Science of Computer Programming 76 (2011) 792–833

Contents lists available at ScienceDirect

Science of Computer Programming

journal homepage: www.elsevier.com/locate/scico

A tactic language for refinement of state-rich concurrent specifications

Marcel Oliveira [a,*], Frank Zeyda [b], Ana Cavalcanti [b]

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

[b] Department of Computer Science, University of York, York, YO10 5GH, UK

ARTICLE INFO

Article history:
Available online 21 December 2010

Keywords:
Concurrency
Refinement calculus
Tactics
Control law diagrams

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, the practical application of this technique can be hard. Useful strategies have been identified, described, and used, and by documenting them as tactics, they can be expressed and repeatedly applied as single transformation rules. Here, we present ArcAngel*C*, a language for defining such tactics; we present the language, its semantics, and its application in the formalisation of an existing strategy for verification of Ada implementations of control systems specified by Simulink diagrams. We also discuss its mechanisation in a theorem prover, ProofPower-Z.

1. Introduction

Circus [8] is a formalism that combines Z [49] and CSP [16] to cover both data and behavioural aspects of a system development or verification. It distinguishes itself from other such combinations like CSP-Z [11], TCOZ [20], and CSP-B [44], in that it has a refinement theory and calculus for code development and verification [32]. Using *Circus*, we can develop state-rich reactive systems in a calculational style [26].

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

We present ArcAngel*C*, a refinement-tactic language for *Circus* whose constructs are similar to those in ArcAngel [35], a refinement-tactic language for sequential programs. Both languages are based on a general tactic language, Angel [23], which is not tailored to any particular proof tool and assumes only that rules transform proof goals. Angel supports the use of angelic choice to define tactics that backtrack to search for successful proofs. The angelic choice of Angel (and of ArcAngel and ArcAngel*C*) distinguishes them from LCF's tactic languages [40,12] and from other theorem provers' tactic languages. This difference has important consequences in both semantical and practical terms.

ArcAngel has a formal semantics and an extensive set of laws that provide a complete tool to reason about tactics of refinement. The semantics of ArcAngel and its set of laws can be found in [31], along with the formalisation of useful refinement strategies. Tool support is provided by Gabriel [38].

Like ArcAngel, as a refinement-tactic language, ArcAngel*C* must take into account the fact that the application of laws yields not only a program, but proof obligations as well. So, the result of applying a tactic is a program and the cumulative

* Corresponding author. Tel.: +55 0 84 3215 3814; fax: +55 084 3215 3813.
E-mail address: marcel@dimap.ufrn.br (M. Oliveira).
URL: http://www.dimap.ufrn.br/~marcel (M. Oliveira).

0167-6423/$ – see front matter © 2010 Elsevier B.V. All rights reserved.
doi:10.1016/j.scico.2010.11.012

set of proof obligations generated by all law applications . The constructs of ArcAngel*C* are similar to those of ArcAngel. The major differences, both in syntax and semantics, arise from the need to deal with the application of all categories of *Circus* refinement laws: we can apply tactics to *Circus* programs or to its components, namely, processes and actions.

Handling a variety of syntactic categories to which laws can be applied is the main challenge in extending ArcAngel to ArcAngel*C*. This has an impact on the syntax of the language: we have in ArcAngel*C* extra structural combinators that support the application of tactics to specific components of any *Circus* program. Most importantly, however, we have a change in the semantic model. We need to introduce, for example, the facility to define polymorphic tactic combinators to avoid the need to program the same tactics for programs, processes, and actions. We do not need to extend the syntax of ArcAngel to accommodate this facility, but need to generalise the semantic model. Additionally, ArcAngel*C* provides tactical support to handle the proof obligations generated by a refinement, as well as the program development itself.

In [34], we have presented novel tactic combinators for ArcAngel*C*. Here, besides an informal introduction to the language, we present its semantics in Z. Of particular interest is the strategy for specification of the semantics of structural combinators, which is of relevance for other tactic languages as well. Our formalisation fosters mechanisation in theorem provers like Z/EVES [51] and ProofPower-Z [41], and reasoning about ArcAngel*C* algebraic laws. We focus on novel aspects of the ArcAngel*C* semantics; a full account can be found elsewhere [33]. We also discuss here our own mechanisation based on ProofPower-Z.

In [6], we have presented a refinement strategy to prove the correctness of implementations of Simulink diagrams [17] in Ada. This strategy is divided into four parts, which deal with different aspects of the specification. In [34], we use ArcAngel*C* to formalise and generalise the first part of this refinement strategy. Here, we extend this work by also providing the formalisation of the second part of this strategy. This formalisation provides structure and abstraction to the refinement strategy, and permits its automation.

In summary, this paper presents the following novel contributions.

(i) The use of a tactic language to formalise an elaborate refinement strategy that is being developed to be applied to real-world large systems. This formalisation gives a clear route to automation.
(ii) A formalisation of a tactic language *for refinement* based on angelic choice using a formal language as meta-language (namely, Z). This opens the possibility of using a theorem prover to prove the tactic laws initially presented for Angel that are also valid for ArcAngel*C*. In addition, we have a general strategy for the formalisation of structural combinators.
(iii) An approach that allows different types of syntactic constructs to be tackled by the tactics, as well as the proof obligations generated by the refinements. As far as we know, all tactic languages in the literature allow the manipulation of only one type of construct.

We also discuss automatic support for the application of ArcAngel*C* tactics.

The next section describes *Circus*. In Section 3, ArcAngel*C* is presented. In Section 4 we describe control diagrams and a refinement strategy to prove that a given Ada program correctly implements a diagram [6]. In Section 5, we formalise parts of the refinement strategy as ArcAngel*C* tactics and use them in the verification of a simple controller. The semantics of ArcAngel*C* is described in Section 6. Section 7 discusses our mechanisation. Finally, in Sections 8 and 9, we discuss related and future work.

2. *Circus*

In *Circus*, a program is a sequence of paragraphs: a channel declaration, a Z paragraph, or a process definition. A process contains its own state, and communicates with the environment via channels. The main constructs are illustrated in the specification below of a register. It stores a value, which is initialised to 0, and can store or add a value to its current value. The stored value can also be output or reset.

$$\textbf{\textit{channel}}\ store, add, out : \mathbb{N};\ result, reset$$

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

Channel declarations $\textbf{\textit{channel}}\ c : T$ introduce a channel c that communicates values of type T. In our example above, we declare three different channels *store*, *add*, and *out*, which communicate natural numbers.

Processes may be declared in terms of other processes or explicitly. An explicit definition, like that of *Register* above, 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 the process *Register* is defined by the Z schema *RegSt*; it contains a component *value* that records the content of the register.

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 actions may be defined using Z schemas, guarded commands, or invocations to other local actions. Finally, actions can be combined using CSP operators like hiding, sequence, external and internal choice, parallelism, interleaving, or their corresponding iterated variants.

The process *Register* initialises its state component *value* to zero, and then exhibits 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 output through *out*, or *reset*.

Circus prefixing and guarded actions are as in CSP. For example, in an action of the form $p \mathbin{\&} c?x \longrightarrow A(x)$, we have a guard given by the predicate p. If the guard is true, the input prefixing assigns the value that is read through the channel c to a new implicitly declared variable x; it deadlocks otherwise.

Besides the set of channels on which the actions synchronise, the parallelism requires additional information to avoid conflicts in writing to variables: two partitions of all variables in scope. In $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2$, the actions synchronise on the channels in the set cs and have access to the initial values of all variables in scope. However, A_1 and A_2 may modify the values of only the variables in ns_1 and ns_2, respectively. The interleaving $A_1 \lll ns_1 \mid ns_2 \ggg A_2$ has a similar behaviour, but the actions do not synchronise on any channel.

Parametrised actions (and processes) and their instantiations are also available in *Circus*. When applied to actions, the renaming operator substitutes state components and local variables.

In *Circus*, the basic notion of refinement is that of action refinement [42], which is also used to define process refinement. In our examples, we take advantage of refinement laws from [32], like Law 21 (par-inter) below, which can be used to transform a parallel composition into an interleaving.

Law 21 (par-inter) $A_1 \llbracket ns_1 \mid cs \mid ns_2 \rrbracket A_2 = A_1 \lll ns_1 \mid ns_2 \ggg A_2$
provided $(usedC(A_1) \cup usedC(A_2)) \cap cs = \varnothing$.

The **provided** clause describes the conditions that need to be met in order to permit the application of the corresponding refinement law. For instance, the application of Law 21 (par-inter) 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. Conditions like these raise proof obligations when the law is applied.

By way of illustration, we consider the following *Circus* action, which is a parallel composition of two *Circus* actions with $\{\!|\, c_3 \,|\!\}$ as the synchronisation channel set: one of the parallel actions synchronises on a channel c_1, and the other outputs a value a on a channel c_2.

$$c_1 \longrightarrow x := 1 \llbracket \{x\} \mid \{\!|\, c_3 \,|\!\} \mid \{y\} \rrbracket c_2!a \longrightarrow y := 1$$

Intuitively, since none of these actions communicate on c_3, this parallel composition may be converted into an interleaving. Law 21 (par-inter) can be used to achieve this result.

$$\sqsubseteq \qquad \text{[Law 21 (par-inter)]}$$

$$c_1 \longrightarrow x := 1 \lll \{x\} \mid \{y\} \ggg c_2!a \longrightarrow y := 1$$

The condition $(\{c_1\} \cup \{c_2\}) \cap \{c_3\} = \varnothing$ is trivially true. Hence, the law application is valid.

Process refinement is defined in terms of action refinement: a process P_2 refines a process P_1, written $P_1 \sqsubseteq_{\mathcal{P}} P_2$, if its main action, which is denoted by $P_2.Act$, refines the main action of P_1, that is, $P_1.Act$. These main actions may work on different states, namely, those defined by P_1 and P_2, and so may not be comparable. Hence, we compare the actions that we obtain by hiding (that is, existentially quantifying) the state components $P_1.State$ and $P_2.State$ of P_1 and P_2, as if they were declared in a local variable block.

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

Since the state of a process is private, it may be changed during refinement. This can be achieved in much the same way as we can data-refine variable blocks and modules in imperative programs [27].

An extensive collection of refinement laws for processes and actions can be found in [32]. Strategies for stepwise development of concurrent programs [8], and for verification of control systems [7] are also available. In the next section, we describe ArcAngel*C*, which can be used to formalise these strategies.

