pfseq}[X] \\
\hline
\forall\, gs, hs : \mathrm{seq}\, X;\ s : \mathrm{pfseq}[X] \bullet \\
\qquad (\mathsf{finite}, gs) \,{}^\wedge\, (\mathsf{finite}, hs) = (\mathsf{finite}, gs \frown hs) \\
\qquad \wedge\ (\mathsf{finite}, gs) \,{}^\wedge\, (\mathsf{partial}, hs) = (\mathsf{partial}, gs \frown hs) \\
\qquad \wedge\ (\mathsf{partial}, gs) \,{}^\wedge\, s = (\mathsf{partial}, gs)
\end{array}
$$

(iii) The function headl_∞ returns a list containing the first element of a possibly infinite list.

$$
\begin{array}{|l}
[X] \\
\mathsf{headl}_\infty : \mathrm{pfiseq}[X] \rightarrow \mathrm{pfiseq}[X] \\
\hline
\forall\, xs : \mathrm{pfiseq}[X] \bullet \mathsf{headl}_\infty(xs) = \mathsf{take}_\infty\, 1\, xs
\end{array}
$$

It uses the function take_∞ that returns a list containing the first n elements of a possibly infinite list.

$$
\begin{array}{|l}
[X] \\
\mathsf{take}_\infty : \mathbb{N} \rightarrow \mathrm{pfiseq}[X] \rightarrow \mathrm{pfiseq}[X] \\
\hline
\forall\, n : \mathbb{N};\ xs : \mathrm{pfiseq}[X] \bullet \\
\qquad n = 0 \Rightarrow \mathsf{take}_\infty\, n\, xs = [_\infty\,]_\infty \\
\qquad \wedge\ n \neq 0 \Rightarrow \\
\qquad\qquad \mathsf{take}_\infty\, n\, \bot[X] = \bot[X] \\
\qquad\qquad \wedge\ \mathsf{take}_\infty\, n\, [_\infty\,]_\infty = [_\infty\,]_\infty \\
\qquad\qquad \wedge\ \mathsf{take}_\infty\, n\, xs = (\mathsf{head}_\infty\, xs) :_\infty (\mathsf{take}_\infty\, (n-1)\, (\mathsf{tail}_\infty\, xs))
\end{array}
$$

For a possibly infinite list xs, the function head_∞ returns the head of xs.

$$
\begin{array}{|l}
[X] \\
\mathsf{head}_\infty : \mathrm{pfiseq}_1[X] \rightarrow X \\
\hline
\forall\, s : \mathrm{pfiseq}_1[X] \bullet \mathsf{head}_\infty(s) = (\mu\, x : X \mid (\mathsf{partial}, \langle\, x\, \rangle) \in s)
\end{array}
$$

On the other hand, the function tail_∞ returns the tail of xs.

$$\begin{array}{l}
[X] \\
\hline
\mathsf{tail}_\infty : \mathrm{pfiseq}_1[X] \rightarrow \mathrm{pfiseq}[X] \\
\hline
\forall s : \mathrm{pfiseq}_1[X] \bullet \mathsf{tail}_\infty(s) = \{ x : s \mid x.2 \neq \langle\,\rangle \bullet (x.1, \mathit{tail}(x.2)) \}
\end{array}$$

For a pair $p = (a, b)$, we have that $p.1 = a$ and $p.2 = b$.

In [12], the conditional in the set comprehension used in the definition of tail_∞ was not present; nevertheless, by definition $(\mathsf{partial}, \langle\,\rangle)$ is a member of every $\mathrm{pfiseq}_1[X]$. Without the conditional, there would be a undefined value involved since *tail* is not defined for empty sequences.

