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 ArcAngelC, 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 ArcAngelC, 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, ArcAngelC 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 ArcAngelC, we adapted the Angel approach to refinements. The constructs of ArcAngelC 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 ArcAngelC. In this paper, besides an informal introduction to the language, we also present the formalisation of ArcAngelC’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 ArcAngelC algebraic laws. The semantics of ArcAngelC 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 ArcAngelC semantics; a full account on the ArcAngelC semantics can be found elsewhere [19].

In order to illustrate the usefulness of ArcAngelC 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 ArcAngelC is currently in progress.

The next section describes Circus. In Section 3, our tactic language for Circus, ArcAngelC, 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 ArcAngelC 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.

channel store, add, out : N; result, reset
process Register ≜ begin state RegSt ≜ [value : N]
RegCycle ≜ store?newValue → value := newValue
□ add?newValue → value := value + newValue
□ result → out!value → Skip
□ reset → value := 0
• value := 0; (µ X • RegCycle; X)
end
Channel declarations channel $c : T$ introduce a channel $c$ that communicates values of type $T$. For instance, channel store, add, out : 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 $p \land c!x \rightarrow A$ 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 $A_1 \parallel [ns_1 | cs | ns_2] A_2$, 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 $A_1 \parallel [ns_1 | ns_2] \parallel A_2$ 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)** $A_1 \parallel [ns_1 | cs | ns_2] A_2 = A_1 \parallel [ns_1 | ns_2] A_2$

provided $(\text{usedC}(A_1) \cup \text{usedC}(A_2)) \cap cs = \emptyset$

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, $(\exists P_1.STATE ; P_1.STATE' \cdot P_1.Act ) \sqsubseteq \mathcal{A} (\exists P_2.STATE ; P_2.STATE' \cdot P_2.Act )$

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].
TacticDecl ::= Tactic N (Decl) Tactic  
[generates Prog]  
[proof obligations Pred+ end]

Tactic ::= law N (Exp∗)  
[law application]

| tactic N (Exp∗)  
| skip | fail | abort  
| applies to Prog do Tactic

| Tactic ; Tactic | Tactic | Tactic  
| μ N • Tactic | ! Tactic

| success Tactic | fails Tactic

| [ ] Tactic | [ ] Tactic

| [ ] Tactic  
| Tactic+ | Tactic

| val Tactic | res Tactic | vres Tactic

| beginend | (N, Tactic)∗, Tactic  

| [ ] Tactic | [ ] Tactic

| Tactic| Tactic

| Tactic| Tactic

| [] Tactic | [ ] Tactic | [ ] Tactic | [] Tactic

| [ ] Tactic | [ ] Tactic | [ ] Tactic | [ ] Tactic

| [ ] Tactic | [ ] Tactic

| program (N, Tactic)∗  

Fig. 1. Abstract Syntax of ArcAngelC

3. ArcAngelC

ArcAngelC 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 ArcAngelC are inherited from Angel, and some of its structural combinators are inherited from ArcAngel; nevertheless, the ArcAngelC’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, ArcAngelC’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 ArcAngelC 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].

4
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 \( n(a) \) \( t \)` 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 \( n(a) \) \( p \)`. 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 \( n(a) \)`, 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. Tactics

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 \( A \parallel [ns_1 \mid cs \mid ns_2] \parallel \text{Skip} \) characterises those parallel compositions whose right-hand action is \( \text{Skip} \); 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 \( A \parallel [ns_1 \mid cs \mid ns_2] \parallel \text{Skip} \) 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 \( \text{Skip} \) and returns the action \( A \) and the proof obligation originated from the application of `inter-unit`.

\[
\text{Tactic \quad interIntroAndSimpl( ) } \equiv \\
\text{applies to } A \parallel [ns_1 \mid cs \mid ns_2] \parallel \text{Skip} \\
\text{do law par-inter(); \ law inter-unit() generates } A \\
\text{proof obligations } usedC(A) \cap cs = \emptyset
\]

Tactics may also be combined as alternatives: \( t_1 | t_2 \). 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.

\textbf{Tactic} promoteVars() \equiv \text{law \ prom-var-state()} | \text{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 \( ! \) behaves like \( t \), except that it returns the first successful application of \( t \). If a subsequent tactic application fails, the whole tactic fails.

\textbf{ArcAngelC} has a fixed-point operator that allows us to define recursive tactics. Using \( \mu \), we can define a tactic like the one below that exhaustively applies a given tactic \( t \), terminating with success when its application fails.

\textbf{Tactic} \text{EXHAUST}(t) \equiv \mu X \cdot (t; X \mid \text{skip})

end

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

Two tactics are used to assert the outcome of applying a tactic. The tactic \( \text{succs} \) fails whenever \( t \) fails, and behaves like \text{skip} whenever \( t \) succeeds. On the other hand, \( \text{fails} \) \( t \) behaves like \text{skip} if \( t \) fails, and fails if \( t \) succeeds. If the application of \( t \) runs indefinitely, then these tacticals behave like \text{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 \text{succs}([\text{law \ par-com()}]).

3.4. Structural Combinators

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

The \textbf{Action Structural Combinators} are the ones that allow us to apply a tactic to parts of a \textit{Circus} action. The first one we present allows us to apply a tactic to an action prefixed by an event. The tactic \texttt{[} \texttt{a} \texttt{]} \texttt{t} applies to actions of the form \( e \to A \). It returns the prefixing \( e \to B \), 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 \texttt{[} \texttt{a} \texttt{]} \texttt{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 \( \mu X \cdot A(X) \), there is the structural combinator \texttt{[} \texttt{a} \texttt{]} \texttt{t} ; it returns recursion obtained by applying \( t \) to \( A(X) \).

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

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

The \textbf{Process Structural Combinators} are those combinators that can be applied only to processes bodies. The only \textit{Circus} constructs that are particular to process are the explicit processes definitions (enclosed by the keywords \texttt{begin} and \texttt{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 \texttt{[} \texttt{a} \texttt{]} \texttt{begin} \texttt{end} \texttt{). This combinator receives two arguments: a possibly-empty sequence of pairs \((n, t)\) of names
we apply like ProofPower-Z and Z-Eves.

embedded the syntax of general type of refinement cell, which can have programs, processes, or actions as its first element. First we to handle different sorts of tactic (action tactics, data refinement tactics, and process tactics), we define a of the Circus definitions. This derives from the fact that definitions from [21]. The structural combinators however, although following the approach from [21], use basic tactics, the pattern matching operator, sequence, alternative, cut, recursion, and assertions follow the a refinement-tactic language for sequential programs. The constructors for law and tactic application, the tactical graphs of a Circu.

choices, obligations composition; CSP constructs that can be used in the definition of both processes and actions; therefore, for each of these constructs we define a 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 $t_1 \parallel t_2$ 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 $\langle \text{skip} \mid \text{tactic interIntroAndSimpl} \rangle$ (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 ($t_1 \mid \mid t_2$), parallel composition ($t_1 \parallel \parallel t_2$), interleaving ($t_1 \parallel_2 t_2$), event hiding ($\{t\} t$), and renaming ($\{n\} t$).

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, $\{ t \}$ can be applied to an indexed sequential composition; $\text{decl} \bullet \text{body}$: the result is that obtained by the application of $t$ to $\text{body}$. 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; $i : 0 \ldots \#sq \bullet s : [s' = s + sq[i]]$. If we apply $\{ \text{law assign-intro}(s := s + sq[i]) \}$, we get the program; $i : 0 \ldots \#sq \bullet s := s + sq[i]$ and proof obligations $\text{true} \Rightarrow s + sq[i] = s + sq[i]$, for every $i$ in $0 \ldots \#sq$.

As for indexed sequential composition, we have $\{ \}$ for indexed external choices, $\llbracket \}$ for indexed internal choices, $\llbracket \$ for indexed parallel composition, and $\llbracket \$ 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 ArcAngelC, 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 ArcAngelC semantics, when compared to the ArcAngel work.

As opposed to the ArcAngel semantics, the formalisation of ArcAngelC 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.

\[
Cell ::= ParActC\langle\langle ParAct\rangle\rangle \mid ParProcC\langle\langle ParProc\rangle\rangle \mid ProcParC\langle\langle seq ProcPar\rangle\rangle \mid ProgC\langle\langle seq ProgPar\rangle\rangle
\]

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.

\[
ParProcCell == ran ParProcC
\]

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.

\[
Refinement ::= ActRefinement\langle\langle ActBody \times ActBody\rangle\rangle \mid ProcRefinement\langle\langle ProcBody \times ProcBody\rangle\rangle \mid ProgRefinement\langle\langle Program \times Program\rangle\rangle \mid DataRefinement\langle\langle SchemaExp \times Decl \times ActBody \times ActBody\rangle\rangle
\]

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

\[
PObs == iseq Pred \times iseq Refinement
\]

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

\[
RCell == Cell \times PObs
\]

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.

\[
Law == \{ L : Cell \rightarrow RCell \mid L\langle ParActCell \rangle \subseteq ParActRCell \wedge L\langle ParProcCell \rangle \subseteq ParProcRCell \wedge L\langle ProcParCell \rangle \subseteq ProcParRCell \wedge L\langle ProgCell \rangle \subseteq ProgRCell \}
\]

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 *RCells* 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.

\[
Tactic == \{ T: RCell \rightarrow pfiseq[RCell] \mid T\langle ParActRCell \rangle \subseteq pfiseq[ParActRCell] \wedge T\langle ParProcRCell \rangle \subseteq pfiseq[ParProcRCell] \wedge T\langle ProcParRCell \rangle \subseteq pfiseq[ProcParRCell] \wedge T\langle ProgRCell \rangle \subseteq pfiseq[ProgRCell] \}
\]

The type pfiseq*[RCell]* is that of possibly infinite lists of *RCells*. 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 \( \text{env}_L \) and a given law name \( n \), we have that \( \text{env}_L(n) \) is also a partial function that relates all valid arguments of \( n \) (sequence of terms) to yet another function, a \text{Law}.

\[
\text{LEnv} == N \mapsto (\text{seq TERM} \mapsto \text{Law})
\]

Similarly, a tactic environment is a function that takes a tactic name and a sequence of arguments, and returns a \text{Tactic}.