3. ArcAngel*C*

ArcAngel*C* includes basic tactics, like a law application, for example; tacticals, which are general tactic combinators; and structural combinators, which support the application of tactics to components of *Circus* programs. The basic tactics and tacticals of ArcAngel*C* are inherited from Angel, and some of its structural combinators are inherited from ArcAngel. The

TacticDecl	::=	**Tactic** N (Decl) Tactic [**generates** Prog] [**proof obligations** Pred $^+$] **end**	[*tactic declaration*]
Tactic	::=	**law** N (Exp*)	[*law application*]
	|	**tactic** N (Exp*)	[*tactic application*]
	|	**skip** | **fail** | **abort**	[*basic tactics*]
	|	**applies to** Prog **do** Tactic	[*pattern matching*]
	|	Tactic ; Tactic | Tactic "|" Tactic	[*sequence/alternative*]
	|	μ_T N • Tactic | ! Tactic	[*recursion/cut*]
	|	**succs** Tactic | **fails** Tactic	[*assertions*]
	|	$\boxed{\longrightarrow}$ Tactic | $\boxed{\&}$ Tactic	[*action combinators*]
	|	$\boxed{\mu}$ Tactic | $\boxed{\textbf{\textit{if}}}$ Tactic $^+$ $\boxed{\textbf{\textit{fi}}}$ | $\boxed{\textbf{\textit{var}}}$ Tactic	
	|	$\boxed{\textbf{\textit{val}}}$ Tactic | $\boxed{\textbf{\textit{res}}}$ Tactic | $\boxed{\textbf{\textit{vres}}}$ Tactic	
	|	$\boxed{\textbf{beginend}}$ ((N, Tactic)*, Tactic)	[*process combinators*]
	|	$\boxed{\odot}$ Tactic | $\boxed{\odot_{inst}}$ Tactic	
	|	$\boxed{\widehat{=}}$ Tactic | Tactic $\boxed{;}$ Tactic	[*action/process combinators*]
	|	Tactic $\boxed{\Box}$ Tactic | $\boxed{\sqcap}$ Tactic	
	|	Tactic $\boxed{\|\|}$ Tactic | Tactic $\boxed{\|\|\|}$ Tactic	
	|	$\boxed{;_i}$ Tactic | $\boxed{\Box_i}$ Tactic | $\boxed{\sqcap_i}$ Tactic | $\boxed{\|\|_i}$ Tactic	
	|	$\boxed{\|\|\|_i}$ Tactic | $\boxed{\setminus}$ Tactic | $\boxed{:=}$ Tactic	
	|	$\boxed{\bullet}$ Tactic | $\boxed{\bullet_{inst}}$ Tactic	
	|	**program** (N, Tactic)*	[*program combinator*]

Fig. 1. Abstract syntax of ArcAngel*C*.

structural combinators related to the CSP constructs of *Circus* are a new feature. Furthermore, unlike ArcAngel tactics, which can be applied to programs only, ArcAngel*C*'s tactics can be applied to *Circus* programs, processes, and actions. Additionally, tactics can be used to discharge proof obligations: those containing action refinement statements.

The syntax of ArcAngel*C* is presented in Fig. 1. We use Exp* to denote a possibly empty sequence of elements of the syntactic category Exp of expressions. The bar used in the tactic language for alternation is smaller than the one used in the BNF notation in Fig. 1. For extra emphasis, we have written the bar of the tactic language in quotes. We use Tactic$^+$ to denote a non-empty sequence of tactics. The categories N, Pred, and Decl are those of the Z identifiers, predicates, and declarations defined in [43]. Finally, the syntactic category Prog is that of the *Circus* programs as in [32].

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 by a successful application of the tactic.

As an example, we present the skeleton of the definition of a tactic interIntroAndSimpl, whose behaviour T is detailed later in this section. It generates an action A, also described later, and has a single proof obligation $usedC(A) \cap cs = \varnothing$, which states that the sets $usedC(A)$ and cs must be disjoint.

Tactic interIntroAndSimpl( ) $\widehat{=}$ T
generates A
proof obligations $usedC(A) \cap cs = \varnothing$
end

It is possible to check the consistency of the clauses **proof obligations** and **generates**. There are two options: (i) remove these clauses and calculate them after the tactic compilation; or (ii) allow their inclusion and check them on tactic compilation. Both approaches involve symbolic application of laws.

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 (process, or action) p, the application succeeds: a new program (process, or action) is derived, 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 a previously defined tactic named n with arguments a 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 2 (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 obligations are generated. Law 2 (copy-rule-action), and all other refinement laws used in this paper can be found in Appendix B.

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

3.3. Tacticals

The tactic **applies to** p **do** t is given a meta-program (or program pattern) p that characterises the programs to which the tactic t is applicable; the meta-variables used in p can then be used in t. For example, the meta-program $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket$ ***Skip*** characterises those parallel compositions whose right-hand action is ***Skip***; here, A, ns_1, cs and ns_2 are the meta-variables. We consider as an example a refinement tactic that transforms a parallel composition into an interleaving: **applies to** $A \llbracket ns_1 \mid cs \mid ns_2 \rrbracket$ ***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, 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 can remove a parallel composition by first transforming it into an interleaving using Law 21 (par-inter), and then simplifying this interleaving using the unit law for interleaving, Law 12 (inter-unit). The two law applications occur in sequence. The tactic interIntroAndSimpl presented below formalises this simple strategy. It applies to parallel compositions in which the right-hand action is ***Skip*** and returns the action A and the proof obligation originated from the application of inter-unit.

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

We include the optional clauses **proof obligations** and **generates**, which simply document the results and proof obligations that are produced by applying the underlying tactic.

Tactics can also be combined in alternation: $t_1 \mid t_2$. First t_1 is applied. If that succeeds, then the composite tactic succeeds; otherwise t_2 is applied. If then 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 local variables declared in a main action to state components. This is the result of an application of either Law 28 (prom-var-state) or Law 29 (prom-var-state-2) depending on whether the process has state or not.

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

The standard form of angelic choice is commutative. ArcAngel$C$'s choice, however, as denoted by the alternation operator is not, as this gives more control to tactic programs. It provides an angelic choice that is implemented through backtracking: on failure, 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 possible through the cut operator: the tactic $!\,t$ behaves like t, except that it only considers the first successful application of t. If a subsequent tactic application fails, the whole tactic fails. As an example, we consider the small extension to promoteVars presented below in which the application of a given tactic T follows the alternation.

Tactic promoteVarsExt( ) $\widehat{=}$ (**law** prom-var-state() | **law** prom-var-state-2()) ; **tactic** T() **end**

If **law** prom-var-state() succeeds, but **tactic** T() subsequently does not, there is no point in backtracking to apply **law** prom-var-state-2(), and then try **tactic** T() again. Instead, we should cut the search and define the tactic as !(**law** prom-var-state() | **law** prom-var-state-2()) ; **tactic** T().

ArcAngel$C$ has a fixed-point operator to define recursive tactics. Using μ_T, we can define the tactic below, which exhaustively applies a given tactic t, terminating with success when its application eventually fails.

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

Recursion may lead to nontermination, in which case the result is the same as that of the basic tactic **abort**.

Two tactics are used to test the outcome of applying a tactic. The tactic **succs** t behaves like **skip** whenever t succeeds and fails whenever t fails. On the other hand, **fails** t behaves like **skip** if t fails, and fails if t succeeds. If the application of t runs indefinitely, then these tacticals behave like **abort**.

A simple example is the tactic **succs**(**law** par-comm()) ; t, which applies t only if the program is a parallel composition. This tactic first checks whether the given program is a parallel composition by trying to apply the commutativity Law 19 (par-comm), which applies only (and always) to parallel compositions. Only if the application succeeds, the tactic applies t to the given program.

3.4. Structural combinators

Often, we want to apply tactics to parts of a program; this is supported by structural tactic combinators. In [35], we define combinators for sequential programs. ArcAngel*C* provides additional combinators; essentially, there is one for each *Circus* program, process, or action constructor (see Fig. 1). Among these, only those for alternation (***if*** _ ***fi***), variable blocks (***var***), argument declaration (***val*** , ***res*** , ***vres***), and sequence (;) are also part of ArcAngel. For all the structural combinators, if the application of a tactic to a component program, process, or action fails or aborts, then so does the application of the whole tactic. In the case of n-ary structural combinators, if one tactic aborts so does the whole structural combinator tactic. Similarly, if one tactic fails and no other tactic behaves abortively, again the whole tactic fails.

Action structural combinators are those that allow us to apply a tactic to parts of a *Circus* action. The first that we present enables us to apply a tactic to a prefixing action. The tactic $\boxed{\longrightarrow} t$ applies to actions of the form $c \longrightarrow A$. It returns the prefixing $c \longrightarrow B$, where B is the program obtained by applying t to A; the proof obligations generated are those arising from the application of t. For example, if applied to the *Circus* action $c \longrightarrow A_1 \fatsemi (A_2 \fatsemi A_3)$, the tactic $\boxed{\longrightarrow}$ **law** seq-assoc() applies the associativity Law 31 (seq-assoc) to $A_1 \fatsemi (A_2 \fatsemi A_3)$ and returns the prefixing $c \longrightarrow (A_1 \fatsemi A_2) \fatsemi A_3$. The law application raises no proof obligations, and for this reason the tactic application does not yield any proof obligations either.

The combinator $\boxed{\&} t$ applies to a guarded action $g \,\&\, A$ and returns the result of applying t to A; the guard is unaffected in the resulting action. It is not possible to apply a tactic to the guard g since refinement laws can only be applied to actions, processes, and programs. If the guard g needs to be replaced, a law that applies to the whole guarded action $g \,\&\, A$ must be used; the guard combinator is not needed. By way of illustration, we consider the application of the tactic $\boxed{\&}$ **law** seq-assoc() to the action $g \,\&\, A_1 \fatsemi (A_2 \fatsemi A_3)$. It returns the guarded action $g \,\&\, (A_1 \fatsemi A_2) \fatsemi A_3$ and no proof obligations.

For recursive actions $\mu X \bullet A(X)$, there is the structural combinator $\boxed{\mu} t$. It returns the recursion obtained by applying t to the body $A(X)$ of the recursion.

For alternation, there is the structural combinator $\boxed{\textbf{\textit{if}}}\, t_1 \,\boxed{[]}\, \ldots \,\boxed{[]}\, t_n \,\boxed{\textbf{\textit{fi}}}$, which applies to an alternation $\textbf{\textit{if}}\ g_1 \longrightarrow A_1 \,[]\, \ldots \,[]\, g_n \longrightarrow A_n\ \textbf{\textit{fi}}$. It returns the result of applying each tactic t_i to the corresponding action A_i. For example, if we apply the tactic

$$\boxed{\textbf{\textit{if}}}\ \textbf{law}\ \text{assign-intro}(x := -1)\ \boxed{[]}\ \textbf{law}\ \text{assign-intro}(x := 1)\ \boxed{\textbf{\textit{fi}}}$$

to the action

$$\textbf{\textit{if}}\ a \leq b \longrightarrow x : [\, x' < 0 \,] \,[]\, a > b \longrightarrow x : [\, x' > 0 \,]\ \textbf{\textit{fi}}$$

we obtain the action $\textbf{\textit{if}}\ a \leq b \longrightarrow x := -1 \,[]\, a > b \longrightarrow x := 1\ \textbf{\textit{fi}}$, and two proof obligations, namely, $\mathit{true} \Rightarrow -1 < 0$ and $\mathit{true} \Rightarrow 1 > 0$. The action $x : [\, x' < 0 \,]$ is a specification statement in the style of Morgan's refinement calculus [26], also used in the Z refinement calculus [9]. It specifies an action that changes the value of x to an arbitrary negative number. Similarly, $x : [\, x' > 0 \,]$ sets x to a positive number.

The structural combinator $\boxed{\textbf{\textit{var}}}\, t$ applies to a variable block; it applies t to the body of the block. To give an example, if we apply the tactic $\boxed{\textbf{\textit{var}}}$ **law** assign-intro($x := 10$) to $\textbf{\textit{var}}\ x : \mathbb{N} \bullet x : [\, x \geq 0 \,]$, we get $\textbf{\textit{var}}\ x : \mathbb{N} \bullet x := 10$ and the proof obligation $\mathit{true} \Rightarrow 10 \geq 0$.

Finally, to apply a tactic to the body of a parametrised action, we have the combinators $\boxed{\textbf{\textit{val}}}\, t$, $\boxed{\textbf{\textit{res}}}\, t$, and $\boxed{\textbf{\textit{vres}}}\, t$, for parameters passed by value, result, or value-result, respectively.

Process structural combinators are used to cater for process definitions. To apply tactics to components of an explicit process declaration, we can use the structural combinator $\boxed{\textbf{beginend}}$. It receives two arguments: a possibly empty sequence of pairs (n, t) of names n and tactics t, and a tactic. For each (n, t) in the sequence, this combinator applies t to the paragraph named n; the second argument is applied to the main action. For example, the tactic $\boxed{\textbf{beginend}}\, (\langle (\mathit{RegCycle}, \textbf{tactic}\ T_1()) \rangle, \textbf{tactic}\ T_2())$ can be applied to the process *Register* presented in the previous section. It applies the tactic T_1 to the body of the action *RegCycle*, and the tactic T_2 to the main action of *Register*.