(iv) The function $\_{}^\circ\_$ applies a possibly infinite list of functions to a single argument.

$$\begin{array}{l}
[X, Y] \\
\hline
\_{}^\circ\_ : (\mathrm{pfiseq}[X \nrightarrow Y] \times X) \rightarrow \mathrm{pfiseq}[Y] \\
\hline
\forall \mathit{fs} : \mathrm{pfiseq}[X \nrightarrow Y];\ f : X \nrightarrow Y;\ x : X \bullet \\
\qquad \bot[X \nrightarrow Y]\,^\circ\, x = [_\infty\,]_\infty \\
\qquad \wedge\ [_\infty\,]_\infty\,^\circ\, x = [_\infty\,]_\infty \\
\qquad \wedge\ (f :_\infty \mathit{fs})\,^\circ\, x = (f\ x) :_\infty (\mathit{fs}\,^\circ\, x)
\end{array}$$

(v) The function Π_∞ is the distributed cartesian product for possibly infinite lists.

Appendix B. Laws of refinement

We use $FV(p)$ to denote the set of free variables of a predicate or expression p. Moreover, we use $\mathbf{L}(n)$ to denote the fact that the **L**ocal action definitions may include references to the action n; the same holds for the **M**ain **A**ction $\mathbf{MA}(n)$. Later references to $\mathbf{L}(A)$ and $\mathbf{MA}(A)$ are the result of substituting the body A of n for some or all occurrences of n in $\mathbf{L}$ and $\mathbf{MA}$.

Law 3 (assign-intro) $w : [\mathit{pre}, \mathit{post}] \sqsubseteq_{\mathcal{A}} x := e$
provided $\mathit{pre} \Rightarrow \mathit{post}[e/x]$

Law 4 (join-proc-hid)

$$\begin{array}{l}
\mathit{CPars}_1\,((P_1\,\alpha_P) \setminus cs) \\
P_1 \mathrel{\widehat=} \mathbf{begin}\ \mathit{PPars}_1\ \mathbf{state}\ P\_\mathit{State} \mathrel{\widehat=} \mathit{SExp}_P\ \mathit{PPars}_2 \bullet \mathit{PAct}\ \mathbf{end} \\
\mathit{CPars}_2\,((P_1\,\alpha_P) \setminus cs) \\
= \\
\mathit{CPars}_1\,((P_2\,(\alpha_P \setminus cs))) \\
P_2 \mathrel{\widehat=} \mathbf{begin}\ \mathit{PPars}_1\ \mathbf{state}\ P\_\mathit{State} \mathrel{\widehat=} \mathit{SExp}_P\ \mathit{PPars}_2 \bullet \mathit{PAct} \setminus cs\ \mathbf{end} \\
\mathit{CPars}_2\,((P_2\,(\alpha_P \setminus cs)))
\end{array}$$

provided
- $P_1 \notin \alpha(\mathit{CPars}_1\,(P_2)) \cup \alpha(\mathit{CPars}_2\,(P_2))$
- $P_2 \notin \alpha(\mathit{CPars}_1\,(P_1)) \cup \alpha(\mathit{CPars}_2\,(P_1))$

Law 5 (join-proc-par)

$CPars_1 \, (P \parallel Q)$
$P \mathrel{\widehat=}$ **begin** $PPars_1$ **state** $P\_State \mathrel{\widehat=} SExp_P \; PPars_2 \bullet PAct$ **end**
$CPars_2 \, (P \parallel Q)$
$Q \mathrel{\widehat=}$ **begin** $QPars_1$ **state** $Q\_State \mathrel{\widehat=} SExp_Q \; QPars_2 \bullet QAct$ **end**
$CPars_3 \, (P \parallel Q)$

$=$

$P\_Q \mathrel{\widehat=}$ **begin state** $P\_Q\_State \mathrel{\widehat=} P\_State \wedge Q\_State$

$PPars_1$

$PPars_2$

$QPars_1$

$QPars_2$

$\bullet \; PAct \; [\![\, \alpha(P\_State) \mid usedC(PAct) \cap usedC(QAct) \mid \alpha(Q\_State) \,]\!] \; QAct$

end
$CPars_1 \, (P\_Q)$
$CPars_2 \, (P\_Q)$
$CPars_3 \, (P\_Q)$

provided
- $P \notin \alpha(Q) \cup \bigcup_i \alpha(CPars_i \, (P\_Q))$
- $Q \notin \alpha(P) \cup \bigcup_i \alpha(CPars_i \, (P\_Q))$
- $\alpha(P\_State) \cap \alpha(Q\_State) = \emptyset$
- $(\alpha(PPars_1) \cup \alpha(PPars_2)) \cap (\alpha(QPars_1) \cup \alpha(QPars_2)) = \emptyset$