### 4.1. Legacy Tactics

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

\[
\text{law}: (N \times (\text{seq TERM})) \rightarrow \text{LEnv} \rightarrow \text{Tactic}
\]

\[
\forall n : N; \text{args} : \text{seq TERM}; r : \text{RCell}; \text{lenv} : \text{LEnv} \bullet
\text{law}(n, \text{args}) \text{lenv} r =
\begin{align*}
\text{if } & (n \in \text{dom(lenv)} \land \text{args} \in \text{dom}(\text{lenv}(n)) \land r.1 \in \text{dom}((\text{lenv}(n))(\text{args}))) \\
\text{then } & [\infty((\text{lenv}(n))(\text{args})(r.1)), \text{MPObs}(r.2, (\text{lenv}(n))(\text{args})(r.1), 2))]_{\infty} \\
\text{else } & [\infty \infty]
\end{align*}
\]

If the law name \( n \) is in the given law environment \( \text{lenv} \), and if the arguments \( a \) and cell \( r.1 \) are appropriate, then the tactic succeeds, and returns a list with a new \( \text{RCell} \). 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 \( \text{RCell} \). Otherwise, the tactic fails with the empty list as result. The function \( \text{MPObs} \) merges two sets of proof obligations. The brackets subscripted with \( \infty \) indicate that this is a possibly infinite list (\text{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 \( \text{tactic} \) \( n(a) \) is similar to that of the \( \text{law} \) construct. The tactic \( \text{skip} \) returns its argument unchanged.

\[
\text{skip} : \text{Tactic}
\]

\[
\forall r : \text{RCell} \bullet \text{skip}(r) = [\infty r]_{\infty}
\]

The tactic \( \text{fail} \) 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.

\[
:: (\text{Tactic} \times \text{Tactic}) \rightarrow \text{Tactic}
\]

\[
\forall t1, t2 : \text{Tactic}; r : \text{RCell} \bullet (t1 ; t2)(r) = \infty[\text{RCell}](t2 \ast (t1(r)))
\]

For a total function \( f : A \rightarrow B, f* : \text{pfiseq} A \rightarrow \text{pfiseq} B \) is the map function that operates on a list by applying \( f \) to each of its elements; the operator \( \infty \) 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 \( (t1 | t1) \), cut \( (!t) \), recursion \( \mu X \bullet t \) , \( \text{abort} \), assertions \( \text{succs} t \) and \( \text{fails} t \), and pattern matching are also very similar to those from [21] and are omitted here for conciseness. A full account on the semantics of \text{ArcAngelC} 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, and 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 $\Omega$ functions. The semantics of the vast majority of the structural combinators has the following structure.

$$(\text{structComb tacs}) \, rc = (\Omega \text{ args}) \ast \text{tacApp}$$

In this template, $\text{structComb}$ stands for the structural combinator, $\text{tacs}$ 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 $\text{tacApp}$ above) and maps (*) a given $\Omega$ function using the arguments required by this function (like the original Circus construct) to the list $\text{tacApp}$ resulting from the tactic application.

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

$$\Omega_D^R : X \rightarrow ((X \times D) \rightarrow R) \rightarrow RCell \rightarrow RCell$$

They receive an element $x$ of type $X$, a function $f : (X \times D) \rightarrow R$, 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 $\Omega$ 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 $f : (X \times D \times D) \rightarrow R$; this is denoted in the name of the function by subscripting the $D$ with a 2 as in $\Omega_D^R$. 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 $\Omega$ function, $\Omega_{\text{ActBody}}$, 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 $f : (X \times \text{ActBody}) \rightarrow \text{ActBody}$, 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.

$$\Omega_{\text{ActBody}} : X \rightarrow ((X \times \text{ActBody}) \rightarrow \text{ActBody}) \rightarrow \text{RCell} \rightarrow \text{RCell}$$

$$\forall x : X; f : (X \times \text{ActBody}) \rightarrow \text{ActBody}; a : \text{ActBody}; pobs : \text{PObs} \\
\Omega_{\text{ActBody}} \, x \, f \, (\text{ParActC}(\text{BaseAct}(a)), pobs) = (\text{ParActC}(\text{BaseAct}(f(x, a))), pobs)$$

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

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

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

Using this $\Omega$ function, we can define the structural combinator used for prefixing. As already explained, the structural combinator $\Box t$ 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 $\Omega_{\text{ActBody}}$ with arguments $c$ and $\rightarrow$.

$$\forall t : \text{Tactic} \quad c : \text{Comm} \quad a : \text{ActBody} \quad \text{pobs} : \text{PObs} \quad (\Box t)(\text{ParActC}(\text{BaseAct}(\rightarrow (c, a))), \text{pobs}) = (\Omega_{\text{ActBody}}(c)(\rightarrow)) * (t(\text{ParActC}(\text{BaseAct}(a)), \text{pobs}))$$

Basically, for every Circus action construct that is defined as a function $f : (X \times \text{ActBody}) \rightarrow \text{ActBody}$, 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 $\Omega_{\text{ActBody}}$. This includes the semantics of structural combinators for guarded action ($\Box x$), recursion ($\Box \mu x$), variable blocks ($\Box \var$), and parametrised actions ($\Box \text{val}$, $\Box \text{res}$, and $\Box \text{vres}$). 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 ($\Box \text{beginend}$ and $\Box \text{vecnd}$) and the one that applies to a process body ($\Box \text{beginend}$. 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 $\Box x$, uses the function $\Omega_{\text{ParProc}}$, which is very similar to $\Omega_{\text{ActBody}}$, but applies to parametrised processes instead.

$$\forall x : X \quad f : (X \times \text{ParProc}) \rightarrow \text{ParProc} \quad p : \text{ParProc} \quad \text{pobs} : \text{PObs} \quad (\Omega_{\text{ParProc}} x f(\text{ParProcC}(p), \text{pobs}) = (\text{ParProcC}(f(x, p)), \text{pobs}))$$

This function is used in the following definition.

$$\forall t : \text{Tactic} \quad p : \text{ParProc} \quad d : \text{Decl} \quad \text{pobs} : \text{PObs} \quad (\Box t)(\text{ParProcC}(\circ(d, p)), \text{pobs}) = (\Omega_{\text{ParProc}}(d)(\circ)) * (t(\text{ParProcC}(p), \text{pobs}))$$

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

The structural combinator $\Box \text{beginend}$ 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 \( \Omega_A \) and rebuilding process \( \Omega_P \)).

\[
\begin{align*}
(\text{structComb}_P \text{ tacs}) \text{ rc} &= (\Omega_P \text{ args}) \ast \text{ tacApp} \\
&\land (\text{structComb}_A \text{ tacs}) \text{ rc} &= (\Omega_A \text{ args}) \ast \text{ tacApp}
\end{align*}
\]

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. It uses two \( \Omega \) functions. The first one receives a generic argument \( x : X \), a function \( f \) of type \((X \times \text{ParAct}) \rightarrow \text{ProcPar}\), and a refinement cell that contains a parameterised action \( a \) and returns a cell that contains a process paragraph resulting from the application of \( f \) to \((x, a)\). In our work, \( \text{ParAct} \) and \( \text{ProcPar} \) are the syntactic classes of parametrised actions and process paragraphs, respectively.

\[
\begin{align*}
\text{ProcPar} : X &\rightarrow ((X \times \text{ParAct}) \rightarrow \text{ProcPar}) \rightarrow \text{RCell} \rightarrow \text{RCell} \\
\forall x : X; f : (X \times \text{ParAct}) \rightarrow \text{ProcPar}; a : \text{ParAct}; \text{pobs} : \text{PObs} \bullet \\
\Omega^\text{ProcPar}_{x f (\text{ParAct}(a), \text{pobs})} &= (\text{ProcPar}(f(x, a)), \text{pobs})
\end{align*}
\]

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

\[
\begin{align*}
\text{ProgPar} : X &\rightarrow ((X \times \text{ParProc}) \rightarrow \text{ProgPar}) \rightarrow \text{RCell} \rightarrow \text{RCell} \\
\forall x : X; f : (X \times \text{ParProc}) \rightarrow \text{ProgPar}; p : \text{ParProc}; \text{pobs} : \text{PObs} \bullet \\
\Omega^\text{ProgPar}_{x f (\text{ParProc}(p), \text{pobs})} &= (\text{ProgPar}(f(x, p)), \text{pobs})
\end{align*}
\]

The structural combinator that applies to an action or process definition, \( \square \), is defined in the way described above. If applied to a process definition \text{process} \( n[\text{gen}] \equiv p \) (denoted by \text{process}((n, \text{gen}), p) in our embedding of the \text{Circus} syntax), where \([\text{gen}]\) 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 \( n \equiv a \) (denoted by \text{ActDef}((n, a)) in our syntactic embedding), it applies the tactic to the action body and reconstructs the action definition.

\[
\begin{align*}
\square : \text{Tactic} &\rightarrow \text{Tactic} \\
\forall t : \text{Tactic}; n : N; \text{gen} : \text{seq} N; a : \text{ParAct}; p : \text{ParProc}; \text{pobs} : \text{PObs} \bullet \\
(\square t)(\text{ProcC}((\text{process}((n, \text{gen}), p)), \text{pobs}) &= \\
&\ast (\text{ProcPar}(n, \text{gen}) (\text{process}))(t(\text{ParProc}(p), \text{pobs})) \\
\land (\square t)(\text{ProcParC}((\text{ActDef}(n, a)), \text{pobs}) &= \\
&\ast (\text{ProcAct}(n) (\text{ActDef}))(t(\text{ParAct}(a), \text{pobs}))
\end{align*}
\]

A similar approach has been adopted for the \text{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 \text{Circus} constructs and we have done so by defining a function \text{generalise} that transforms functions \(((X \times X) \rightarrow X)\) into functions of type \(((\text{NIL}_2 \text{VAL} \times X \times X) \rightarrow X)\) in the obvious way.