Action and process structural combinators. Most of the *Circus* constructs originating from CSP can be used in the definition of both processes and actions. For each of these constructs we define a single combinator. Their application is oblivious to whether we are applying the tactic to an action or a process.

The tactic $t_1 \,\boxed{;}\, t_2$, for example, applies to actions or processes $p_1 \fatsemi p_2$. It generates the sequential composition of the actions (or processes) obtained by applying t_1 to p_1 and t_2 to p_2; the proof obligations are those arising from both

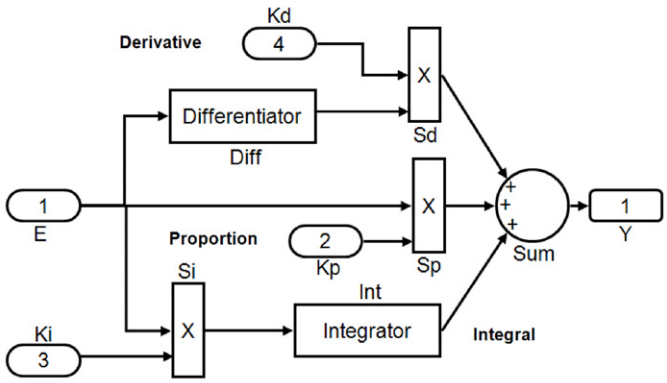

Fig. 2. PID (Proportional Integral Derivative) controller.

tactic applications. This combinator is used in Section 5; for instance, one of the steps of the refinement strategy is **skip** $[;]$ **tactic** interIntroAndSimpl() (see page 806 for more details). This tactic applies to a sequential composition: the first action is unchanged and the tactic interIntroAndSimpl is applied to the second action. Similar combinators are available for external choice ($t_1 [\Box] t_2$), internal choice ($t_1 [\sqcap] t_2$), parallel composition ($t_1 [\|] t_2$), interleaving ($t_1 [|||] t_2$), event hiding ($[\setminus]\ t$), and renaming ($[:=]\ t$).

We also have a corresponding structural combinator for each of the indexed CSP constructs. For instance, $[;_i]\ t$ can be applied to an indexed sequential composition of the form $(;\ d \bullet p)$. The result is that obtained by the application of t to p. 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 $(;\ i : 0 \,..\, \#sq \bullet s : [s' = s + sq(i)])$. If we apply $[;_i]$ **law** assign-intro$(s := s + sq(i))$, we get the action $(;\ i : 0 \,..\, \#sq \bullet s := s + sq(i))$ and proof obligations $true \Rightarrow s + sq(i) = s + sq(i)$, for every i in $0 \,..\, \#sq$. We also have the combinators $[\Box_i]$ for indexed external choices, $[\sqcap_i]$ for indexed internal choices, $[\|_i]$ for indexed parallel composition, and $[|||_i]$ for indexed interleaving.

Program structural combinator. We have just one combinator for programs. 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 (n, t) in the sequence, it applies the tactic t to the paragraph named n in the *Circus* program. The tactics used in our case study in Section 5 illustrate the use of this constructor.

This concludes the description of the ArcAngel*C* constructs. Using ArcAngel*C*, we are able to write complex tactics as the one presented in Section 5. Tactics cannot be inconsistent, in the sense that they cannot generate incorrect implementations. Their soundness, or more specifically, the correctness of the refinement, is guaranteed by the soundness of the refinement laws, provided the proof obligations can be discharged.

4. A refinement strategy for verification of control system implementations

Control systems are often used in safety-critical applications and their verification has been of great interest. In [6], we present an approach in which we aim at proof of correctness of code, as opposed to validation of requirements or designs. We give a semantics to discrete-time Simulink diagrams [17] using *Circus*, and propose a verification technique for parallel Ada implementations. In this section, we briefly describe this verification technique based on a refinement strategy that has already been applied to industrial examples. The formalisation of the refinement strategy is the subject of Section 5.

Control diagrams model systems as directed graphs of blocks interconnected by wires. Simulink is a popular tool for drawing and analysing such diagrams, and generating code; its use in the avionics and automotive sectors is very widespread. A simple example of a Simulink diagram is presented in Fig. 2; it contains a PID (Proportional Integral Derivative) controller, a generic control loop feedback mechanism that attempts to correct the error between a measured process variable and a desired set-point by calculating and then outputting a corrective action that can adjust the process accordingly.

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

Typically, a block takes input signals and produces outputs according to its characteristic 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 subordinate diagrams of the model. For example, the block Diff in Fig. 2 corresponds to a subsystem defined in the diagram presented in Fig. 3.

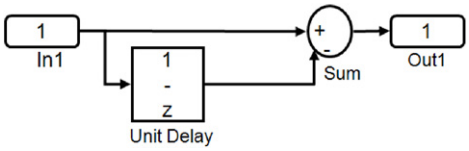

Fig. 3. PID differentiator.

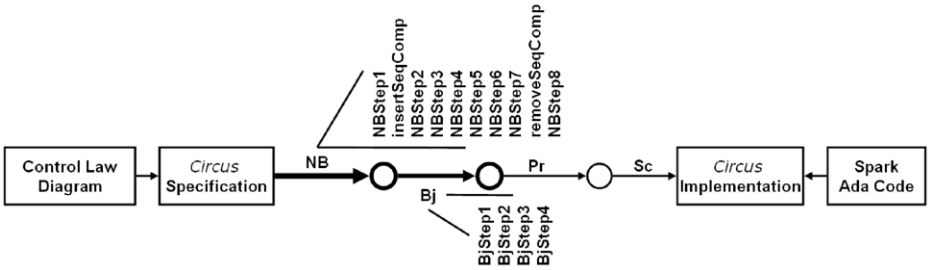

Fig. 4. The refinement strategy.

Blocks can have state. For instance, in Fig. 3, the block Unit Delay stores the value of the input signal In1, and outputs the value stored in the previous cycle. The block Diff outputs a signal Out1 that corresponds to the difference of the current input In1 and the output of the Unit Delay block.

4.1. The refinement strategy

In [6], we present a technique to verify Ada programs with respect to Simulink diagrams using *Circus*. The approach is based on calculating a *Circus* model for the diagram using the semantics given in [6], calculating a *Circus* model for the Ada program, and proving that the former is refined by the latter.

In the *Circus* model of the diagram, each block is represented by a process, and the diagram by a parallel composition of such processes. A channel *end_cycle* records the end of a cycle of execution, and keeps all the block processes in synchrony. A more detailed account of this model and an example are given in the following section. The refinement proof is achieved using a strategy that comprises the following four phases:

NB—Normalise Blocks: For each block, refine the corresponding *Circus* process in the diagram model to write its main action in a normal form: a recursion that iteratively executes an action that captures the behaviour of a cycle as an interleaving of inputs, followed by output calculations and state update, followed by an interleaving of outputs, and synchronisation on *end_cycle*.

BJ—Blocks Join: Collapse (that is, remove) the parallelism between the processes of the blocks that are implemented by a single procedure in the Ada program. Each of the resulting processes should be refined to put them back into the normal form described in phase NB.

Pr—Procedures: For each of the processes created in phase BJ, introduce the action in the model of the program that specifies the corresponding procedure, and prove that the calculations of the outputs and of the new value of the state can be refined by a call to that action.

Sc—Scheduler: Refine the process that corresponds to the system to get the parallel programs of the implementation. Unless the program explores all the parallelism in the diagram, this involves collapsing some of the remaining parallel processes. The same approach taken in the phase BJ can be adopted.

Fig. 4 illustrates the overall approach. In Section 5 we formalise the first two phases: NB and BJ. In this formalisation, we split each phase of the strategy into small steps, which correspond to those presented in [6]. For instance, the phase BJ is split into four steps: BJStep1 to BJStep4.

4.2. The Circus model

The *Circus* model of a diagram can be generated automatically. It includes components that cater for examples, which are much more complex than our PID. The refinement strategy that we formalise in the sequel in fact copes with these more complex models too.

As previously mentioned, in the model of the diagram, there is a (basic) *Circus* process for each block, and the diagram itself is specified by a 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 an 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 that 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.

To illustrate the construction of the formal model, we describe the translation of the block Diff shown in Fig. 3.[1] The block is translated into a *Circus* process with its original name.

$$\textbf{\textit{process}}\ Diff\ \widehat{=}\ \textbf{\textit{begin}}$$

The state of *Diff*, defined by *Diff_State*, includes a component that stores the state of the block Unit Delay.

$$\textbf{\textit{state}}\ Diff\_State\ \widehat{=}\ [\, pid\_Diff\_UnitDelay\_St : \mathbb{U}\,]$$

Next, the schema *pid_Diff* characterises the PID Diff block. The components of a schema that models a block represent its inputs, outputs, and, in the case of subsystem blocks, the blocks of the subsystem diagram. For the inputs, we have components *In*1?, *In*2? and so on, depending on the number of inputs of the block. Similarly, for the outputs, we have components *Out*1!, *Out*2!, and so on. The block in our example has one input and one output. The schema modelling the block Diff also involves types *pid_Diff_Sum* and *pid_Diff_UnitDelay*, because it has a component *Sum* and a component *UnitDelay* of these types, respectively. Whereas *pid_Diff_Sum* and *pid_Diff_UnitDelay* are schemas (presented below) that describe the general behaviour of those types of blocks, *Sum* and *UnitDelay* are particular instances of them.

pid_Diff

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

The predicate determines how the wires are connected: the input In1 of the block Diff is equated with the input In1 of the blocks UnitDelay and Sum; the output Out1 of the block UnitDelay is equated with the input In2 of the block Sum; and the output Out1 of the block Sum is equated with the output Out1 of Diff.

The types *pid_Diff_Sum* and *pid_Diff_UnitDelay* are defined using a formalisation of the Simulink block library that provides a schema definition for each block [1]. The schema *Sum_PM* specifies a sum block that negates its second input; in effect, this is a subtraction.

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

The function *UnitDelay_g* takes the initial value of its state as its argument. It is a (generic) function that takes a binding with a single component *X*0 specifying the type and initial value of the stored data item, and yields a set of bindings (defined by a schema) that characterises the unit delay block.

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

Next, we have the initialisation of the state of the process *Diff*. The component *pid_Diff_UnitDelay_St* is initialised with the value of the component *initial_state* of the bindings in *pid_Diff_UnitDelay*.

Init

$$\begin{array}{l} Diff\_State' \\ \hline \exists\, b : pid\_Diff\_UnitDelay \bullet pid\_Diff\_UnitDelay\_St' = b.initial\_state \end{array}$$

In a type that represents a block with state, components *state*, *state*′, and *initial_state* record the value of the state at the beginning and at the end of each cycle, and at the beginning of the first cycle. Above *b.initial_state* refers to the initial state of the *pid_Diff_UnitDelay*.

The operation *Calc_Diff* lifts *pid_Diff* to an operation over *Diff_State*: that is the state of the *pid_Diff* *Circus* process rather than the one encapsulated in the *pid_Diff* schema. For that, we establish a correspondence between the *state* and *state*′ components of the bindings in the components *UnitDelay* of *pid_Diff* and the state components *pid_Diff_UnitDelay_St* and *pid_Diff_UnitDelay_St*′.

[1] The complete *Circus* model of the PID controller can be found at http://www.cs.york.ac.uk/circus.