Law 6 (copy-rule-action)

begin (**state** S) $(n \mathrel{\widehat=} A)$ $\mathbf{L}(n) \bullet \mathbf{MA}(n)$ **end**
$=$ **begin** (**state** S) $(n \mathrel{\widehat=} A)$ $\mathbf{L}(A) \bullet \mathbf{MA}(A)$ **end**

Law 7 (hid-contract) $(P \, \alpha_P) \setminus cs = (P \, \alpha_P) \setminus cs'$ **where** $cs' = cs \cap \alpha_P$

Law 8 (hid-idem) $A \setminus cs = A$
provided $usedC(A) \cap cs = \emptyset$

Law 9 (hid-inter-dist) $(A_1 \; |\!|\![ns_1 \mid ns_2]\!|\!| \; A_2) \setminus cs_2 = (A_1 \setminus cs_2) \; |\!|\![ns_1 \mid ns_2]\!|\!| \; (A_2 \setminus cs_2)$

Law 10 (hid-par-dist) $(A_1 \; [\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!] \; A_2) \setminus cs_2 = (A_1 \setminus cs_2) \; [\![\, ns_1 \mid cs_1 \mid ns_2 \,]\!] \; (A_2 \setminus cs_2)$
provided $cs_1 \cap cs_2 = \emptyset$

Law 11 (hid-rec-dist) $(\mu \, X \bullet A) \setminus cs = \mu \, X \bullet A \setminus cs$

Law 12 (hid-seq-dist) $(A_1; \, A_2) \setminus cs = (P_1 \setminus cs); \, (P_2 \setminus cs)$

Law 13 (hid-step) $(c \to A) \setminus cs = A \setminus cs$
provided $c \in cs$

Law 14 (inter-comm) $A_1 \; [\![\, ns_1 \mid ns_2 \,]\!] \; A_2 = A_2 \; [\![\, ns_2 \mid ns_1 \,]\!] \; A_1$

Law 15 (inter-index)

$$\begin{array}{l} ||| \; x : \{v_1, \ldots, v_n\} \bullet [\![\, ns(x) \,]\!] \; A(x) \\ = \\ A(v_1) \; |[ns(v_1) \mid \bigcup\{x : \{v_2, \ldots, v_n\} \bullet ns(x)\}]| \; (\ldots (A(v_{n-1}) \; |[ns(v_{n-1}) \mid ns(v_n)]| \; A(v_n))) \end{array}$$

Law 16 (inter-unit) $A \; |[ns_1 \mid ns_2]| \; Skip = A$

Law 17 (inter-seq-assig)

$$v_1 := e_1 \; |[ns_1 \mid ns_2]| \; v_2 := e_2 = v_1 := e_1 ; \; v_2 := e_2$$

provided
- $v_1 \notin \{v_2\} \cup FV(e_2)$
- $v_1 \in ns_1$
- $v_2 \in ns_2$

Law 18 (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
- $usedC(A_2) = \emptyset$
- $usedV(A_2) \cap wrtV(A_3) = \emptyset$
- $wrtV(A_1) \subseteq ns_1$
- $wrtV(A_2) \subseteq ns_1$

Law 19 (inter-unused-name) $A_1 \; |[\{x\} \cup ns_2 \mid ns_3]| \; A_2 = A_1 \; |[ns_2 \mid ns_3]| \; A_2$
provided $x \notin wrtV(A_1)$

Law 20 (join-blocks) **var** $x : T_1 \bullet$ **var** $y : T_2 \bullet A =$ **var** $x : T_1 ; \; y : T_2 \bullet A$

Law 21 (par-comm) $A_1 \; [\![\, ns_1 \mid cs \mid ns_2 \,]\!] \; A_2 = A_2 \; [\![\, ns_2 \mid cs \mid ns_1 \,]\!] \; A_1$