After this, we could define \( \Omega \) functions (\( \Omega_{\text{ProcBody}_2} \) and \( \Omega_{\text{ActBody}_2} \)) 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. \text{applyTacs}_{\text{ProcBody}_2} and \text{applyTacs}_{\text{ActBody}_2}, combines both lists of results using the distributed
In our example, there are four inputs, are sampled at fixed intervals. The inputs and outputs are represented by rounded boxes containing numbers. 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, $E$, $K_p$, $K_i$, and $K_d$, and one output, $Y$.

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.

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 $\Omega$ functions, which have to deal with different types of arguments. Besides, a null value, $\text{nil}$, is used as the first argument of this call.

Besides, a null value, $\text{nil}$, is used as the first argument of this call.
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.

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 shown 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, state0, 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
process Diff ≜ begin
  state Diff_St ≜ [pid_Diff_UnitDelay_St : U]
  pid_Diff_Sum ≜ Sum_PM
  pid_Diff_UnitDelay ≜ UnitDelay_g(X₀ ≜ 0 ∈ 0)

  pid_Diff
  In1? : U; Out1! : U
  Sum : pid_Diff_Sum
  UnitDelay : pid_Diff_UnitDelay
  Out1! = Sum.Out1!
  UnitDelay.In1? = Sum.In1?
  Sum.In1? = In1?
  Sum.In2? = UnitDelay.Out1!

  Init
  Diff_State'
  ∃ b : pid_Diff_UnitDelay • pid_Diff_UnitDelay_state' = b.initialstate

  Calculate_Diff
  ∆Diff_State
  In1?m, Out1! : U
  ∃ b : pid_Diff •
    b.In1? = In1?
    ∧ b.UnitDelay.state = pid_Diff_UnitDelay_state
    ∧ b.UnitDelay.state' = pid_Diff_UnitDelay_state'
    ∧ b.Out1! = Out1!

  Calculate_Diff_Out ≜ Calculate_Diff \ (pid_Diff_UnitDelay_state') \ ∃Diff_State
  Exec_Diff_out ≜ var In1 : U • E?x → In1 := x; var Out1 : U • Calc_Diff_out; Diff_out!Out1 → Skip
  Flows ≜ Exec_Diff_out
  Calculate_Diff_State ≜ Calculate_Diff \ Out1!
  Diff_StUpdt ≜ var In1 : U • E?x → In1 := x; Calc_Diff_St
  • Init; μ X • (Flows \ { } \ { E } \ { pid_Diff_UnitDelay_St } \ Diff_StUpdt); end_cycle → X
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 \( \text{Exec}_f \) 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 \( \text{Exec}_\text{Diff}_\text{out} \) there is one input variable \( \text{In1} \), and one output variable \( \text{Out1} \). The inputs are received in any order. The value \( x \) of the input is recorded in the corresponding variable \( \text{In1} \). 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 \( \text{Diff}_\text{out} \).

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 \( \text{end}_\text{cycle} \). 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 \( \text{Diff}_\text{out} \),
there is only one flow, so the parallelism in the action is reduced to a single action \texttt{Exec\_Diff\_out} that synchronises with \texttt{Diff\_StUpdt} on \texttt{E}.

The proof of refinement uses a four-phase strategy. In the first of them, \texttt{NB}, we refine the \texttt{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 \texttt{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 \texttt{NB} phase, we have the phase \texttt{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 \texttt{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, \texttt{Pr}, and \texttt{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 \texttt{NB} and \texttt{BJ}

In this section, we present the tactics \texttt{NB} and \texttt{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 \texttt{NB} and \texttt{BJ} of the refinement strategy by using refinement tactics.

6.1. Phase \texttt{NB}

In [2], we describe the \texttt{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 \texttt{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 \( t \).

\[
\text{Tactic } \text{TRY}(t) \equiv !(t | \text{skip})
\]

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

Tactic APPLYL(l, args) ≜ TRY(l ! hd args; APPLYL(l ! s))
end

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 that execute the flows as arguments. It applies to explicit process definitions, and transforms the process using Law copy-rule-action() .

Tactic applyCopyRule(fs)
\\\begin{align*}
\text{apply to process } P \equiv & \begin{PPars} \text{Main} \end{PPars} \\
\text{do } \\
& \text{law copy-rule-action("Flows");} \\
& \text{APPLYL(copy-rule-action, fs);} \\
& \text{TRY(law copy-rule-action(P+"_StUpdt"));}
\end{align*}
end

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).

Tactic NBStep1(fs)
\\\begin{align*}
\text{applyCopyRule(fs)}
\end{align*}
end

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 \begin{cases}
\text{var In1 : U • E?x → In1 := x;}
\text{var Out1 : U • Calc.Diff out; Diff.out!Out1 → Skip}
\end{cases}
; \text{end_cycle → 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 dIn to hold each of the input values, and take all of them in interleaving \( A_{In} \). 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).

Tactic syncInput() ≜
\\\begin{align*}
\text{applies to } (\text{var } d_{In} : U • A_{In}; A_{Out}) \parallel (ns1 \mid cs \mid ns2) \parallel (\text{var } d_{In} : U • A_{In}; A_{St}) \\
\text{do law var-exp-par-2();} \text{ var law par-seq-step-2();} \\
\text{generates } \text{var } d_{In} : U • A_{In}; (A_{Out} \parallel (ns1 \mid cs \mid ns2) \parallel A_{St}) \\
\text{proof obligations } usedC(A_{In}) \subseteq cs, \text{ wrtV}(A_{In}) \subseteq \{ d_{In} \}
\end{align*}
end

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.

\[
I; \mu X \bullet \left( ((\text{var } d \bullet A_{I_{n}}; A_{Out_n}) \parallel \ldots \parallel (\text{var } d \bullet A_{I_{n}}; A_{Out_n})) \parallel (\text{var } d \bullet A_{I_{n}}; A_{St}) \right); EC
\]

Our strategy is to remove the declarations \(d\) and interleaving \(A_{I_{n}}\) from the parallel composition of all the flows by recursively applying \text{syncInput}. Only then, we remove \(d\) and \(A_{I_{n}}\) from the outermost parallel composition. The auxiliary tactic \text{foldl} recursively applies a given tactic \(t\), from the innermost to the outermost parallel composition of an action \(A_{1} \parallel \ldots \parallel A_{n} \).

\[
\text{Tactic \ foldl}(t) \equiv \mu X \bullet \text{tactic TRY}((\text{skip}X); t)
\]

end

For example, the application of tactic \text{foldl} (\text{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 \(\parallel\) fails (lines 1 to 6), in which case, since we are in a TRY tactic, the tactic skips and returns \((\text{var } d \bullet A_{I_{n}}; A_{Out_n})\) (line 7). Then, the tactic applies \text{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).

\[
(var \ d \bullet A_{1}; A_{a_0}) \parallel ((var \ d \bullet A_{1}; A_{a_1}) \parallel (var \ d \bullet A_{1}; A_{a_2}))
\]

(1)

\[
= [\text{tactic TRY}((\text{skip}\parallel\text{tactic foldl}(\text{tactic syncInput()})); \ldots)]
\]

(2)

\[
(var \ d \bullet A_{1}; A_{a_1}) \parallel (var \ d \bullet A_{1}; A_{a_2})
\]

(3)

\[
= [\text{tactic TRY}((\text{skip}\parallel\text{tactic foldl}(\text{tactic syncInput()})); \ldots)]
\]

(4)

\[
(var \ d \bullet A_{1}; A_{a_1}) \parallel (var \ d \bullet A_{1}; A_{a_2})
\]

(5)

\[
= [\text{tactic TRY}((\text{skip}\parallel\text{tactic foldl}(\text{tactic syncInput()})); \ldots)]
\]

(6)

\[
(var \ d \bullet A_{1}; A_{a_2})
\]

(7)

\[
= [\text{tactic TRY}(\ldots; \text{tactic syncInput()})]
\]

(8)

\[
(var \ d \bullet A_{1}; (A_{a_1} \parallel A_{a_2}))
\]

(9)

\[
= [\text{tactic TRY}(\ldots; \text{tactic syncInput()})]
\]

(10)

\[
\text{var } d \bullet A_{1}; (A_{a_0} \parallel (A_{a_1} \parallel A_{a_2}))
\]

(11)

In the same way, we may use foldl in the n-ary case to join all the variables declarations \(d\) and interleaving \(A_{1}\) in the left-hand action of the outermost parallel composition. This is captured by the tactic that follows.

\[
\text{Tactic \ joinFlowsInput} \equiv \text{tactic foldl}(\text{tactic syncInput()})
\]

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.

\[
\text{Tactic \ NBStep2_f}() \equiv (\text{tactic joinFlowsInput}() \parallel \text{skip}); \text{tactic syncInput()}
\]

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 \(\text{Skip}\). 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.

Tactic NBStep2_1() \equiv \text{tactic joinFlowsInput(); } \boxed{\text{var skip; tactic createPar()} \bbox} 
end

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.

Tactic NBStep2() \equiv \text{tactic NBStep2_f(); tactic NBStep2_l()} 
end

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

\[
\begin{align*}
\text{Init; } \mu X \cdot \\
& \begin{cases}
\text{var } \text{In } 1 : \mathbb{U} \bullet \ E?x \rightarrow \text{In } 1 := x; \\
\{ \} | \{ E \} | \{ \text{pid}_{-}\text{Diff}_{-}\text{UnitDelay}_{-}\text{St} \} \\
\text{Calc}_{-}\text{Diff}_{-}\text{St}
\end{cases}
\end{align*}
\]

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.

\[
I; \mu X \cdot (\text{var } d_0 \bullet A_0; \ldots; (\text{var } d_n \bullet A_n)) || A_{St}); EC
\]

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, \texttt{expDisjVarPar}, which extracts both variable blocks from a parallel composition of two variable blocks, and joins them; we use foldl 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 \texttt{AIn}, and join the variable blocks.