Calc_Diff

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

$$\exists b : pid\_Diff \bullet b.In1? = In1? \land b.Out1! = Out1! \land b.UnitDelay.state = pid\_Diff\_UnitDelay\_St \land b.UnitDelay.state' = pid\_Diff\_UnitDelay\_St'$$

Each flow in a block is modelled by a *Circus* action that calculates the value of the output that determines the flow. In our example, as shown in Fig. 3, the block Diff has a single flow that calculates the value output through the channel *Diff_out*. We, therefore, define an action *Calc_Diff_Out*. It is specified in terms of *Calc_Diff* using the schema calculus: we hide the final value of the state and conjoin the result with $\Xi Diff\_State$ so that the state is not modified.

$$Calc\_Diff\_Out \mathrel{\widehat=} Calc\_Diff \setminus (pid\_Diff\_UnitDelay\_St') \land \Xi Diff\_State$$

For each flow of execution f, an action *Exec_f* is provided that takes the required inputs, and calculates and produces the outputs. The name f is determined by the unique outputs that the flow produces. The inputs are received in any order; similarly, outputs are sent in any order. In our example, we have the action *Exec_Diff_out*, which uses one input variable *In*1, and one output variable *Out*1. The value x of the input is recorded in a corresponding variable *In*1. Since there is only one input and one output, there is no interleaving: it receives an input through channel E and produces an output through channel *Diff_out*.

$$Exec\_Diff\_out \mathrel{\widehat=} \textbf{var}\ In1 : \mathbb{U} \bullet E?x \longrightarrow In1 := x \fatsemi \textbf{var}\ Out1 : \mathbb{U} \bullet Calc\_Diff\_Out \fatsemi Diff\_out!Out1 \longrightarrow \textbf{Skip}$$

The action *Flows* combines all the flows of execution in parallel. Since in *Diff* there is only one flow, the otherwise required parallelism in the action *Flows* is reduced to the action *Execute_Diff_out*.

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

The schema *Calc_Diff* is also used to specify the operation *Calc_Diff_State* that changes the state of the block. In this case the output variable *Out*1! of *Calc_Diff* is hidden.

$$Calc\_Diff\_State \mathrel{\widehat=} Calc\_Diff \setminus \{Out1!\}$$

The action *Diff_StUpdt* reads the input through E and executes *Calc_Diff_State* to update the state.

$$Diff\_StUpdt \mathrel{\widehat=} \textbf{var}\ In1 : \mathbb{U} \bullet E?x \longrightarrow In1 := x \fatsemi Calc\_Diff\_State$$

The main action starts with the initialisation, and recursively proceeds in parallel to execute the flows and update the state, before synchronising on *end_cycle*. In *Diff*, there is only one flow, so the parallelism reduces to a single action *Exec_Diff_out* that synchronises with *Diff_StUpdt* on input signal E.

$$\bullet\ Init \fatsemi \mu X \bullet (Flows \llbracket \{\} \mid \{\!|E|\!\} \mid \{pid\_Diff\_UnitDelay\_St\} \rrbracket Diff\_StUpdt) \fatsemi end\_cycle \longrightarrow X$$

end

This concludes the *Circus* model of the block Diff of our example, which is used in the next section to illustrate the application of the ArcAngelC tactics that formalise the refinement strategy.

5. Case study—the tactics NB and BJ

In this section, we present the tactics NB and BJ that formally describe the first two phases of the refinement presented in Section 4. Their application to the example presented here is also discussed.

5.1. *Phase* NB

In [6], we explain the NB phase. Informally, its steps are described as follows: to normalise the model of a block we remove the parallelism between the actions that model the flows 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 with syntactic checks. Given the way in which models are constructed, as described in the previous section, flows share their inputs as depicted in Configuration 4 in Fig. 5. Additionally, the state update is also combined in this way with the flows.

Formally, the first step of this phase is a series of applications of the refinement Law 2 (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

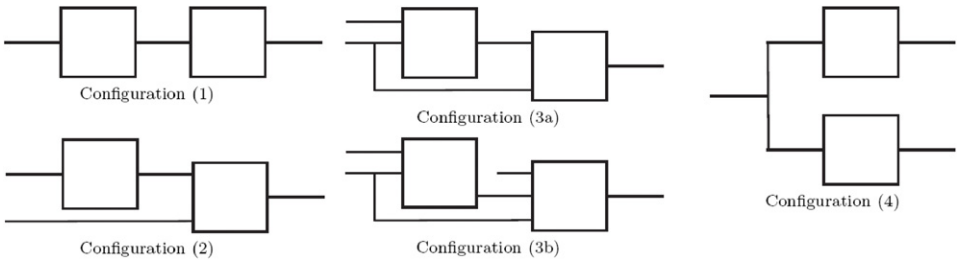

Fig. 5. Blocks configurations.

in its definition. The first one, TRY, is presented and discussed in Section 6.3. The next tactic is used to apply in sequence repeatedly a given law l using the elements of a given list *args* as arguments. It uses the tactic TRY in order to skip when it reaches the base case, an empty list of arguments.

Tactic APPLYL(l, *args*) $\widehat{=}$ TRY(**law** I(*hd args*) ; APPLYL(l, *tl args*)) **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 definitions of tactics and functions that we omit here can be found in [33].

The tactic below formalises the series of applications of the Law 2 (copy-rule-action). It receives a list *fs* of the names of the actions *Exec_*f that execute the flows as arguments. It applies to explicit process definitions, and transforms the process using the Law 2 (copy-rule-action).

Tactic applyCopyRule(*fs*) $\widehat{=}$ **applies to** ***process*** $P \widehat{=}$ ***begin*** *PPars* • *Main* ***end***
do $\boxed{\widehat{=}}$ (**law** copy-rule-action("*Flows*") ;
APPLYL(copy-rule-action, *fs*) ;
TRY(**law** copy-rule-action($P \frown$ "*_StUpdt*")))
end

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

Tactic NBStep1(*fs*) $\widehat{=}$ **tactic** applyCopyRule(*fs*) **end**

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

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

Throughout this section, we box the target of the next refinement step as illustrated above.

5.1.1. Synchronise inputs

The action resulting from the application of the previous tactic models the behaviour of a block. In general, the inputs of the block that are needed by more than one flow of execution are shared, and all inputs are shared with the state update. This structure is followed by all block models, which are those to which the refinement strategy applies. Formally, this assumption is checked by the tactic syncInput below: if the inputs are not shared, the application of the Law 26 (par-seq-step-2) does not apply, and the tactic fails.

All flows in the main action should require all inputs, and so does the state update. Therefore, all parallel actions in the body of the recursion declare local variables d_{In} to hold each of the input values, and take all of them in interleaving in A_{In}. In our example, an interleaving is not needed because we have a single input. In this step, we extract from the parallelism the declarations d_{In}, using Law 36 (var-exp-par-2), and the interleaving A_{In}, using a law that distributes an action over a parallel composition, Law 26 (par-seq-step-2).

Tactic syncInput( ) $\widehat{=}$
applies to ($\textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In} \; ; \; A_{Out}$) $[\![ns_1 \mid cs \mid ns_2]\!]$ ($\textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In} \; ; \; A_{St}$)
do law var-exp-par-2() ; $\boxed{\textbf{var}}$ **law** par-seq-step-2()
generates $\textbf{var}\ d_{In} : \mathbb{U} \bullet A_{In} \; ; \; (A_{Out} \; [\![ns_1 \mid cs \mid ns_2]\!] \; A_{St})$
end

This tactic generates a program that declares the variables to hold the input values just once, takes the inputs just once, and then 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 of the form presented below in which the parallel composition of all the flows is in parallel with the state update. The actions I and EC stand for the initialisation in the main action and the synchronisation on *end_cycle*.

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

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

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

For example, the application of **tactic** $\mathsf{foldl}_{\parallel}$ (**tactic** syncInput()) to an instantiation of the generic case described above in which there are three flows is presented below. The tactic recurs until the point in which the application of the structural combinator $\boxed{\parallel}$ fails (lines 1–6), in which case, since we are in a TRY tactic, the tactic skips and returns ($\mathbf{var}\ d \bullet A_{In} \fatsemi A_{Out_2}$) (line 7). Then, the tactic applies **tactic** syncInput() to each result of 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).

$(\mathbf{var}\ d \bullet A_i \fatsemi A_{o_0}) \parallel ((\mathbf{var}\ d \bullet A_i \fatsemi A_{o_1}) \parallel (\mathbf{var}\ d \bullet A_i \fatsemi A_{o_2}))$	(1)	
$=$ [**tactic** TRY ((**skip** $\boxed{\parallel}$ (**tactic** $\mathsf{foldl}_{\parallel}$ (**tactic** syncInput()))) ; $\ldots$)]	(2)	✓
$(\mathbf{var}\ d \bullet A_i \fatsemi A_{o_1}) \parallel (\mathbf{var}\ d \bullet A_i \fatsemi A_{o_2})$	(3)	
$=$ [**tactic** TRY ((**skip** $\boxed{\parallel}$ (**tactic** $\mathsf{foldl}_{\parallel}$ (**tactic** syncInput()))) ; $\ldots$)]	(4)	✓
$(\mathbf{var}\ d \bullet A_i \fatsemi A_{o_2})$	(5)	
$=$ [**tactic** TRY ((**skip** $\boxed{\parallel}$ (**tactic** $\mathsf{foldl}_{\parallel}$ (**tactic** syncInput()))) $\fatsemi$ $\ldots$)]	(6)	✗
$(\mathbf{var}\ d \bullet A_i \fatsemi A_{o_2})$	(7)	
$=$ [**tactic** TRY ($\ldots$; **tactic** syncInput())]	(8)	✓
$(\mathbf{var}\ d \bullet A_i \fatsemi (A_{o_1} \parallel A_{o_2}))$	(9)	
$=$ [**tactic** TRY ($\ldots$; **tactic** syncInput())]	(10)	✓
$\mathbf{var}\ d \bullet A_i \fatsemi (A_{o_0} \parallel (A_{o_1} \parallel A_{o_2}))$	(11)	

In the same way, we may use $\mathsf{foldl}_{\parallel}$ in the n-ary case to join all the variable declarations d and the interleaved A_i before the outermost parallel composition. This is achieved and formalised by the tactic that follows.

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

The process to which we need to apply this step may or may not have state: the main action of a process with state 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, synchronises the inputs of the whole action.

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

Stateless processes, however, do not have a parallel composition with a state update; in this case, 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 with ***Skip***. This unifies the structure of the actions that result from the application of this step, allowing the remaining tactics to be used for both of them.

Tactic NBStep2_l( ) $\mathrel{\widehat{=}}$ **tactic** joinFlowsInput() ; $\boxed{\mathbf{var}}$ (**skip** $\boxed{\fatsemi}$ **tactic** createPar()) **end**

The tactic createPar creates a parallel composition using Laws 12 (inter-unit) and 22 (par-inter-2) in sequence.

Finally, we define the tactic for the second step of the NB phase, NBStep2: it tries, in alternation, to apply the tactic for processes with state, and the tactic that applies to processes without states.

Tactic NBStep2( ) $\mathrel{\widehat{=}}$ **tactic** NBStep2_f() | **tactic** NBStep2_l() **end**

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

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

The next step of this phase expands the scope of the variable blocks that declare output variables.

5.1.2. 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 (although a flow can define several outputs), we can expand the scope of the output variable blocks and join the resulting nested blocks using Laws 35 (var-exp-par), 38 (var-exp-seq) and 16 (join-blocks).

As for the previous step, we need to define a tactic that supports multiple flows. At this point, the structure of the main action has a parallel composition of the form presented below, in which a parallel composition of variable blocks declares different output variables.