Law 22 (par-hid-dist)

$$\begin{array}{l} ((P_1 \, \alpha_1) \parallel (P_2 \, \alpha_2) \parallel \ldots \parallel (P_n \, \alpha_n)) \setminus cs \\ = \\ (((P_1 \, \alpha_1) \setminus cs_1) \parallel ((P_2 \, \alpha_2) \setminus cs_2) \parallel \ldots \parallel ((P_n \, \alpha_n) \setminus cs_n)) \setminus cs' \end{array}$$

where
- $\forall \, i : 1 \mathinner{..} n \bullet cs_i = cs \setminus (\alpha_i \cap (\bigcup_{j:(1..n)\setminus\{i\}} \alpha_j))$
- $cs' = cs \setminus \bigcup_{i:1..n} cs_i$

Law 23 (par-inter-2) $A_1 \; |[ns_2 \mid ns_2]| \; A_2 = A_1 \; [\![\, ns_2 \mid \emptyset \mid ns_2 \,]\!] \; A_2$

Law 24 (par-out-inp-inter-exchange)

$$\begin{array}{l} (A_1 ; \; c_1!v \rightarrow Skip) \; [\![\, ns_1 \mid \{\!| \, c_1 \, |\!\} \mid ns_2 \,]\!] \; ((c_1?x \rightarrow A_2(x) \; |[ns_3 \mid ns_4]| \; A_3) ; \; A_4) \\ = \\ ((A_1 ; \; c_1!v \rightarrow A_2(v)) \; |[ns_1 \cup ns_3 \mid ns_4]| \; A_3) ; \; A_4 \end{array}$$

provided
- $c_1 \notin initials(A_3)$
- $c_1 \notin \bigcup_{i:1..4} usedC(A_i)$
- $ns_3 \cup ns_4 \subseteq ns_2$
- $wrtV(A_1) \subseteq ns_1$
- $wrtV(A_2) \subseteq ns_3$
- $wrtV(A_4) \subseteq ns_2$

Law 25 (par-out-inp-inter-exchange-n)

$$\begin{array}{l}(B_1;\ (\interleave\ i : 1\ ..\ n \bullet \llbracket ns_i \rrbracket\ (c_i!v_i \rightarrow \mathit{Skip})))\\ \llbracket ns_1 \mid \{\!|\ c_1, \ldots, c_n\ |\!\} \mid ns_2 \rrbracket\\ (((\interleave\ i : 1\ ..\ n \bullet \llbracket ns_i \rrbracket\ (c_i?x \rightarrow A_i(x)))\ \llbracket ns_3 \mid ns_4 \rrbracket\ B_2);\ B_3)\\ =\\ (B_1;\ ((\interleave\ i : 1\ ..\ n \bullet \llbracket ns_i \rrbracket\ (c_i!v_i \rightarrow A_i(v_i)))\ \llbracket ns_1 \cup ns_3 \mid ns_4 \rrbracket\ B_2));\ B_3\end{array}$$

provided
- $\{c_1, \ldots, c_n\} \cap \mathit{initials}(B_2) = \emptyset$
- $\{c_1, \ldots, c_n\} \cap (\mathit{usedC}(B_1) \cup \mathit{usedC}(B_2) \cap \mathit{usedC}(B_3) \cup (\bigcup_{i:1..n} \mathit{usedC}(A_i))) = \emptyset$
- $ns_3 \cup ns_4 \subseteq ns_2$
- $\mathit{wrtV}(B_1) \subseteq ns_1$
- $\bigcup_{i:1..n} \mathit{wrtV}(A_i) \subseteq ns_3$
- $\mathit{wrtV}(B_3) \subseteq ns_2$

Law 26 (par-seq-step)

$$(A_1;\ A_2)\ \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ A_3 = A_1;\ (A_2\ \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ A_3)$$

provided
- $\mathit{usedC}(A_1) = \emptyset,\ \mathit{usedV}(A_3) \cap \mathit{wrtV}(A_1) = \emptyset$
- $\mathit{wrtV}(A_1) \subseteq ns_1 \cup ns_1'$

Law 27 (par-seq-step-2)

$$\mathbf{var}\ d \bullet (A_1;\ A_2)\ \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ (A_1;\ A_3) = \mathbf{var}\ d \bullet A_1;\ (A_2\ \llbracket\, ns_1 \mid cs \mid ns_2 \,\rrbracket\ A_3)$$

provided
- $\mathit{usedC}(A_1) \subseteq cs$
- $\mathit{wrtV}(A_1) \subseteq \alpha(d)$

Law 28 (prefix-seq-assoc)

$$c \rightarrow (A_1;\ A_2) = (c \rightarrow A_1);\ A_2$$

provided $FV(A_2) \cap \alpha(c) = \emptyset$

The reference to $\mathbf{L}(\_)$ 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, as x is a state component.