The tactic \texttt{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.

Tactic expDisjVarPar() \equiv
\begin{align*}
\text{applies to } & (\text{var } d_0 \bullet A_0) || (n s_1 | cs | n s_2) || (\text{var } d_1 \bullet A_1) \\
\text{do law } & \text{var-exp-par(); var law par-comm(); law var-exp-par(); var law par-comm();} \\
& \text{law join-blocks()} \\
\text{generates } & \text{var } d_0; d_1 \bullet (A_0 || n s_1 | cs | n s_2) || A_1) \\
\text{proof obligations } & \{d_0, d_0\}' \cap FV(A_1) = \emptyset, \{d_1, d_1\}' \cap FV(A_0) = \emptyset
\end{align*}
end

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.

```
Tactic joinFlowsOutVarScope ≜ (tactic foldl_∥ (tactic expDisjVarPar()))∥skip
end
```

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 `AIn` 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 backwards) indicates that the law shall be applied from right to left.

```
Tactic expOutVarScope() ≜
applies to var d • AIn; ((var dO • A_O) □ ns1 | cs | ns2 □ ASt)
do [var] (skip∥(law var-exp-par(); law seq-right-unit()); law var-exp-seq();
   law skip∥law seq-right-unit^b())∥[];
   law join-blocks()
generates var d; dO • AIn; (A_O □ ns1 | cs | ns2 □ ASt)
proof obligations {dOut, d'_Out} ∩ FV(ASt) = ∅, {dOut, d'_Out} ∩ FV(AIn) = ∅
end
```

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.

```
Tactic NBStep3() ≜ [var]skip∥tactic joinFlowsOutVarScope(); tactic expOutVarScope()
end
```

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; μ X •
\[
\begin{aligned}
&\text{var } In1 : U;\ Out1 : U \bullet \\
&E?x \rightarrow In1 := x;\\
&(Calc_Diff_out; Diff_out!Out1 \rightarrow Skip)\\
&\{ \{ \} | \{ E \} | \{ \{ pid_Diff_UnitDelay_St \} \} }\\
&Calc_Diff_St\\
\end{aligned}
\]
; 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; μ X • (var d; dO • AIn; ((A_C0; A_O0) || (\ldots || (A_Cn; A_On)) || ASt))\{EC
```

As before, the strategy is to define a tactic that isolates the output communications in a parallel composition of two flows, use `foldl_∥` to isolate all the output communications in the left-hand action of the outermost composition.
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() \(\triangleq\)

  ```
  applies to \((A_{C_0}; A_{O_0}) \parallel [ns_1 | cs | ns_2] (A_{C_1}; A_{O_1})\)
  do law par-seq-step();
  (skip) [law par-comm(); law par-seq-step(); (skip) law par-comm()];
  law seq-assoc() generates \((A_{C_0}; A_{C_1}); (A_{O_0} [ns_1 | cs | ns_2] A_{O_1})\)
  proof obligations \(usedC(A_{C_0}) = \emptyset, usedC(A_{C_1}) = \emptyset,\)
  \(\text{wrt V}(A_{C_0}) \subseteq ns_1 \cap ns'_1, \text{wrt V}(A_{C_1}) \subseteq ns_2 \cap ns'_2\)
  \(usedV(A_{C_1}; A_{C_1}) \cap \text{wrt V}(A_{C_0}) = \emptyset, usedV(A_{C_0}) \cap \text{wrt V}(A_{C_1}) = \emptyset\)

end
```

The proof obligations are originated from the applications of Law par-seq-step. Using this tactic, we may isolate all the output communications \(A_{O_1}\) in the left-hand action of the outermost parallel composition. This is captured by the tactic joinFlowsCalc declared below.

**Tactic** joinFlowsCalc \(\triangleq\) (tactic fold\(\parallel\) (tactic isolateSeqActions()) \(\parallel\) skip)

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() \(\triangleq\)

  ```
  applies to \((A_{C}; A_{O}) \parallel [ns_1 | cs | ns_2] A_{St}\)
  do (skip) [law seq-right-unit()]; law par-seq-step();
  (skip) [law par-com(); law par-seq-step(); (skip) law par-com()];
  law seq-assoc() generates \((A_{C}; A_{St}); (A_{O} [ns_1 | cs | ns_2] Skip)\)
  proof obligations \(usedV(A_{St}) \cap \text{wrt V}(A_{Calc}) = \emptyset,\)
  \(usedV(A_{Comm}) \cap \text{wrt V}(A_{St}) = \emptyset,\)
  \(\text{wrt V}(A_{Calc}) \subseteq ns_1 \cup ns'_1, \text{wrt V}(A_{St}) \subseteq ns_2 \cup ns'_2\)

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 $\text{isolateIn}$ to the parallel composition. Finally, Law $\text{seq-assoc}$ isolates the parallel composition as the second part of a sequential composition.

Tactic $\text{NBStep4}() \equiv (\text{skip} \square (\text{tactic joinFlowsCalc}(); \text{tactic isolateIn}())); \text{law seq-assoc()}$

end

In our example, the application of the tactic $\text{joinFlowsCalc}$ immediately skips. The application of the tactic $\text{isolateIn}$ yields the following action.

Init; $\mu X \bullet$ \[
\begin{cases}
\text{var} \; \text{In}_1 : U; \; \text{var} \; \text{Out}_1 : U \bullet \\
((E? x \rightarrow \text{In}_1 := x); (\text{Calc}\_\text{Diff}_\text{out}; \text{Calc}\_\text{Diff}_\text{St})); \\
\text{Diff}\_\text{out}!\text{Out}_1 \rightarrow \text{Skip} \\
\langle \{ \} | \{E\} | \{ \text{pid}\_\text{Diff}_\text{UnitDelay}_\text{St} \} \rangle \\
\text{Skip} \\
\end{cases}; \text{end\_cycle} \rightarrow X
\]

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 $\text{interIntroAndSimpl}$ 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 $\text{interIntroAndSimpl}$ only to the second part.

Tactic $\text{NBSteps5\_6}() \equiv \text{skip} \square \text{tactic interIntroAndSimpl()}$

end

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

Init; $\mu X \bullet$ \[
\begin{cases}
\text{var} \; \text{In}_1 : U; \; \text{var} \; \text{Out}_1 : U \bullet \\
((E? x \rightarrow \text{In}_1 := x); (\text{Calc}\_\text{Diff}_\text{out}; \text{Calc}\_\text{Diff}_\text{St})); \\
(\text{Diff}_\text{out}!\text{Out}_1 \rightarrow \text{Skip})
\end{cases}; \text{end\_cycle} \rightarrow X
\]

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

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

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

\[\text{Tactic } \text{removeSeqComp}( ) \equiv \text{TRY}(\text{law seq-left-unit}^{b}) ; \text{TRY}(\text{law seq-right-unit}^{b})\]

end

The tactic \text{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 \text{NBStep2} (creating a parallel composition with \text{Skip} if needed), \text{NBStep3}, \text{NBStep4}, and \text{NBSteps5}–\text{6}. Finally, we apply the seventh step and remove any sequential composition with \text{Skip} in the variable block.

\[\text{Tactic } \text{NBMain}( ) \equiv \text{tactic insertSeqComp}; \begin{cases} \text{skip} \begin{cases} \text{tactic NBStep2}; \text{tactic NBStep3}; \begin{cases} \text{var tactic NBStep4}; \text{tactic NBSteps5–6} \end{cases} \end{cases} \end{cases} \text{skip} ; \text{tactic NBStep7}; \begin{cases} \text{var tactic removeSeqComp} \end{cases} \end{cases}\]

end

The tactic \text{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 \textit{Circus} process. First, it applies the tactic \text{NBStep1} using a list that contains the names of the actions of the process that execute its flows that is returned by the function \text{FNames}. Then, it applies the tactic \text{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 (\text{NBStep8}).

\[\text{Tactic } \text{NBProc}(pname) \equiv \text{program} \begin{cases} \langle pname, \text{tactic NBStep1(FNames(pname))}; \begin{cases} \text{beginend}(); \text{tactic NBMain}(); \text{tactic NBStep8}() \end{cases} \end{cases} \end{cases} \]

end

This tactic refines the corresponding \textit{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 \text{NBProc}. We achieve this by applying the following tactic to the \textit{Circus} program that contains their specifications.

\[\text{Tactic } \text{NB}(\text{ind\_blocks}) \equiv \text{APPLYT(NBProc, ind\_blocks)}\]

end

This tactic receives an argument that is a list of block names. In our example, we have that the list (\textit{Diff}, \textit{Sd}, \textit{Int}, \textit{Si}, \textit{Sp}, \textit{Sum}) can be used to apply the phase \text{NB} to the whole \textit{Circus} program.

Although not presented in this paper, \textit{Si}, \textit{Sd}, \textit{Sp}, and \textit{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, \textit{Si}, \textit{Sd}, and \textit{Sp}, take two input values and produce one output value; the last one of them \textit{Sum} takes three input values and produces one output value. Regardless of the difference in the internal structure of these processes, however, the tactic \text{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 \texttt{Calc\_Derivative} that implements the functionality of the blocks \texttt{Diff} and \texttt{Sd}. Similarly, we can also also find a procedure \texttt{Calc\_Integral} that implements the blocks \texttt{Si} and \texttt{Int}. Finally, the main program has procedures \texttt{Calc\_Proportion}, which implements the block \texttt{Sp}, and \texttt{Calc\_Output}, which implements the block \texttt{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 \texttt{Calc\_Proportion} and \texttt{Calc\_Output}, which implement the blocks \texttt{Sp} and \texttt{Sum}, are scheduled by the same program. Therefore, in this phase, we also join the processes \texttt{Sp} and \texttt{Sum} to produce a process \texttt{Sp\_Sum}.