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

The strategy to define the tactic in this step is similar to that used before: we define a tactic, expDisjVarPar, which extracts both variable blocks from a parallel composition of two variable blocks, and joins them; we then use $\text{foldl}_{\parallel}$ to join all the variables blocks; and finally, define a tactic that expands the scope of the variable blocks to outside the parallel composition and A_{In}, and joins the variable blocks.

The tactic expDisjVarPar specified below applies to a parallel composition of two variables block whose sets of declared variables are disjoint. It applies Law 35 (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( ) $\widehat{=}$
  applies to $(\mathbf{var}\ d_0 \bullet A_0)\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ (\mathbf{var}\ d_1 \bullet A_1)$
  do **law** var-exp-par() ; $\boxed{\textbf{var}}$ (**law** par-comm() ; **law** var-exp-par() ; $\boxed{\textbf{var}}$ **law** par-comm()) ;
    law join-blocks()
  generates $\mathbf{var}\ d_0;\ d_1 \bullet (A_0\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A_1)$
end

Using this tactic, we can join all the variable declarations with the tactic joinFlowsOutVarScope below.

Tactic joinFlowsOutVarScope $\widehat{=}$ (**tactic** $\text{foldl}_{\parallel}$ (**tactic** expDisjVarPar()))$\boxed{\parallel}$**skip end**

The top-level structural combinator is necessary to avoid application of expDisjVarPar to the parallelism with the state update action; to that action, we apply the **skip** tactic.

Finally, we define the tactic expOutVarScope; intuitively, it applies to actions that declare the input variables, receive their values, and then, declare the output variables, and calculate and produce the outputs in parallel with the state update. First, using Law 35 (var-exp-par), we expand the scope of the variable blocks to outside the parallelism. Next, we introduce ***Skip*** to obtain an action in the form accepted by Law 38 (var-exp-seq), which is applied to move the variable declaration to include A_{In} in its scope. Finally, we remove the ***Skip*** and join both variable blocks. Formally, each equality law, like $A_1 = A_2$, for example, gives rise to a pair of refinement laws $A_1 \sqsubseteq A_2$ and $A_2 \sqsubseteq A_1$. We use the superscript b (for *b*ackwards) to refer to the second law.

Tactic expOutVarScope( ) $\widehat{=}$
  applies to $\mathbf{var}\ d \bullet A_{In} \mathbin{;} \ ((\mathbf{var}\ d_{Out} \bullet A_{Out})\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A_{St})$
  do $\boxed{\textbf{var}}$ $\left(\begin{array}{l}(\textbf{skip}\ \boxed{;}\ (\textbf{law}\ \text{var-exp-par}()\ ;\ \textbf{law}\ \text{seq-right-unit}()))\ ;\ \textbf{law}\ \text{var-exp-seq}()\ ; \\ \boxed{\textbf{var}}\ (\textbf{skip}\ \boxed{;}\ \textbf{law}\ \text{seq-right-unit}^b())\end{array}\right)$;
    law join-blocks()
  generates $\mathbf{var}\ d;\ d_{Out} \bullet A_{In} \mathbin{;} \ (A_{Out}\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A_{St})$
end

The result of this tactic 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 variable blocks in the left-hand action of the outermost parallel composition. Afterwards, 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( ) $\widehat{=}$ ($\boxed{\textbf{var}}$(**skip** $\boxed{;}$ **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.

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

The next step removes all schemas that calculate the outputs and updates the state from the parallelism.

5.1.3. Isolating the input processing

The fourth step isolates the communication of the outputs. In general, we now have a parallel composition like that below, in which the nested parallel composition of the flows is in parallel with the state update.

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

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

The tactic isolateSeqActions presented below applies to a parallel composition $(A_{C_0} \fatsemi A_{O_0}) \parallel (A_{C_1} \fatsemi A_{O_1})$. It applies Law 25 (par-seq-step) to remove the schema A_{C_0} from the parallel composition, resulting in a sequential composition whose left-hand side is A_{C_0}. Next, it commutes the remaining parallel composition and uses the Law 25 (par-seq-step) again to remove A_{C_1} from it. Finally, it commutes the parallel composition back into its original order, and applies the associativity law for sequence to aggregate A_{C_0} and A_{C_1}.

Tactic isolateSeqActions( ) $\widehat{=}$
  applies to $(A_{C_0} \fatsemi A_{O_0})\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ (A_{C_1} \fatsemi A_{O_1})$
  do **law** par-seq-step() ;
    (**skip** $\boxed{;}$ (**law** par-comm() ; **law** par-seq-step() ; (**skip** $\boxed{;}$ **law** par-comm()))) ;
    law seq-assoc()
  generates $(A_{C_0} \fatsemi A_{C_1}) \fatsemi (A_{O_0}\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A_{O_1})$
end

Using this tactic, we can isolate all the output communications A_{O_i} in the left-hand action of the outermost parallel composition. This is captured by the tactic joinFlowsCalc declared below.

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

Finally, we define isolateIn, which introduces a ***Skip*** into the right branch of the parallelism and then uses Law 25 (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 acts on the second part of this sequential composition: it commutes the parallel composition and then applies again Law 25 (par-seq-step) to remove the schema A_{St} that calculates the state. It commutes the remaining parallelism. Finally, it applies the associativity Law 31 (seq-assoc) to the sequential composition to aggregate the output calculation and the state update.

Tactic isolateIn( ) $\widehat{=}$
  applies to $(A_C \fatsemi A_O)\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A_{St}$
  do (**skip** $\boxed{\parallel}$ (**law** seq-right-unit())) ; **law** par-seq-step() ;
    (**skip** $\boxed{;}$ (**law** par-comm() ; **law** par-seq-step() ; (**skip** $\boxed{;}$ **law** par-comm()))) ;
    law seq-assoc()
  generates $(A_C \fatsemi A_{St}) \fatsemi (A_O\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ \mathbf{Skip})$
end

The objective is to apply isolateIn to the parallelism in the main action; there may be, however, many flows. Hence, we first isolate all the output communications in A_{Out}. Afterwards, we apply isolateIn. Finally, Law 31 (seq-assoc) isolates the parallel composition as the second part of a sequential composition.

Tactic NBStep4( ) $\widehat{=}$ (**skip** $\boxed{;}$ (**tactic** joinFlowsCalc() ; **tactic** isolateIn())) ; **law** seq-assoc() **end**

In our example, joinFlowsCalc immediately skips; isolateIn yields the following action.

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

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

5.1.4. Introducing and simplifying interleaving of outputs

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

Tactic NBSteps5_6( ) $\widehat{=}$ **skip** $\boxed{;}$ **tactic** interIntroAndSimpl() **end**

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

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

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

5.1.5. Extend scope of the variable declarations to the outermost level

At this stage, the main action's format is $I \fatsemi (\mu X \bullet (\textbf{var}\ d \bullet A) \fatsemi EC)$. We expand the scope of d to the outer level using the unit laws for sequence, and Laws 37 (var-exp-rec) and 38 (var-exp-seq) as follows. First, we introduce a ***Skip*** to the left of the sequential composition in the body of the recursion. Next, we expand the scope of d to the whole sequential composition in the body of the recursion (Law 38 (var-exp-seq)), remove the ***Skip*** that was introduced, and expand the scope of d over the recursion (Law 37 (var-exp-rec)). Finally, we introduce a ***Skip*** in the sequential composition of the main action, expand the scope of d to the whole sequential composition (Law 38 (var-exp-seq)), and remove the ***Skip*** that was introduced. At the end, we have $\textbf{var}\ d \bullet I \fatsemi (\mu X \bullet (A \fatsemi EC))$ as the main action.

Tactic extendVarScope( ) $\widehat{=}$
applies to $I \fatsemi (\mu X \bullet (\textbf{var}\ d \bullet A) \fatsemi EC)$
do (**skip** $\boxed{;}$ (($\boxed{\mu}$ (**law** seq-left-unit() ; **law** var-exp-seq() ; $\boxed{\textbf{var}}$ (**law** seq-left-unit^b()))) ; **law** var-exp-rec() ; **law** var-exp-seq())) ;
law var-exp-seq() ; $\boxed{\textbf{var}}$ (**skip** $\boxed{;}$ **law** seq-right-unit^b())
generates $\textbf{var}\ d \bullet I \fatsemi (\mu X \bullet (A \fatsemi EC))$
end

The simple application of extendVarScope represents the seventh step of the phase **NB**.

Tactic NBStep7( ) $\widehat{=}$ **tactic** extendVarScope() **end**

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

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

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

5.1.6. Promote local variables to state components

In this last step, the tactic NBStep8 simply invokes a tactic promoteVars to turn the input and output variables into state components. This concludes the application of the first phase of the refinement strategy, which, in our example, produces the new definition sketched below for the process *Diff*.

process *Diff* $\widehat{=}$ ***begin***
state *Diff_State* $\widehat{=}$ [*pid_Diff_UnitDelay_St* : $\mathbb{U}$; *In1* : $\mathbb{U}$; *Out1* : $\mathbb{U}$]
...
• $Init \fatsemi (\mu X \bullet \left(\begin{array}{l} ((E?x \longrightarrow In1 := x) \fatsemi (Calc\_Diff\_Out \fatsemi Calc\_Diff\_State)) \fatsemi \\ (Diff\_out!Out1 \longrightarrow \textbf{Skip}) \end{array}\right) \fatsemi end\_cycle \longrightarrow X$
end

There is one tactic NBStep*i*, for each of the steps *i* of the refinement strategy. We compose most of these tactics in the tactic NBMain. Furthermore, two auxiliary tactics are used in NBMain. As previously discussed, the process we are dealing with may have a state or not. The example presented here falls in the first case: its main action is a sequential composition of a schema that initialises the state and a recursion. In the second case, however, since there is no state to initialise, the main action is

just a recursion. In order to have the same structure (a sequential composition) in both cases, we use two auxiliary tactics, insertSeqComp and removeSeqComp. In the absence of a sequential composition, the tactic insertSeqComp introduces one with ***Skip*** as its first action, using Law 32 (seq-left-unit); otherwise, it skips.

Tactic insertSeqComp( ) $\widehat{=}$ TRY(**fails**(**skip** $\boxed{;}$ **skip**) ; (**law** seq-left-unit())) **end**

Complementarily, the second auxiliary tactic, removeSeqComp, which is presented below, removes any sequential composition with ***Skip*** using Laws 32 (seq-left-unit) and 33 (seq-right-unit).

Tactic removeSeqComp( ) $\widehat{=}$ TRY(**law** seq-left-unit^b()) ; TRY(**law** seq-right-unit^b()) **end**

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

Tactic NBMain( ) $\widehat{=}$
tactic insertSeqComp() ;
$\left(\textbf{skip} \boxed{;} \ \boxed{\mu} \left(\left(\begin{array}{l} \textbf{tactic}\ \text{NBStep2}() \ ; \ \textbf{tactic}\ \text{NBStep3}() \ ; \\ \boxed{\textit{var}}\ (\textbf{tactic}\ \text{NBStep4}() \ ; \ \textbf{tactic}\ \text{NBSteps5\_6}()) \end{array} \right) \boxed{;} \textbf{skip} \right) \right)$;
tactic NBStep7() ; $\boxed{\textit{var}}$ **tactic** removeSeqComp()
end

The tactic NBProc below can be applied to normalise a process that corresponds to an individual block: it takes the process name as its argument. First, it applies NBStep1 using as an argument a list that contains the names of the actions of the process that execute the flows; this is identified by the function FNames. Then, it applies the tactic NBMain to the main action of the process. Finally, it promotes the variables declared at the top level of the resulting main action to state components (NBStep8).

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

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

Tactic NB(*ind_blocks*) $\widehat{=}$ APPLYT(NBProc, *ind_blocks*) **end**

This tactic takes as argument a list of block names. In our example, $\langle Diff, Sd, Int, Si, Sp, Sum\rangle$ can be used to apply the phase NB to the whole *Circus* program that models the PID control diagram.