Law 29 (prom-var-state)

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

Law 30 (prom-var-state-2)

$$\begin{array}{l}\mathbf{begin}\ \mathbf{L}(x : T) \bullet (\mathbf{var}\ x : T \bullet \mathbf{MA})\ \mathbf{end}\\ = \mathbf{begin}\ (\mathbf{state}\ [\, x : T \,])\ \mathbf{L}(\_) \bullet \mathbf{MA}\ \mathbf{end}\end{array}$$

Law 31 (seq-assoc) $A_1;\ (A_2;\ A_3) = (A_1;\ A_2);\ A_3$

Law 32 (seq-left-unit) $A = \mathit{Skip};\ A$

Law 33 (seq-right-unit) $A = A;\ \mathit{Skip}$

Law 34 (var-exp-par)

$$(\mathbf{var}\ d : T \bullet A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = (\mathbf{var}\ d : T \bullet A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)$$

provided $\{ d, d' \} \cap FV(A_2) = \emptyset$

Law 35 (var-exp-par-2)

$$(\mathbf{var}\ d \bullet A_1) \llbracket ns_1 \mid cs \mid ns_2 \rrbracket (\mathbf{var}\ d \bullet A_2) = (\mathbf{var}\ d \bullet A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2)$$

Law 36 (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 37 (var-exp-seq) $A_1;\ (\mathbf{var}\ x : T \bullet A_2);\ A_3 = (\mathbf{var}\ x : T \bullet A_1;\ A_2;\ A_3)$
provided $\{ x, x' \} \cap (FV(A_1) \cup FV(A_3)) = \emptyset$

References

[1] R. J. R. Back and J. von Wright. Refinement Concepts Formalised in Higher Order Logic. *Formal Aspects of Computing*, **2**:247–274, 1990.

[2] A. L. C. Cavalcanti and P. Clayton. Verification of Control Systems using *Circus*. In *Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems*, pages 269 – 278. IEEE Computer Society, 2006.

[3] A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing*, **15**(2–3):146–181, 2003.

[4] C. Fischer. How to combine Z with a process algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, *ZUM '98: Proceedings of the 11th International Conference of Z Users on The Z Formal Specification Notation*, pages 5–23. Springer-Verlag, 1998.

[5] L. Groves, R. Nickson, and M. Utting. A Tactic Driven Refinement Tool. In C. B. Jones, R. C. Shaw, and T. Denvir, editors, *5th Refinement Workshop*, Workshops in Computing, pages 272–297. Springer-Verlag, 1992.

[6] A. C. Gurgel, C. G. de Castro, and M. V. M. Oliveira. Tool Support for the *Circus* Refinement Calculus. In J. P. Bowen E. Brger, M. Butler and P. Boca, editors, *ABZ Conference*, volume **5238** of *Lecture Notes in Computer Science*, page 349. Springer-Verlag, 2008.

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

[8] Brian R. Hunt, Ronald L. Lipsman, and Jonathan M. Rosenberg. *A guide to MATLAB: for beginners and experienced users.* Cambridge University Press, New York, NY, USA, 2001.

[9] He Jifeng, C. A. R. Hoare, and J. W. Sanders. Data Refinement Refined. In E. Robinet and R. Wilhelm, editors, *ESOP'86 European Symposium on Programming*, volume **213** of *Lecture Notes in Computer Science*, pages 187–196, March 1986.

[10] Saunders Mac Lane. *Categories for the Working Mathematician.* Graduate Texts in Mathematics. Springer, 1991.