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

6.2.1. Create a single process

The first step of the phase \texttt{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 \((\langle \text{Diff}, \text{Sd} \rangle, \langle \text{Int}, \text{Si} \rangle, \langle \text{Sp}, \text{Sum} \rangle)\) (henceforth called \texttt{pid\_blocks}) as argument. That means that, for instance, that processes \texttt{Diff} and \texttt{Sd} are to be joined. In fact, the argument \texttt{ind\_blocks} given to the tactic \texttt{NB} is the distributed concatenation of \texttt{pid\_blocks}.

The tactic below uses Law 5 (\texttt{join-proc-par}) to join the processes. This law is based directly on the definition of process parallelism [23]. It describes \(P_1 \parallel \langle \text{cs} \rangle P_2\) 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 \texttt{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 \texttt{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 \texttt{join\_all} to \(\langle \langle P_1, P_2, P_3 \rangle \rangle\), we get \(\langle \langle P_1, P_2 \rangle, \langle P_1, P_2, P_3 \rangle \rangle\) 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 \(\langle \langle \text{Diff}, \text{Sd} \rangle, \langle \text{Int}, \text{Si} \rangle, \langle \text{Sp}, \text{Sum} \rangle \rangle\), which is interactively used, as presented below, as argument for Law \texttt{join-proc-par}.

\begin{verbatim}
Tactic createSingleProcesses(blocks) \equiv APPLYL(join-proc-par, join\_all(blocks))
end
\end{verbatim}

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

\begin{verbatim}
Tactic BJSt1(blocks) \equiv createSingleProcesses(blocks)
end
\end{verbatim}

In our example, the application of this tactic, using the list \texttt{pid\_blocks} as argument, we get the \texttt{Circus} program sketched in Figure 6. The \texttt{Init} and \texttt{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 \texttt{Diff\_out} and \texttt{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 \equiv \textbf{begin state} \ Diff\_State \land Sd\_State \\
Diff\_State \equiv \textbf{[\{pid\_Diff\_UnitDelay\_state : \mathbb{U}; pid\_Diff\_In1 : \mathbb{U}; pid\_Diff\_Out1 : \mathbb{U}\]} \\
Sd\_State \equiv \textbf{[\{pid\_Sd\_In1, pid\_Sd\_In2 : \mathbb{U}; pid\_Sd\_Out1 : \mathbb{U}\]} \\

\begin{align*}
\text{end process Si}_\text{Int} \equiv \ldots \text{end} \\
\text{process Sp}_\text{Sum} \equiv \ldots \text{end} \\
\text{process PID} \equiv \ldots \text{end}
\end{align*}

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.

\begin{align*}
(I_0; R_0) \parallel (I_1; R_1) \parallel \ldots \parallel (I_n; R_n)
\end{align*}

Where $I_i$ are the state initialisation, if any, and $R_i$ are the recursive behaviours that synchronise on $\textit{end\_cycle}$ at the end of each cycle.

The tactic $\textit{isolateSeqActions}$ 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 $\textit{insertSeqComp}$ on both sides of the parallel composition before invoking $\textit{isolateSeqActions}$. Finally, we remove any sequential composition with $\textit{Skip}$ that might have been included earlier.

\begin{align*}
\text{Tactic isolateInitBin()} \equiv (\text{tactic insertSeqComp}() \parallel \text{tactic insertSeqComp}()) ; \\
\text{tactic isolateSeqActions}(); (\text{tactic removeSeqComp}() \parallel \text{skip}) \\
\text{end}
\end{align*}

For a given process name, the tactic $\text{BJSt2Proc}$ 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 \textit{isolateSeqActions}. For this reason, we use the tactic \texttt{foldr}, which is very similar to the previously presented \texttt{foldl}, but applies to left-associated parallel compositions ((\(A_1 \parallel A_2 \parallel \ldots \parallel A_n\)) (moving right)).

\textbf{Tactic BJSt2Proc}(pname) \triangleq
\begin{align*}
\text{program}\langle (\text{pname}, \text{\$beginend}\langle \text{} \rangle, \text{\$foldr}\langle \text{\$tactic isolateInitBin()}\rangle) \rangle
\end{align*}
\text{end}

Using this tactic, we may refine all processes that resulted from joining processes in parallel in the previous step (\textit{Diff\_Sd}, \textit{Int\_Si}, and \textit{Sp\_Sum}). This can be achieved by applying the following tactic to the \textit{Circus} program.

\textbf{Tactic BJSt2}(blocks) \triangleq \text{APPLYT}(\text{BJSt2Proc}, \text{join\_names}(blocks))
\text{end}

This tactic receives a sequence that contains sequences of blocks that are to be joined (for instance, \textit{pid\_blocks}) as arguments. It uses the function \textit{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 \textit{BJSt2Proc}. In our example, the tactic \textit{BJSt2Proc} is applied three times: one for each of the processes \textit{Diff\_Sd}, \textit{Int\_Si}, and \textit{Sp\_Sum}. For the first one, we have the resulting main action presented below.

\begin{verbatim}

 pid\_Diff\_Init;

 | µ X • (E?x → µ X • ((pid\_Diff\_In1 := x; Calculate\_Diff\_out); Calculate\_Diff\_State ; (Diff\_out!pid\_Diff\_Out1 → Skip)) ; endCycle → X) || {pid\_Diff\_UnitDelay\_state, pid\_Diff\_In1, pid\_Diff\_Out1} || \{Diff\_out, end\_cycle\} || {pid\_Sd\_In1, pid\_Sd\_In2, pid\_Sd\_Out1}\ |

 | \{Kd?x → pid\_Sd\_In1 := x ; endCycle → X\} || (pid\_Sd; Sd\_out!pid\_Sd\_Out1 → Skip))

\end{verbatim}

We are left with the initialisation of the state components related to the original \textit{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 \textit{end\_cycle}. For that, we use the fixed-point Law 2 (\textit{rec-sync}).
Law 2 (rec-sync)

\[
(\mu X \cdot A_1; c \rightarrow X) \parallel [ns_1 | c] \cup cs | ns_2] (\mu X \cdot A_2; c \rightarrow X)
\]

\[
= \mu X \cdot (A_1 [ns_1 | cs | ns_2] A_2); c \rightarrow X
\]

provided

• \(c \not\in cs \cup \text{usedC}(A_1) \cup \text{usedC}(A_2)\)
• \(\text{wrtV}(A_1) \cap \text{usedV}(A_2) = \emptyset\)
• \(\text{wrtV}(A_2) \cap \text{usedV}(A_1) = \emptyset\)

The first proviso ensures that in the parallelism of recursive actions, the channel \(c\) is only used at the end of the bodies \(A_1; c \rightarrow X\) and \(A_2; c \rightarrow X\) of each recursion. The set \(\text{usedC}(A)\) 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 \(\text{wrtV}(A)\) to refer to the set of variables whose values can potentially be changed by the action \(A\), and \(\text{usedV}(A)\) to the set of variables that are used by \(A\).

The tactic \(\text{isolateEC}\) extracts the synchronisation on \(\text{end\_cycle}\) 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 \(\text{NB}\) and \(\text{isolateInitBin}\), we include a sequential composition, if needed, using \(\text{insertSeqComp}\).

Tactic \(\text{isolateEC}() \equiv \text{tactic insertSeqComp}() \parallel \text{(skip} || \text{foldr} || \text{law rec-sync}())\)

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 \(\text{end\_cycle}\) from the parallel composition of all the parallel branches by recursively using Law 2. For this reason, we use the tactic \(\text{foldr}\). For a given process name, the tactic \(\text{BJSt3Proc}\) extracts the synchronisation on \(\text{end\_cycle}\). It works on the whole \(\text{Circus}\) programs but only changes, using \(\text{isolateEC}\), the main action of the specified process.

Tactic \(\text{BJSt3Proc}(pname) \equiv \text{program} \langle\langle pname, \text{beginend}(), \text{tactic isolateEC}()\rangle\rangle\)

end

The tactic that refines all processes that resulted from joining processes in parallel (\(\text{Diff\_Sd}, \text{Int\_Si}\), and \(\text{Sp\_Sum}\)) uses the tactic \(\text{BJSt3Proc}\) and the names of the blocks that have been joined.

Tactic \(\text{BJSt3}(blocks) \equiv \text{APPLYT}(\text{BJSt3Proc}, \text{join\_names}(blocks))\)

end

In our example, the tactic \(\text{BJSt3Proc}\) is applied to \(\text{Diff\_Sd}, \text{Int\_Si},\) and \(\text{Sp\_Sum}\). For the first one, this application yields the following recursive behaviour after the initialisation.

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

\[
\end\_cycle \rightarrow X
\]
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.

\[
\text{Tactic } \text{BJSt4A-all-config()} \equiv \\
\text{tactic BJSt4A-config1()} | \text{tactic BJSt4A-config2()} \\
| \text{tactic BJSt4A-config3A()} | \text{tactic BJSt4A-config3B()} | \text{tactic BJSt4A-config4()} \\
\text{end}
\]

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.

\[
\text{(TakeInAndCalc; OutInternalComm) } || \text{(InpInternalComm } \text{|| InpExternal}; \text{ CalcOutAndComm})
\]

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.

Tactic BJSt4A − config2() \[\triangleq\]

\[
\begin{align*}
(E?x & \rightarrow ((\text{pid}_\text{Diff}_\text{In}1 := x; \text{Calculate}_\text{Diff}_\text{out}); \text{Calculate}_\text{Diff}_\text{State})); \\
(D\text{iff}_\text{out}!\text{pid}_\text{Diff}_\text{Out}1 & \rightarrow \text{pid}_\text{Sd}_\text{In}2 := \text{pid}_\text{Diff}_\text{Out}1)
\end{align*}
\]

\[
||\{\text{pid}_\text{Diff}_\text{UnitDelay}_\text{state}, \text{pid}_\text{Diff}_\text{In}1, \text{pid}_\text{Diff}_\text{Out}1, \text{pid}_\text{Sd}_\text{In}2\} | \{\text{pid}_\text{Sd}_\text{In}1\}||
\]

\[
(Kd?x & \rightarrow \text{pid}_\text{Sd}_\text{In}1 := x)
\]

\[
(\text{pid}_\text{Sd}; \text{Sd}_\text{out}!\text{pid}_\text{Sd}_\text{Out}1 \rightarrow \text{Skip})
\]

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

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.