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

5.2. *Phase* BJ

After the NB phase, we have the phase BJ, in which we collapse the parallelism between the processes of the blocks that are implemented by a single procedure in the Ada program, and then between the processes that represent procedures handled by a single scheduler. Each of the resulting processes is then put back into the normal form used in the previous phase. The success of this phase confirms that the architecture of the implementation groups blocks and procedures that can indeed be implemented sequentially.

We use information about the Ada procedures that implement block functionality, namely, the blocks that they implement, and about the procedures handled by each scheduler. For our example, we identify a procedure `Calc_Derivative` that implements the blocks Diff and Sd. Similarly, we also have a procedure `Calc_Integral` that implements the blocks Si and Int. Finally, the main program has procedures `Calc_Proportion`, which implements Sp, and `Calc_Output`, which implements Sum.

We consider each of the procedures 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 group

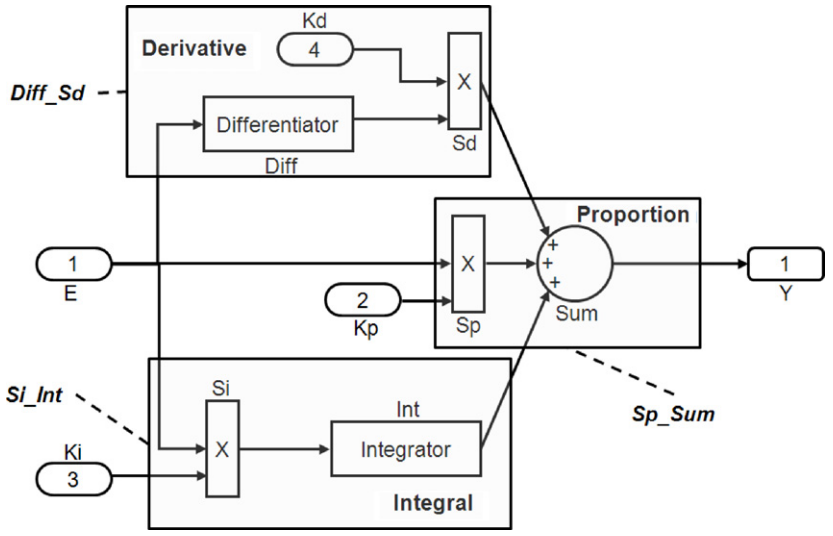

Fig. 6. Structure of the PID to match its implementation.

the processes that correspond to procedures scheduled by a single task. In our case, the procedures `Calc_Proportion` and `Calc_Output`, which implement the blocks Sp and Sum, are scheduled by the same program. Therefore, in this phase, we also join the processes *Sp* and *Sum* to produce a process *Sp_Sum*. In Fig. 6, we illustrate how the original diagram would relate to a hypothetical diagram that corresponds exactly to the model generated by this phase. Nevertheless, the original diagram is not changed; Fig. 6 is given simply to exemplify the *Circus* model transformation.

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

5.2.1. Create a single process

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

The tactic below uses Law 18 (join-proc-par) to join the processes. This law is basically the definition of process parallelism [36]. It describes $P_1 \llbracket cs \rrbracket P_2$ as a basic process whose state includes all the components of P_1 and P_2 and whose main action is the parallel composition of the main actions A_1 of P_1 and A_2 of P_2. If there are clashes in the names of the state components (or any other definitions) of P_1 and P_2, they are resolved by renaming. The name sets associated to A_1 and A_2 in the parallelism are the state components of P_1 and P_2. The program is changed to refer to the new process, instead of the parallel composition, and the individual processes are removed (as they are no longer used).

The processes are joined two at a time. For this reason, we use an auxiliary function *join_all* that takes as argument a list of lists of the names of the processes that must be joined.

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

If the size of the list is less than two, *join_all* ignores it, otherwise *join_all* creates a list of arguments for Law 18 (join-proc-par). This function is useful when we have procedures implementing more than two blocks. For instance, if we apply *join_all* to $\langle\langle P_1, P_2, P_3\rangle\rangle$, we get $\langle(P_1, P_2), (P_1\_P_2, P_3)\rangle$. So, we first join P_1 and P_2 creating a new process $P_1\_P_2$, and then we join $P_1\_P_2$ and P_3. In our example, we get $\langle(Diff, Sd), (Int, Si), (Sp, Sum)\rangle$ applying *join_all* to *pid_blocks*, which is used as argument for Law 18.

A simple invocation of createSingleProcesses formalises the first step of the BJ phase.

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

In our example, with the application of this tactic, using the list *pid_blocks* as argument, we get the *Circus* program sketched in Fig. 7. The *Ini* and *Outj* variables in the state are renamed when the processes are joined to avoid clashes as explained above. The parallelism requires synchronisation on the intersection of the alphabets of the original processes: in our example, the channels *Diff_out* and *end_cycle*. The parallel actions have write access to the state components of the corresponding original processes.

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

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

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

...

$$
\bullet \quad \boxed{
\begin{array}{l}
\left(
\begin{array}{l}
pid\_Diff\_Init\,; \\
\mu X \bullet \left(
\begin{array}{l}
\left(
\begin{array}{l}
E?x \longrightarrow \\
\left(
\begin{array}{l}
(pid\_Diff\_In1 := x\ ;\ Calc\_Diff\_Out)\,; \\
Calc\_Diff\_State
\end{array}
\right)
\end{array}
\right)\,; \\
(Diff\_out!pid\_Diff\_Out1 \longrightarrow \textbf{\textit{Skip}})
\end{array}
\right)\,; \\
end\_cycle \longrightarrow X
\end{array}
\right) \\
[\![\ \{pid\_Diff\_UnitDelay\_St, pid\_Diff\_In1, pid\_Diff\_Out1\} \\
\quad | \ \{\!|\, Diff\_out, end\_cycle \,|\!\} \ | \\
\quad \{pid\_Sd\_In1, pid\_Sd\_In2, pid\_Sd\_Out1\}\ ]\!] \\
\left(
\begin{array}{l}
\mu X \bullet \left(
\begin{array}{l}
\left(
\begin{array}{l}
Diff\_out?x \longrightarrow pid\_Sd\_In2 := x \\
[\![\, \{ pid\_Sd\_In2 \} \mid \{ pid\_Sd\_In1 \} \,]\!] \\
Kd?x \longrightarrow pid\_Sd\_In1 := x
\end{array}
\right)\,; \\
(Calc\_pid\_Sd\ ;\ Sd\_out!pid\_Sd\_Out1 \longrightarrow \textbf{\textit{Skip}})
\end{array}
\right)\,; \\
end\_cycle \longrightarrow X
\end{array}
\right)
\end{array}
}
$$

end

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

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

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

Fig. 7. Result of applying BJStep1

5.2.2. Extract initialisations

At this stage, the main actions of the processes that have been joined have the following structure.

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

I_i are the optional state initialisations, and R_i are the recursive actions that synchronise on *end_cycle*.

The tactic isolateSeqActions from Section 5.1.3 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 or may not have a state initialisation, before invoking this tactic, we must guarantee that there will be a sequential composition. For that, we use the tactic insertSeqComp on both sides of the parallel composition before invoking isolateSeqActions. Finally, we remove any sequential composition with ***Skip*** that might have been included earlier.

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

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

Tactic BJSt2Proc(*pname*) $\mathrel{\widehat=}$
program $\langle (pname, \boxed{\mathrel{\widehat=}} (\boxed{\textbf{beginend}} (\langle\rangle, \mathsf{foldr}_{\parallel}\ (\textbf{tactic}\ \mathsf{isolateInitBin}())))))\rangle$
end

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

Tactic BJSt2(*blocks*) $\mathrel{\widehat=}$ APPLYT(BJSt2Proc, *join_names*(*blocks*)) **end**

This tactic takes the list that contains the list of processes that are to be joined (for instance, *pid_blocks*) as argument. It uses the function *join_names*, which identifies the names of the final processes that resulted from joining each of the lists of

processes. The resulting list is used as argument for the application of tactic BJSt2Proc. In our example, the tactic BJSt2Proc is applied three times: one for each of the processes *Diff_Sd*, *Int_Si*, and *Sp_Sum*. For the first one, we have the resulting main action presented below.

$$
\begin{array}{l}
pid\_Diff\_Init ; \\
\left(\begin{array}{l}
\left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l}
\left(\begin{array}{l} E?x \longrightarrow \\ \left(\begin{array}{l} (pid\_Diff\_In1 := x \ ; \ Calc\_Diff\_Out) ; \\ Calc\_Diff\_State \end{array}\right) \end{array}\right) ; \\
(Diff\_out!pid\_Diff\_Out\,1 \longrightarrow \textbf{\textit{Skip}})
\end{array}\right) ; \ end\_cycle \longrightarrow X
\end{array}\right) \\
[\![\ \{pid\_Diff\_UnditDelay\_St, pid\_Diff\_In1, pid\_Diff\_Out\,1\} \\
\quad | \ \{\!| Diff\_out, end\_cycle |\!\} \ | \\
\quad \{pid\_Sd\_In1, pid\_Sd\_In2, pid\_Sd\_Out\,1\} \ ]\!] \\
\left(\begin{array}{l}
\mu X \bullet \left(\begin{array}{l}
\left(\begin{array}{l} Diff\_out?x \longrightarrow pid\_Sd\_In2 := x \\ [\![\{ pid\_Sd\_In2 \} \mid \{ pid\_Sd\_In1 \}]\!] \\ Kd?x \longrightarrow pid\_Sd\_In1 := x \end{array}\right) ; \\
(Calc\_pid\_Sd \ ; \ Sd\_out!pid\_Sd\_Out\,1 \longrightarrow \textbf{\textit{Skip}})
\end{array}\right) ; \ end\_cycle \longrightarrow X
\end{array}\right)
\end{array}\right)
\end{array}
$$

We are left with the initialisation of the state components of the original *Diff* process, followed by a parallel composition of two recursive actions, which we transform into a single recursive action in the next steps.

5.2.3. Extract the synchronisation on end_cycle

In this step of the BJ phase of the refinement strategy, we extract the synchronisation on *end_cycle*. For that, we use the fixed-point Law 30 (rec-sync) in the tactic isolateEC. As previously described, the main actions of all processes may be sequential compositions of a state initialisation followed by a recursive behaviour. However, the state initialisation is not always present. For uniformity, as we did in the tactics NB and isolateInitBin, we include a sequential composition, if needed, using insertSeqComp.

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

For the same reason as in the previous step, we may have a nested parallel composition (associated to the left). Hence, our strategy is again to remove the synchronisations on *end_cycle* from the parallel composition of all the parallel branches by recursively using Law 30. For this reason, we use the tactic foldr$_{\|}$. For a given process name, the tactic BJSt3Proc extracts the synchronisation on *end_cycle*. It works on the whole *Circus* program but only changes, using isolateEC, the main action of the specified process.