[11] B. P. Mahony and J. S. Dong. Blending Object-Z and Timed CSP: an Introduction to TCOZ. In K. Torii, K. Futatsugi, and R. A. Kemmerer, editors, *The 20th International Conference on Software Engineering (ICSE'98)*, pages 95–104. IEEE Computer Society Press, 1998.

[12] A. Martin. Infinite Lists for Specifying Functional Programs in Z. Technical report, University of Queensland, Queensland - Australia, March 1995.

[13] A. P. Martin, P. H. B. Gardiner, and J. C. P. Woodcock. A Tactical Calculus. *Formal Aspects of Computing*, **8**(4):479–489, 1996.

[14] C. Morgan. *Programming from Specifications.* Prentice-Hall, 1994.

[15] C. Morgan and P. H. B. Gardiner. Data refinement by calculation. *Acta Informatica*, **27**(6):481–503, 1990.

[16] M. V. M. Oliveira. Tactics of refinement. Technical report, Centro de Informática - Universidade Federal de Pernambuco, Pernambuco - Brazil, December 2000. At http://www.cs.york.ac.uk/˜marcel/gabriel/.

[17] M. V. M. Oliveira. *ArcAngel: a Tactic Language for Refinement and its Tool Support*. Master's thesis, Centro de Informática – Universidade Federal de Pernambuco, Pernambuco, Brazil, 2002. At http://www.ufpe.br/sib/.

[18] M. V. M. Oliveira. *Formal Derivation of State-Rich Reactive Programs using Circus*. PhD thesis, Department of Computer Science, University of York, 2006. YCST-2006/02.

[19] M. V. M. Oliveira. ArcAngel*C*. Technical report, Departamento de Informática e Matemática Aplicada - Universidade Federal do Rio Grande do Norte, Natal, Brazil, February 2007.

[20] M. V. M. Oliveira and A. L. C. Cavalcanti. ArcAngelC: a Refinement Tactic Language for *Circus*. *Electronic Notes in Theoretical Computer Science*, **214C**:203 – 229, 2008.

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

[22] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. Unifying theories in ProofPower-Z. In S. Dunne and B. Stoddart, editors, *UTP 2006: First International Symposium on Unifying Theories of Programming*, volume **4010** of *Lecture Notes in Computer Science*, pages 123–140. Springer-Verlag, 2006.

[23] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 2008. DOI 10.1007/s00165-007-0052-5.

[24] M. V. M. Oliveira, A. C. Gurgel, and C. G. de Castro. Tool Support for the *Circus* Refinement Calculus. In *6th IEEE International Conferences on Softwar Engineering and Formal Methods*. IEEE Computer Society Press, 2008. To Appear.

[25] M. V. M. Oliveira, M. Xavier, and A. L. C. Cavalcanti. Refine and Gabriel: Support for Refinement and Tactics. In Jorge R. Cuellar and Zhiming Liu, editors, *2nd IEEE International Conference on Software Engineering and Formal Methods*, pages 310–319. IEEE Computer Society Press, Sep 2004.

[26] A. C. A. Sampaio, J. C. P. Woodcock, and A. L. C. Cavalcanti. Refinement in *Circus*. In L. Eriksson and P. A. Lindsay, editors, *FME 2002: Formal Methods—Getting IT Right*, volume **2391** of *Lecture Notes in Computer Science*, pages 451–470. Springer-Verlag, 2002.

[27] J. M. Spivey. *The Z Notation: A Reference Manual.* Prentice-Hall, 2nd edition, 1992.

[28] H. Treharne and S. Schneider. Using a process algebra to control B operations. In K. Araki, A. Galloway, and K. Taguchi, editors, *Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456. Springer, June 1999.

[29] T. Vickers. A language of refinements. Technical Report TR-CS-94-05, Computer Science Department, Australian National University, 1994.

[30] J. von Wright. Program Refinement by Theorem Prover. In D. Till, editor, *6th Refinement Workshop*, Workshops in Computing, pages 121–150, London, 1994. Springer-Verlag.

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

[32] M. A. Xavier, A. L. C. Cavalcanti, and A. C. A. Sampaio. Type Checking *Circus* Specifications. In A. M. Moreira and L. Ribeiro, editors, *SBMF 2006: Brazilian Symposium on Formal Methods*, pages 105 – 120, 2006.