Tactic BJSt4AProc(pname) \[\triangleq\]

\[
\text{program } ((\text{pname}, \begin{align*}
\begin{vmatrix}
\end{align*}
\end{vmatrix}
\text{skip} | \text{tactic BJSt4A_all_config() | skip}))
\]

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.

Tactic BJSt4A(blocks) \[\triangleq\] APPLYT(BJSt4AProc, join_names(blocks))

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 (\((P_1 \alpha_1) \parallel \ldots \parallel (P_n \alpha_n)\) \& 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 mapr, 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 \texttt{mapr} applies a tactic to all elements in a nested right-associated parallel composition. It is defined as follows.

\begin{verbatim}
Tactic \texttt{mapr}(t) \triangleq \mu X \cdot (\parallel X) \mid t
\end{verbatim}

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 \texttt{BjSt4\_HidPrep} applies the Law 4 (\texttt{join\_proc\_hid}) to the overall \textit{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 \textit{Circus} specification. The new process \( P_1 \) is very similar to \( P \), but hide these events in its main action. In the overall \textit{Circus} specification, we may now reference \( (P_1 (\alpha P) \setminus cs) \); the new interface removes \( cs \) from the original one.

\begin{verbatim}
Tactic \texttt{BjSt4\_HidPrep}(main) \triangleq
    program \{ (main, \parallel (law par-hid-dist()) \parallel (law hid-contract())) \};
    APPLYL(join-proc-hid, join_names(blocks))
\end{verbatim}

In our example, we have that the \( PID \) is the main action and its definition, before the application of the tactic \texttt{BjSt4\_HidPrep} is as follows.

\begin{verbatim}
PID \triangleq \\
\parallel \parallel \parallel \\
Diff_{Sd} E, Kd, Diff_{out}, Sd_{out}, end_cycle \\
\mid Int_{Si} E, Ki, Si_{out}, Int_{out}, end_cycle \\
\mid Sp_{Sum} E, Kp, Sd_{out}, Int_{out}, Sp_{out}, Y, end_cycle \\
\setminus \parallel \parallel \parallel \\
\setminus Diff_{out}, Int_{out}, Sd_{out}, Sp_{out}
\end{verbatim}

The application of the tactic \texttt{BjSt4\_HidPrep} distributes the hiding over the parallel composition and changes the overall \textit{Circus} program including the definition of \( PID \). A sketch of the resulting \textit{Circus} program is presented below.

\begin{verbatim}
... Diff_{Sd1} \triangleq \ldots \ldots \parallel \parallel Diff_{out} \parallel \parallel \parallel \parallel Diff_{Sd}
\end{verbatim}

\begin{verbatim}
PID \triangleq \\
\parallel \parallel \parallel \\
Diff_{Sd1} E, Kd, Sd_{out}, end_cycle \\
\mid Int_{Si1} E, Ki, Int_{out}, end_cycle \\
\mid Sp_{Sum1} E, Kp, Sd_{out}, Int_{out}, Y, end_cycle \\
\setminus \parallel \parallel \parallel \\
\setminus Int_{out}, Sd_{out}
\end{verbatim}

The new definition of \( Diff_{Sd} \), \( Diff_{Sd1} \), 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 (\texttt{hid\_step}) to remove the communication. This distribution is achieved by the tactic \texttt{HidDistStep}, which exhausts the application of the hiding distribution laws over the \textit{Circus} operators and, when it is no longer possible to distribute the hiding, it applies the step Law 13. Finally, it uses
Law 8 \((\text{hid-idem})\) to remove the hiding.

\[
\text{Tactic HidDistStep}(\) \equiv \mu \, X \bullet \text{TRY}
\left(
\begin{array}{l}
(\text{law hid-seq-dist}() ; (X \boxplus X)) \\
| (\text{law hid-par-dist}() ; (X \boxplus X)) \\
| (\text{law hid-inter-dist}() ; (X \boxplus X)) \\
| (\text{law hid-rec-dist}() ; (\mu X)) \\
| (\text{law hid-step}() ; X)
\end{array}
\right)
\]

end

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

\[
\text{Tactic BJSt4BProc}(\text{pname}) \equiv \text{program} \langle (\text{pname}, \mu X \bullet \text{TRY}) (\begin{array}{l}
(\text{law hid-seq-dist}() ; (X \boxplus X)) \\
| (\text{law hid-par-dist}() ; (X \boxplus X)) \\
| (\text{law hid-inter-dist}() ; (X \boxplus X)) \\
| (\text{law hid-rec-dist}() ; (\mu X)) \\
| (\text{law hid-step}() ; X)
\end{array})angle
\]

end

The tactic that refines all process that resulted from joining two process in parallel (\(\text{Diff}_Sd, \text{Int}_Si, \text{and} \text{Sp}_Sum\)) uses the tactic \(\text{BJSt4Proc}\) and the names of the blocks that have been joined.

\[
\text{Tactic BjSt4B}(\text{blocks}) \equiv \text{APPLYT}(\text{BJSt4BProc}, \text{join_names}(\text{blocks}))
\]

end

The function \(\text{join_names}\) is very similar to \(\text{join}\), but sufixes all names with \(1\).

In our example, the application of the tactic \(\text{BJSt4B}\) removes the communication through \(\text{Diff}_\text{out}\) in the main action of process \(\text{Diff}_Sd\) yielding the main action presented below.

\[
\begin{array}{l}
\text{pid}_Sd\text{In}1 := \text{pid}_Sd\text{Out}1 \\
\text{pid}_\text{Diff}_\text{Init} := \text{pid}_\text{Diff}_\text{In}1 := x; \text{Calculate}_\text{Diff}_\text{Out}; \text{Calculate}_\text{Diff}_\text{State})
\end{array}
\]

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 \(\text{BJSt4CProc}\)
that applies this step to the main action of a specific process, given its name. Finally, we define a tactic \texttt{BjSt4C} that applies this tactic to each process that resulted from joining two process in parallel. In \texttt{Diff} \_\texttt{Sd}, there is only one internal communication; the application of \texttt{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.

\[
I; \mu X \bullet (((((i_1\?x \rightarrow ((v_1 := x; A_1); A_2)); A_3) \| ns_1 \| ns_2) (i_2\?x \rightarrow v_2 := x)); A_4)\end_cycle \rightarrow X
\]

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

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

\[
(i_1\?x \rightarrow ((v_1 := x; A_1); A_2)); A_3
\]

First, it applies the associativity law to the action prefixed by the input on \(i_1\) to isolate the assignment.

\[
((i_1\?x \rightarrow (v_1 := x; (A_1; A_2))); A_3
\]

Then, it applies the associativity law for prefixing, Law 28 (prefix-seq-assoc), to isolate the prefixed action \((i_1\?x \rightarrow st_1 := x). \)

\[
((i_1\?x \rightarrow v_1 := x); (A_1; A_2)); A_3
\]

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.

\[
(i_1\?x \rightarrow v_1 := x); ((A_1; A_2); A_3)
\]

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.

\[
(((i_1\?x \rightarrow v_1 := x) \| ns_1 \| ns_2) (i_2\?x \rightarrow v_2 := x)); ((A_1; A_2); A_3)); A_4
\]

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 \texttt{pid\_Diff\_UnitDelay\_state}, \texttt{pid\_Diff\_Out1}, and \texttt{pid\_Sd\_In2} 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** \texttt{BjSt4D-config2} defines

applying to \(((i_1\?x \rightarrow ((st_1 := x; A_1); A_2)); A_3) \| ns_1 \| ns_2) (((st_2 := x)); A_4)\)

do

\[
\left(\begin{array}{c}
\left(\begin{array}{c}
\text{law seq-assoc}^b(); \text{law prefix-seq-assoc}() & \text{\text{\$skip}}
\end{array}\right) & \text{\text{\$skip}}
\end{array}\right) ;
\]

\[
\begin{array}{c}
\text{law inter-seq-extract-snd}()
\end{array}
\]

\[
\left(\begin{array}{c}
\text{law inter-unused-name}(); \text{law inter-comm}()
\end{array}\right) & \text{\text{\$skip}}
\end{array}\right)
\]

\[
\text{law seq-assoc}()
\]

end

For a given process name, the tactic \texttt{BJSt4DProc} introduces the interleaving of inputs in the main action. It works on the whole \texttt{Circus} programs but only changes, using \texttt{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.

Tactic BJSt4DProc(pname) ≜

\[
\text{program} \left( \text{program} \langle \text{pname}, \hat{=} = \langle \text{beginend} \langle \langle \text{skip} \hat{=} \langle \text{TRY} (\text{tactic BJSt4D-config2()} \hat{=} \text{skip}) \rangle \rangle \rangle \rangle \right)
\]

end

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.

\[
\text{pid}_\text{Diff}_\text{Init}; \mu X \bullet \left( \left( (E?x \rightarrow \text{pid}_\text{Diff}_\text{In}1 := x) \right) \left( (Kd?x \rightarrow \text{pid}_\text{Sd}_\text{In}1 := x) \right) \left( (\text{Calculate}_\text{Diff}_\text{out}; \text{Calculate}_\text{Diff}_\text{State}) \right) \left( (\text{pid}_\text{Sd}_\text{In}2 := \text{pid}_\text{Diff}_\text{Out}1) \right) \left( (\text{pid}_\text{Sd}; \text{Sd}_\text{out}!\text{pid}_\text{Sd}_\text{Out}1 \rightarrow \text{Skip}) \right) \right) \right) ; \text{endCycle} \rightarrow X
\]

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.