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

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

Tactic BjSt3(*blocks*) $\widehat{=}$ APPLYT(BJSt3Proc, *join_names*(*blocks*)) **end**

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

$$
\begin{array}{l}
\mu X \bullet \left(\begin{array}{l}
\boxed{\begin{array}{l}
\left(\begin{array}{l} (E?x \longrightarrow ((pid\_Diff\_In1 := x \ ; \ Calc\_Diff\_Out) \ ; \ Calc\_Diff\_State)) ; \\ (Diff\_out!pid\_Diff\_Out\,1 \longrightarrow \textbf{\textit{Skip}}) \end{array}\right) \\
[\![\ \{pid\_Diff\_UnditDelay\_St, pid\_Diff\_In1, pid\_Diff\_Out\,1\} \\
\quad | \ \{\!| Diff\_out |\!\} \ | \ \{pid\_Sd\_In1, pid\_Sd\_In2, pid\_Sd\_Out\,1\}]\!] \\
\left(\begin{array}{l}
\left(\begin{array}{l} Diff\_out?x \longrightarrow pid\_Sd\_In2 := x \\ [\![\{ pid\_Sd\_In2 \} \mid \{ pid\_Sd\_In1 \}]\!] \\ Kd?x \longrightarrow pid\_Sd\_In1 := x \end{array}\right) ; \\
(Calc\_pid\_Sd \ ; \ Sd\_out!pid\_Sd\_Out\,1 \longrightarrow \textbf{\textit{Skip}})
\end{array}\right)
\end{array}}
\end{array}\right) ; \\
end\_cycle \longrightarrow X
\end{array}
$$

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

In the next step, we remove the remaining parallel composition.

5.2.4. Remove parallelism

The particular steps required to remove a parallelism depend on the way in which the parallel blocks are arranged. Also, removing 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 in Fig. 5.

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

The first one that succeeds defines the behaviour of the tactic.

The configurations presented in Fig. 5 on Page 802 cover all possible 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. We describe below the tactic that formalises this step for our example, in which the blocks are connected as in Configuration (2). The formalisation of the remaining configurations is in [33].

This step of the BJ phase has four stages: evaluate the synchronisation entailed by the internal communications; remove internal communications; sequentialise assignments; and introduce interleaving of inputs. We now describe each one of them, and then define the tactic BJSt4 that formalises the whole step.

5.2.5. Evaluate the synchronisation entailed by the internal communications

In this stage, we use highly specialised, but similar, refinement laws. For Configuration (2), we have the following parallelism in the body of the recursion that we are refining.

$$(TakeInAndCalc \fatsemi OutInternalComm) \parallel ((InpInternalComm \mid\mid\mid InpExternal) \fatsemi CalcOutAndComm)$$

The first parallel action takes the inputs and calculates the new state and outputs (*TakeInAndCalc*), and then communicates the outputs (*OutInternalComm*). The second action takes the inputs communicated from the first (*InpInternalComm*) and the external inputs (*InpExternal*) in interleaving. Afterwards, it calculates the state and outputs, and communicates the outputs (*CalcOutAndComm*).

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, 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 11 (inter-index) that is based on the definition of indexed interleaving and transforms an explicit interleaving into an indexed one. Simple prefixed actions like in our example are transformed into an indexed interleaving in which the range of the indexing variable has cardinality one. We apply Law 11 (inter-index) to *OutInternalComm* and *InpInternalComm*. We are then able to use a generalisation of the Law 23 (par-out-inp-inter-exchange) from [34], 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 24 (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 interleavings using Law 11 (inter-index), evaluates the synchronisation entailed by the internal communications for Configuration (2) using Law 24 (par-out-inp-inter-exchange-n), and uses Law 11 (inter-index) once again to expand the indexed interleaving that resulted from the evaluation.

Tactic BJSt4A_config2() $\widehat{=}$
  ((**skip** $[\fatsemi]$ **law** inter-index^b())$[\parallel]$((**law** inter-index^b()$[\mid\mid\mid]$**skip**) $[\fatsemi]$**skip**)) ;
  law par-out-inp-inter-exchange-n() ;
  ((**skip** $[\fatsemi]$ (**law** inter-index()$[\mid\mid\mid]$**skip**)) $[\fatsemi]$**skip**)
end

In our example, the result of applying BJSt4A_config2 is presented below.

$$\left(\begin{array}{l}\left(\begin{array}{l}(E?x \longrightarrow ((pid\_Diff\_In1 := x \fatsemi Calc\_Diff\_Out) \fatsemi Calc\_Diff\_State)) \fatsemi \\ (Diff\_out!pid\_Diff\_Out1 \longrightarrow pid\_Sd\_In2 := pid\_Diff\_Out1)\end{array}\right) \\ |[\{pid\_Diff\_UnitDelay\_St, pid\_Diff\_In1, pid\_Diff\_Out1, pid\_Sd\_In2\} \mid \{pid\_Sd\_In1\}]| \\ (Kd?x \longrightarrow pid\_Sd\_In1 := x)\end{array}\right) ; \\ (Calc\_pid\_Sd \fatsemi Sd\_out!pid\_Sd\_Out1 \longrightarrow \textbf{\textit{Skip}})$$

As before, we have a tactic that formalises this stage of the refinement step 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.

Tactic BJSt4AProc(*pname*) $\widehat{=}$
  program $\langle$ (*pname*, $\boxed{\widehat{=}}$ $\boxed{\textbf{beginend}}$ ($\langle\rangle$, **skip** $\boxed{;}$ $\boxed{\mu}$ (**tactic** BJSt4A_all_config() $\boxed{;}$ **skip**))) $\rangle$
end

The tactic that formalises the application of this first refinement stage to all processes is presented below. It follows the same pattern as the tactics previously presented.

Tactic BjSt4A(*blocks*) $\widehat{=}$ APPLYT(BJSt4AProc, *join_names*(*blocks*)) **end**

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

5.2.6. Remove internal communications

Before further refining each individual process, we rewrite the overall diagram model, in our example, the *PID* process. Our aim is to distribute the external hiding over to the processes that describe the internal communications. The tactic BjSt4_HidPrep presented below applies to the whole *Circus* program. It takes the name *main* of the process that models the diagram and works on the body of its definition.

First, it uses Law 20 (par-hid-dist), which applies to processes of the form $((P_1\,\alpha_1) \parallel \ldots \parallel (P_n\,\alpha_n)) \setminus cs$ and distributes the hiding over the parallelism. 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 $\text{mapr}_{\parallel}$, BjSt4_HidPrep applies Law 3 (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 $\text{mapr}_{\parallel}$ below applies a tactic to all elements in a nested right-associated parallel composition.

Tactic $\text{mapr}_{\parallel}(t) \mathrel{\widehat{=}} \mu_T\, X \bullet (t \boxed{\parallel} X) \mid t$ **end**

If it finds a parallel composition, it applies *t* to the left branch and recurs its application to the right branch. At the last branch it will apply only *t*, since the structural combinator for parallelism fails.

Finally, the tactic BjSt4_HidPrep applies the Law 17 (join-proc-hid) to the overall *Circus* program. This law states that if a process *P* is only referenced in the overall program by hiding some of its events $((P\,\alpha_P) \setminus cs)$, then we may replace *P* by a new process P_1. The new process P_1 is very similar to *P*, but hide these events in its main action. In the overall *Circus* specification, we may now reference $(P_1\,(\alpha_P \setminus cs))$; the new interface removes *cs* from the original one.

Tactic BjSt4_HidPrep(*main*) $\widehat{=}$
  program $\langle$ (*main*, $\boxed{\widehat{=}}$ (**law** par-hid-dist() ; $\text{mapr}_{\parallel}$ (**law** hid-contract()))) $\rangle$;
  APPLYL(join-proc-hid, *join_names*(*blocks*))
end

In our example, the definition of *PID* is as follows.

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

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

...

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

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

The new definition of *Diff_Sd*, that is, $Diff\_Sd_1$, hides the internal communication between the original processes *Diff* and *Sd* through the channel *Diff_out* in its main action. Now, we further refine the main action of each process resulting from a join, like $Diff\_Sd_1$, as informally described in [6].

For each process, we use the distribution laws of hiding to localise the hiding of the channel used in the internal communication around the prefixing that carries it out. We then apply Law 9 (hid-step) to remove the communication. This distribution is achieved by the following tactic HidDistStep, which exhaustively applies the distribution laws over the *Circus* operators and, when it is no longer possible to distribute the hiding, applies the step Law 9 (hid-step). Finally, it uses

Law 4 (hid-idem) to remove the hiding.

Tactic HidDistStep() $\widehat{=}$

$$\mu_T\, X \bullet \mathsf{TRY}\left(\begin{array}{l}(\textbf{law}\ \text{hid-seq-dist}()\ ;\ (X\ \boxed{;}\ X))\\ |\ (\textbf{law}\ \text{hid-par-dist}()\ ;\ (X\ \boxed{[\![}\ X))\\ |\ (\textbf{law}\ \text{hid-inter-dist}()\ ;\ (X\ \boxed{|||}\ X))\\ |\ (\textbf{law}\ \text{hid-rec-dist}()\ ;\ (\boxed{\mu}\ X))\\ |\ (\textbf{law}\ \text{hid-step}()\ ;\ \textbf{law}\ \text{hid-idem}())\end{array}\right)$$

end

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

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

The tactic that refines all processes that result from joining two process in parallel (*Diff_Sd*$_1$, *Int_Si*$_1$, and *Sp_Sum*$_1$) uses the tactic BJSt4Proc and the names of the blocks that have been joined.

Tactic BjSt4B(*blocks*) $\widehat{=}$ APPLYT(BJSt4BProc, *join_names*$_1$(*blocks*)) **end**

The function *join_names*$_1$ is very similar to *join_names*, but suffixes all names with $_1$.

In our example, the application of the tactic BJSt4B removes the communication through *Diff_out* in the main action of process *Diff_Sd*$_1$ yielding the main action presented below.

$$\begin{array}{l} pid\_Diff\_Init\ ;\\ \mu X \bullet \left(\begin{array}{l}\boxed{\left(\begin{array}{l}\left(\begin{array}{l}(E?x \longrightarrow ((pid\_Diff\_In1 := x\ ;\ Calc\_Diff\_Out)\ ;\ Calc\_Diff\_State))\ ;\\ pid\_Sd\_In2 := pid\_Diff\_Out1\end{array}\right)\\ [\![\ \{pid\_Diff\_UnitDelay\_St, pid\_Diff\_In1, pid\_Diff\_Out1, pid\_Sd\_In2\}\\ \quad |\ \{pid\_Sd\_In1\}]\!]\\ (Kd?x \longrightarrow pid\_Sd\_In1 := x)\end{array}\right)}\ ;\\ (Calc\_pid\_Sd\ ;\ Sd\_out!pid\_Sd\_Out1 \longrightarrow \textbf{\textit{Skip}})\\ end\_cycle \longrightarrow X\end{array}\right)\ ; \end{array}$$

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

5.2.7. Sequentialise assignments

We transform the interleaving into a sequence of assignments using the Law 13 (inter-seq-assign). However, this law only needs to be applied to the interleaving of assignments. For this reason, the tactic SeqAssign presented below, recursively searches for such structures and, once it finds an interleaving, it searches for further interleaved assignments, and then, it tries to apply the Law 13 (inter-seq-assign).

Tactic SeqAssign() $\widehat{=}$

$$\mu_T\, X \bullet \mathsf{TRY}((X\ \boxed{;}\ X)\ |\ (X\ \boxed{[\![}\ X)\ |\ (\boxed{\longrightarrow}\ X)\ |\ (\boxed{\mu}\ X)\ |\ ((X\ \boxed{|||}\ X)\ ;\ \mathsf{TRY}(\textbf{law}\ \text{inter-seq-assign}())))$$

end

The definition of the tactic that implements the application of this step of the refinement strategy has the same structure as the tactics corresponding to the previous steps. First, we define a tactic BJSt4CProc that applies this step to the main action of a specific process, given its name. Finally, we define a tactic BjSt4C that applies this tactic to each process that resulted from joining two process in parallel. In *Diff_Sd*$_1$, there is only one internal communication; the application of BjSt4C leaves the main action unchanged. We are left with the last stage of this step, which introduces an interleaving of inputs.