Tactic BJSt4D(blocks) ≜ APPLYT(BJSt4DProc, join_names1(blocks))

end

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.

Tactic BJSt4(main, blocks) ≜

\[
\text{tactic BJSt4A(blocks)} ; \text{tactic BJSt4_HidPrep(main)} ; \text{tactic BJSt4B(blocks)} ; \text{tactic BJSt4C(blocks)} ; \text{tactic BJSt4D(blocks)}
\]

end

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.

Tactic BJ(main, blocks) ≜

\[
\text{tactic BJSt1(blocks)} ; \text{tactic BJSt2(blocks)} ; \text{tactic BJSt3(blocks)} ; \text{tactic BJSt4(main, blocks)}
\]

end

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 ArcAngelC, 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 ArcAngelC 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 RCells containing the possible output programs with their corresponding set of proof obligations. In ArcAngelC, 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 ArcAngelC. 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 ArcAngelC; 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. ArcAngelC 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

\[
PF ::= \text{partial} | \text{finite} \\
\text{pseq}_X ::= PF \times \text{seq } X
\]

We define an order ≤ on these pairs such that for a, b : pseq X, if a is finite, then a ≤ 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 \(b\).

\[
\begin{array}{l}
\forall gs, hs : \text{seq } X \bullet \\
\quad (\text{finite}, gs) \leq (\text{finite}, hs) \Leftrightarrow gs = hs \\
\quad (\text{finite}, gs) \leq (\text{partial}, hs) \Leftrightarrow \text{false} \\
\quad (\text{partial}, gs) \leq (\text{finite}, hs) \Leftrightarrow gs \text{ prefix } hs \\
\quad (\text{partial}, gs) \leq (\text{partial}, hs) \Leftrightarrow gs \text{ prefix } hs
\end{array}
\]

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

\[
\begin{array}{l}
\forall c : \text{chain } X \bullet \\
\quad c \in \text{chain} \Leftrightarrow (\forall x, y : c \bullet x \leq y \lor y \leq x)
\end{array}
\]

The set \(\text{pchain}\) contains all downward closed chains.

\[
\begin{array}{l}
\forall c : \text{pchain } X \bullet \\
\quad c \in \text{pchain} \Leftrightarrow (\forall x : c ; y : \text{pfseq } X | y \leq x \bullet y \in c)
\end{array}
\]

The set \(\text{pfiseq}\) contains partial, finite, and infinite list of elements of \(X\), which are prefixed-closed chains of elements in \(\text{pfseq } X\).

\(\text{pfiseq } X = \text{pchain } X\)

The idea is that \(\perp = \{(\text{partial}, (\langle \rangle))\}\), the empty list \(\langle \rangle \infty = \{(\text{finite}, (\langle \rangle))\}\), the finite list \(\langle e_1, e_2, \ldots, e_n \rangle\) is represented by the set containing \(\{(\text{finite}, \langle e_1, e_2, \ldots, e_n \rangle)\}\) 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.

\[
\begin{array}{l}
\forall xs : \text{seq } X ; f : X \rightarrow Y ; \text{pfmap } (\text{pf}, xs) = (\text{pf}, (f \circ xs))
\end{array}
\]

The function \(\text{pfmap}\) 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.

\[
\begin{array}{l}
\forall s : \text{pfiseq } \text{pfiseq } X \bullet \text{pfiseq } X = \bigcup_{\infty} \{ c : s \bullet \infty \}
\end{array}
\]

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

\[
\begin{align*}
\mathcal{X} : \text{pfiseq[pfiseq}[X]] & \to \text{pfiseq}[X] \\
\mathcal{X}^{\infty} / (\text{finite}, \langle \rangle) &= \{ \} \\
\mathcal{X}^{\infty} / (\text{partial}, \langle \rangle) &= \{ (\text{partial}, \langle \rangle) \} \\
\forall g : \text{pfiseq}[X] \cdot \mathcal{X}^{\infty} / (\text{finite}, (g)) &= g \land \mathcal{X}^{\infty} / (\text{partial}, (g)) = g \land \{ (\text{partial}, \langle \rangle) \} \\
\forall gs, hs : \text{seq}(\text{pfiseq}[X]) \cdot \mathcal{X}^{\infty} / (\text{finite}, gs \cdot hs) &= (\mathcal{X}^{\infty} / (\text{finite}, gs)) \cdot (\mathcal{X}^{\infty} / (\text{finite}, hs)) \\
\land \mathcal{X}^{\infty} / (\text{partial}, gs \cdot hs) &= (\mathcal{X}^{\infty} / (\text{finite}, gs)) \cdot (\mathcal{X}^{\infty} / (\text{partial}, hs))
\end{align*}
\]

The function $\mathcal{X}^{\infty}$ is the concatenation function for possibly infinite lists. Its definition is

\[
\begin{align*}
\mathcal{X}^{\infty} : \text{pfiseq}[X] \times \text{pfiseq}[X] & \to \text{pfiseq}[X] \\
\forall a, b : \text{pfiseq}[X] \cdot a \mathcal{X}^{\infty} b &= \{ x : a \land y : b \land x \land y \}
\end{align*}
\]

where the function $\land$ is the concatenation function for $\text{pfiseq}[X]$ defined as

\[
\begin{align*}
\land : \text{pfiseq}[X] \times \text{pfiseq}[X] & \to \text{pfiseq}[X] \\
\forall gs, hs : \text{seq}[X] \cdot s : \text{pfiseq}[X] \cdot (\text{finite}, gs) \land (\text{finite}, hs) &= (\text{finite}, gs \land hs) \\
\land (\text{partial}, gs) \land (\text{partial}, hs) &= (\text{partial}, gs \land hs) \\
\land s &= (\text{partial}, gs)
\end{align*}
\]

(iii) The function $\text{head}_{\infty}$ returns a list containing the first element of a possibly infinite list.

\[
\begin{align*}
\text{head}_{\infty} : \text{pfiseq}[X] & \to \text{pfiseq}[X] \\
\forall xs : \text{pfiseq}[X] \cdot \text{head}_{\infty}(xs) &= \text{take}_{\infty} 1 xs 
\end{align*}
\]

It uses the function $\text{take}_{\infty}$ that returns a list containing the first $n$ elements of a possibly infinite list.

\[
\begin{align*}
\text{take}_{\infty} : \mathbb{N} & \to \text{pfiseq}[X] \\
\forall n : \mathbb{N} \cdot xs : \text{pfiseq}[X] \cdot n = 0 & \Rightarrow \text{take}_{\infty} n xs = [\infty]_\infty \\
\land n \neq 0 & \Rightarrow \text{take}_{\infty} n \perp [X] = \perp [X] \\
\land \text{take}_{\infty} n [\infty]_\infty = [\infty]_\infty \\
\land \text{take}_{\infty} n xs = (\text{head}_{\infty} xs) : \infty (\text{take}_{\infty} (n - 1) (\text{tail}_{\infty} xs))
\end{align*}
\]

For a possibly infinite list $xs$, the function $\text{head}_{\infty}$ returns the head of $xs$.

\[
\begin{align*}
\text{head}_{\infty} : \text{pfiseq}[X] & \to X \\
\forall s : \text{pfiseq}[X] \cdot \text{head}_{\infty}(s) &= (\mu x : X \mid (\text{partial}, \langle x \rangle) \in s)
\end{align*}
\]

On the other hand, the function $\text{tail}_{\infty}$ returns the tail of $xs$. 

37
\[
\text{tail}_\infty : \text{pfiseq}[X] \rightarrow \text{pfiseq}[X]
\]

\[
\forall s : \text{pfiseq}[X] \bullet \text{tail}_\infty (s) = \{ x : s \mid x.2 \neq \langle \rangle \} \bullet (x.1, \text{tail}(x.2))
\]

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 \( \text{tail}_\infty \) was not present; nevertheless, by definition (\text{partial}, \langle \rangle) is a member of every \( \text{pfiseq}[X] \). Without the conditional, there would be an undefined value involved since \( \text{tail} \) is not defined for empty sequences.

(iv) The function \( \circ \) applies a possibly infinite list of functions to a single argument.

\[
\text{Π}_\infty : (\text{pfiseq}[X] \rightarrow [X \rightarrow Y] \times X) \rightarrow \text{pfiseq}[Y]
\]

\[
\forall fs : \text{pfiseq}[X \rightarrow Y]; f : X \rightarrow Y; x : X \bullet
\]

\[
\begin{align*}
\downarrow [X \rightarrow Y] \circ x &= [\infty]_\infty \\
\land [\infty]_\infty \circ x &= [\infty]_\infty \\
\land (f \circ \infty \circ fs) \circ x &= \ell (f \circ x) \circ x
\end{align*}
\]

(v) The function \( \Pi_\infty \) 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 \( L(n) \) to denote the fact that the Local action definitions may include references to the action \( n \); the same holds for the \text{Main Action \( MA(n) \). Later references to \( L(A) \) and \( MA(A) \) are the result of substituting the body \( A \) of \( n \) for some or all occurrences of \( n \) in \( L \) and \( MA \).

Law 3 (assign-intro) \( w : [\text{pre}, \text{post}] \sqsubseteq_A x := e \) provided \( \text{pre} \Rightarrow \text{post}[e/x] \)

Law 4 (join-proc-hid)

\[
CPars_1 ((P_1 \alpha \_P) \setminus cs)
\]

\[
P_1 \equiv \begin{array}{l}
\text{begin PPars}_1 \text{ state } P\_State \equiv \ SExp : PPars_2 \bullet PAct \end{array}
\]

\[
CPars_2 ((P_1 \alpha \_P) \setminus cs)
\]

\[
= CPars_1 ((P_2 (\alpha \_P \setminus cs)))
\]

\[
P_2 \equiv \begin{array}{l}
\text{begin PPars}_1 \text{ state } P\_State \equiv \ SExp : PPars_2 \bullet PAct \setminus cs \end{array}
\]

\[
CPars_2 ((P_2 (\alpha \_P \setminus cs)))
\]

provided

\bullet \quad P_1 \notin \alpha (CPars_1 (P_2)) \cup \alpha (CPars_2 (P_2))
\bullet \quad P_2 \notin \alpha (CPars_1 (P_1)) \cup \alpha (CPars_2 (P_1))

38
Law 5 (join-proc-par)

\[ CPars_1 (P \parallel Q) \]
\[ P \triangleq \text{begin } PPars_1 \text{ state } P_{\text{State}} \triangleq SExp_P \text{ PPars}_2 \bullet PAct \text{ end } \]
\[ CPars_2 (P \parallel Q) \]
\[ Q \triangleq \text{begin } QPars_1 \text{ state } Q_{\text{State}} \triangleq SExp_Q \text{ QPars}_2 \bullet QAct \text{ end } \]
\[ CPars_3 (P \parallel Q) = \]
\[ P \triangleq \text{begin state } P_{\text{State}} \triangleq SExp_P \text{ PPars}_1 \text{ PPars}_2 \bullet PAct \text{ end } \]
\[ Q \triangleq \text{begin state } Q_{\text{State}} \triangleq SExp_Q \text{ QPars}_1 \text{ QPars}_2 \bullet QAct \text{ end } \]

provided
- \( P \notin \alpha(Q) \cup \bigcup_i \alpha(CPars_i (P \parallel Q)) \)
- \( Q \notin \alpha(P) \cup \bigcup_i \alpha(CPars_i (P \parallel Q)) \)
- \( \alpha(P_{\text{State}}) \cap \alpha(Q_{\text{State}}) = \emptyset \)
- \( \alpha(PPars_1) \cup \alpha(PPars_2) \cap \alpha(QPars_1) \cup \alpha(QPars_2) = \emptyset \)

Law 6 (copy-rule-action)

\[ \text{begin } (\text{state } S) (n \triangleq A) L(n) \bullet MA(n) \text{ end } \]
\[ = \text{begin } (\text{state } S) (n \triangleq A) L(A) \bullet MA(A) \text{ end } \]

Law 7 (hid-contract)

\[ (P \alpha_P) \setminus cs = (P \alpha_P) \setminus cs' \text{ where } cs' = cs \cap \alpha_P \]

Law 8 (hid-idem)

\[ A \setminus cs = A \]

provided \( usedC(A) \cap cs = \emptyset \)

Law 9 (hid-inter-dist)

\[ (A_1 [\parallel n_1 | n_2] | A_2) \setminus cs_2 = (A_1 \setminus cs_2) [\parallel n_1 | n_2] (A_2 \setminus cs_2) \]

Law 10 (hid-par-dist)

\[ (A_1 [\parallel n_1 | n_2] | A_2) \setminus cs_2 = (A_1 \setminus cs_2) [\parallel n_1 | n_2] (A_2 \setminus cs_2) \]

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)

\[ (c \rightarrow A) \setminus cs = A \setminus cs \]

provided \( c \in cs \)

Law 14 (inter-comm)

\[ A_1 [\parallel n_1 | n_2] A_2 = A_2 [\parallel n_2 | n_1] A_1 \]
Law 15 (inter-index)
\[ \| x : \{v_1, \ldots, v_n\} \bullet \| ns(x) \| A(x) \]
= \[ A(v_1) \| ns(v_1) \| \bigcup \{ x : \{v_2, \ldots, v_n\} \bullet \| ns(x) \| \} \ldots \| A(v_{n-1}) \| \| ns(v_{n-1}) \| \| ns(v_n)\| A(v_n) \| \]

Law 16 (inter-unit) \[ A \| ns_1 \| ns_2 \| \text{Skip} = A \]

Law 17 (inter-seq-assig)
\[ v_1 := e_1 \| ns_1 \| ns_2 \| v_2 := e_2 = v_1 := e_1 ; v_2 := e_2 \]
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)
\[ (A_1 ; A_2) \| ns_1 \| ns_2 \| A_3 = (A_1 \| ns_1 \| ns_2 \| A_3) ; A_2 \]
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) \[ A_1 \| \{x \cup ns_2 \| ns_3\| A_2 = A_1 \| ns_2 \| ns_3\| A_2 \]
provided \[ x \notin \text{wrtV}(A_1) \]

Law 20 (join-blocks) \[ \text{var } x : T_1 \bullet \text{var } y : T_2 \bullet A = \text{var } x : T_1 ; y : T_2 \bullet A \]

Law 21 (par-comm) \[ A_1 \| ns_1 \| cs \| ns_2 \| A_2 = A_2 \| ns_2 \| cs \| ns_1 \| A_1 \]

Law 22 (par-hid-dist)
\[ ((P_1 \alpha_1) \| (P_2 \alpha_2) \| \ldots \| (P_n \alpha_n)) \| cs \]
= \[ ((P_1 \alpha_1) \| cs_1) \| ((P_2 \alpha_2) \| cs_2) \| \ldots \| ((P_n \alpha_n) \| cs_n)) \| cs' \]
where
- \[ \forall i : 1 \ldots n \bullet cs_i = cs \setminus (\alpha_i \cap (\bigcup_{j \neq i} \alpha_j)) \]
- \[ cs' = cs \cup \bigcup_{i=1 \ldots n} cs_i \]

Law 23 (par-inter-2) \[ A_1 \| ns_2 \| ns_2 \| A_2 = A_1 \| ns_2 \| \emptyset \| ns_2 \| A_2 \]

Law 24 (par-out-imp-inter-exchange)
\[ (A_1 ; c_1 ! v \rightarrow \text{Skip}) \| ns_1 \| \emptyset \| \emptyset \| ns_2 \] \[ ((c_1 ? x \rightarrow A_2(x) \| ns_3 \| ns_4) \| A_3) ; A_4 \]
= \[ ((A_1 ; c_1 ! v \rightarrow A_2(v)) \| ns_1 \cup ns_3 \| ns_4) \| A_3) ; A_4 \]
provided
- \[ c_1 \notin \text{initials}(A_3) \]
- \[ c_1 \notin \bigcup_{i \in 1 \ldots 4} usedC(A_i) \]
- \[ ns_3 \cup ns_4 \subseteq ns_2 \]
- \[ wrtV(A_1) \subseteq ns_1 \]
- \[ wrtV(A_2) \subseteq ns_3 \]
- \[ wrtV(A_4) \subseteq ns_2 \]
Law 25 (par-out-inp-inter-exchange-n)

\[(B_1; (\| i : 1 \ldots n \bullet [ns_1 |\{c_i:v_i \rightarrow \text{Skip}\}] ns_2))\]
\[\| [ns_1 |\{c_1, \ldots , c_n \}] ns_2\]
\[(((\| i : 1 \ldots n \bullet [ns_1 | c_i:x \rightarrow A_i(x)] ns_3 | ns_4]) B_2); B_3)\]
\[=\]
\[(B_1; (((| i : 1 \ldots n \bullet [ns_1 | c_i:v_i \rightarrow A_i(v_i)]) [ns_1 \cup ns_3 | ns_4]) B_2)); B_3\]

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

Law 26 (par-seq-step)

\[(A_1; A_2) [ns_1 | cs | ns_2] A_3 = A_1; (A_2 [ns_1 | cs | ns_2] A_3)\]

provided
\cdot \text{usedC}(A_1) = \emptyset, \text{usedV}(A_3) \cap \text{wrtV}(A_1) = \emptyset
\cdot \text{wrtV}(A_1) \subseteq ns_1 \cup ns_2

Law 27 (par-seq-step-2)

\[\text{var } d \bullet (A_1; A_2) [ns_1 | cs | ns_2] (A_1; A_3) = \text{var } d \bullet A_1; (A_2 [ns_1 | cs | ns_2] A_3)\]

provided
\cdot \text{usedC}(A_1) \subseteq cs
\cdot \text{wrtV}(A_1) \subseteq \alpha(d)

Law 28 (prefix-seq-assoc)

\[c \rightarrow (A_1; A_2) = (c \rightarrow A_1); A_2\]

provided \(\text{FV}(A_2) \cap \alpha(c) = \emptyset\)

The reference to \(L(\_\_\_\_)\) 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)

\[\text{begin } (\text{state } S) [L(x : T) \bullet (\text{var } x : T \bullet \text{MA}) \text{ end} = \text{begin } (\text{state } S \land \lfloor x : T \rfloor) L(\_\_\_\_) \bullet \text{MA end}\]

Law 30 (prom-var-state-2)

\[\text{begin } L(x : T) \bullet (\text{var } x : T \bullet \text{MA}) \text{ end = begin } (\text{state } \lfloor x : T \rfloor) L(\_\_\_\_) \bullet \text{MA end}\]

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

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

Law 33 (seq-right-unit) \(A = A; \text{Skip}\)
Law 34 (var-exp-par)

\[(\text{var } d : T \bullet A_1) \parallel [n s_1 | cs | n s_2] A_2 = (\text{var } d : T \bullet A_1 \parallel n s_1 | cs | n s_2) A_2\]

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

Law 35 (var-exp-par-2)

\[(\text{var } d \bullet A_1) \parallel [n s_1 | cs | n s_2 | A_2] = (\text{var } d \bullet A_1 \parallel n s_1 | cs | n s_2) A_2\]

Law 36 (var-exp-rec)

\[\mu X \bullet (\text{var } x : T \bullet F(X)) = \text{var } x : T \bullet (\mu X \bullet F(X))\]

provided \(x\) is initialised before use in \(F\)

Law 37 (var-exp-seq)

\[A_1; (\text{var } x : T \bullet A_2); A_3 = (\text{var } x : T \bullet A_1; A_2; A_3)\]

provided \(\{x, x'\} \cap (FV(A_1) \cup FV(A_3)) = \emptyset\)

References