5.2.8. Introduce interleaving of inputs

As illustrated by our example, at this stage we have an interleaving that may include more than just the inputs; we simplify it using the tactic in what follows. First, we recall the shape of the main action. For blocks connected according to Configuration (2), the main action is of the following form.

$$I\ ;\ \mu X \bullet \left(\begin{array}{l}\boxed{((((in_1?x \longrightarrow ((v_1 := x\ ;\ A_1)\ ;\ A_2))\ ;\ A_3)\ [\![\ ns_1 \mid ns_2\ ]\!]\ (in_2?x \longrightarrow v_2 := x))\ ;\ A_4)}\ ;\\ end\_cycle \longrightarrow X\end{array}\right)$$

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

In the first part, the tactic BjSt4D_config2 works on the first interleaved action.

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

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

$$(in_1?x \longrightarrow (v_1 := x \mathbin{;} (A_1 \mathbin{;} A_2))) \mathbin{;} A_3$$

Then, it applies the associativity Law 27 (prefix-seq-assoc) to isolate the prefixing and the assignment.

$$((in_1?x \longrightarrow v_1 := x) \mathbin{;} (A_1 \mathbin{;} A_2)) \mathbin{;} A_3$$

Next, it uses again the associativity to isolate the prefixing.

$$(in_1?x \longrightarrow v_1 := x) \mathbin{;} ((A_1 \mathbin{;} A_2) \mathbin{;} A_3)$$

Only then, we are able to use the Law 14 (inter-seq-extract-snd) that removes the processing of inputs from the interleaving, leaving just prefixings of assignments to input variables.

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

The next part of the tactic works specifically on the interleaving of inputs. It uses the Laws 15 (inter-unused-name) and 10 (inter-comm) to retain in the name sets only the input variables. In our example, we remove *pid_Diff_UnitDelay_St*, *pid_Diff_Out*1, and *pid_Sd_In*2 from the first name set. The second name set already contains only the right input variable. Finally, we isolate the interleaving of inputs as the left-hand side of a sequential composition by applying, once again, the associativity law for sequences.

Tactic BjSt4D_config2() $\widehat{=}$
applies to $(((in_1?x \longrightarrow ((st_1 := x \mathbin{;} A_1) \mathbin{;} A_2)) \mathbin{;} A_3) \llbracket ns_1 \mid ns_2 \rrbracket (in_2?x \longrightarrow st_2 := x)) \mathbin{;} A_4$
do

$$\left(\left(\left(\left(\left(\left(\left(\left(\longrightarrow\right) \textbf{ law } \text{seq-assoc}^b() \text{ ; } \textbf{law } \text{prefix-seq-assoc}()\right) \mathbin{;} \textbf{skip}\right) \text{ ; } \textbf{law } \text{seq-assoc}^b()\right) \llbracket\rrbracket \textbf{ skip}\right) \text{ ; } \textbf{law } \text{inter-seq-extract-snd}() \text{ ;}\right.\right.\right.$$
$$\left.\left.\left.\left(\left(\textbf{law } \text{inter-unused-name}(\{st_2\}) \text{ ; } \textbf{law } \text{inter-comm}() \right.\right.\right.\right.$$
$$\left.\left.\left.\left.\textbf{law } \text{inter-unused-name}(\{st_1\}) \text{ ; } \textbf{law } \text{inter-comm}()\right) \mathbin{;} \textbf{skip}\right)\right) \mathbin{;} \textbf{skip}\right) \text{ ; } \textbf{law } \text{seq-assoc}()$$

end

For a given process, BJSt4DProc below introduces the interleaving of inputs in its main action. It works on the whole *Circus* program, but only changes, using BjSt4D_config2, the specified process. It works specifically on the first action in the sequential composition within the body of the recursion.

Tactic BJSt4DProc(*pname*) $\widehat{=}$
program $\langle (pname, \widehat{=} (\textbf{beginend}) (\langle\rangle, \textbf{skip} \mathbin{;} \mu (\text{TRY}(\textbf{tactic } \text{BjSt4D-config2}()) \mathbin{;} \textbf{skip})))) \rangle$
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 processes corresponding to groups of blocks connected according to Configuration (2), like in our example. In that context, application of this tactic results in the main action presented below.

$$pid\_Diff\_Init \mathbin{;} \mu X \bullet \left(\begin{array}{l} \left(\begin{array}{l} (E?x \longrightarrow pid\_Diff\_In1 := x) \\ \llbracket \{pid\_Diff\_In1\} \mid \{pid\_Sd\_In1\} \rrbracket \\ (Kd?x \longrightarrow pid\_Sd\_In1 := x) \end{array}\right); \\ \left(\left(\begin{array}{l} (Calc\_Diff\_Out \mathbin{;} Calc\_Diff\_State); \\ (pid\_Sd\_In2 := pid\_Diff\_Out1) \end{array}\right); \right. \\ \left. (Calc\_pid\_Sd \mathbin{;} Sd\_out!pid\_Sd\_Out1 \longrightarrow \textbf{\textit{Skip}}) \right) \end{array}\right) \mathbin{;} end\_cycle \longrightarrow X$$

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 processes that resulted from joining two process in parallel ($Diff\_Sd_1$, Int_Si_1, and Sp_Sum_1) uses the tactic BJSt4DProc and the names of the blocks that have been joined.

Tactic BjSt4D(*blocks*) $\widehat{=}$ APPLYT(BJSt4DProc, $join\_names_1(blocks)$) **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*) $\widehat{=}$
tactic BjSt4A(*blocks*) ; **tactic** BjSt4_HidPrep(*main*) ;
tactic BjSt4B(*blocks*) ; **tactic** BjSt4C(*blocks*) ; **tactic** BjSt4D(*blocks*)
end

This tactic takes the name of the diagram process and the list that contains the lists of processes that are to be joined. It invokes the tactics that formalise each of the refinement stages in sequence.

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

Tactic BJ(*main*, *blocks*) $\widehat{=}$
  tactic BJSt1(*blocks*) ; **tactic** BJSt2(*blocks*) ;
  tactic BJSt3(*blocks*) ; **tactic** BJSt4(*main*, *blocks*)
end

It has the same argument as BJSt4 and also simply invokes the tactics that corresponds to each of the steps.

Using BJ, we obtain the process $Diff\_Sd_1$. We also join the *Circus* processes *Si* and *Int* to produce a process Si_Int_1, and the processes *Sp* and *Sum* to produce a process Sp_Sum_1. Regardless of the difference in the internal structure of these processes, however, the tactic BJ can be applied with success reducing considerably the amount of effort used in the correctness proof of the PID.

The remainder of the refinement strategy can be formalised in the same way; this is left as future work. The formalisation of the refinement strategy for control laws in ArcAngel*C* fosters its automatic application using tools like that described in Section 7.

6. The denotational semantics of ArcAngel*C*

In this section, we describe our formalisation of the semantics of ArcAngel*C*. As opposed to the specification of the ArcAngel semantics, where we use a custom mathematical language, the formalisation of ArcAngel*C* uses Z as a meta-language, and we have a specification of the syntax of *Circus* embedded in Z. This makes possible the mechanisation of the ArcAngel*C* semantics using Z theorem provers.

Semantically, a tactic is applied to a refinement cell: a pair in which the first element is a term, and the second element is the proof obligations accumulated so far. The tactic application to a refinement cell returns a possibly infinite list of refinement cells: each element is a possible outcome of the tactic application. To handle different sorts of tactics (action, process, and program tactics), we define the term to be of type *Cell*: it can be a parametrised action, a parametrised process, a sequence of process paragraphs, or a sequence of program paragraphs.

$$Cell ::= ParActC \langle\langle ParAct \rangle\rangle \mid ParProcC \langle\langle ParProc \rangle\rangle \mid ProcParC \langle\langle \mathrm{seq}\, ProcPar \rangle\rangle \mid ProgC \langle\langle \mathrm{seq}\, ProgPar \rangle\rangle$$

The sets *ParAct*, *ParProc*, *ProcPar*, and *ProgPar* contain (Z representations of) parametrised actions, parametrised processes, process paragraphs, and program paragraphs. We also define the sets that contain each sort of cell. For instance, *ParProcCell* contains all cells corresponding to parametrised processes.

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

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 can also have data refinement laws, for which we need to record a retrieve relation (given as a schema expression, that is an element of *SchemaExp*) and the declaration (that is, an element of *Decl*) of any local variables.

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

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

$$PObs == \mathrm{seq}\, Pred \times \mathrm{seq}\, Refinement$$

The use of sets instead of sequences in the definition of *PObs* would have no impact in the majority of the semantics definitions. We have, however, tacticals that can be used to discharge refinement proof obligations by providing a list of tactics (see Section 6.3). The use of sequences makes it simpler to define such tacticals because we can establish a simple correspondence between the refinement proof obligations in a list $\langle A_1 \sqsubseteq C_1, \ldots, A_n \sqsubseteq C_n \rangle$ and the tactics in a list $\langle t_1, \ldots, t_n \rangle$. It is the extra structure in the model of proof obligations, and the distinction between simple predicates and refinement conjectures, that allows us to use ArcAngel*C* to tackle both program derivations and the proof obligations.

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.

$$\begin{aligned} Law == & (ParActCell \nrightarrow ParActRCell) \\ & \cup (ParProcCell \nrightarrow ParProcRCell) \\ & \cup (ProcParCell \nrightarrow ProcParRCell) \\ & \cup (ProgCell \nrightarrow ProgRCell) \end{aligned}$$

As said before, the result of a tactic application (to a refinement cell) is a possibly infinite list of refinement cells that contains all possible outcomes of its application: every program it can generate, together with the corresponding proof obligations. They include existing proof obligations and those generated by the tactic application. Different possibilities arise from the use of alternation, and the list can be infinite because the application of a tactic may run indefinitely. On the other hand, if the application of the tactic fails, then the empty list is the result. Like for laws, there is a correspondence between the type of refinement cell to which the tactic is applied and that of the cells that result from the application.

$$\begin{aligned} Tactic == & (ParActRCell \to \text{pfiseq}[ParActRCell]) \\ & \cup (ParProcRCell \to \text{pfiseq}[ParProcRCell]) \\ & \cup (ProcParRCell \to \text{pfiseq}[ProcParRCell]) \\ & \cup (ProgRCell \to \text{pfiseq}[ProgRCell]) \end{aligned}$$

The type pfiseq[*RCell*] is that of the possibly infinite lists of *RCell*s. We use the model for infinite lists proposed in [21]; it is summarised in Appendix A. In this model, finite, partial, and infinite lists are considered. A partial list ends in an undefined list, denoted $\bot$. An infinite list is a limit of a directed set of partial lists.

To give a semantics to named laws and tactics, we maintain two environments. The law environment records the known laws; it is a partial function whose domain is the set of their names.

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

For a law environment env_L and a given law name n, we have that $env_L\, n$ is also a partial function that relates all valid arguments of n (sequences of terms) to yet another function, a *Law*, as previously defined. Similarly, a tactic environment takes a tactic name and a sequence of arguments, and returns a *Tactic*.

6.1. Tactics

The basic tactic **law** n(a) applies a simple law to an *RCell*; it is defined as follows.

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

If the law name n is in the given law environment *lenv*, and if the sequence of arguments a and cell $r.1$ are appropriate, that is, if a is in the domain of the law, and the program $r.1$ is in the domain of the law when applied to a, then the tactic succeeds, and returns a list with a new *RCell*. The refinement cell r is transformed by applying the law to the cell $r.1$; the new proof obligations are merged with the proof obligations $r.2$ of the original *RCell*. Otherwise, the tactic fails with the empty list as result. The brackets subscripted with ∞ indicate that we have a possibly infinite list (that is, an element of pfiseq).

The function *MPObs* merges two lists of proof obligations. This is needed because we have two different sorts of proof obligations (predicates and refinements), which form the pair of sequences that represent proof obligations. Hence, merging existing proof obligations with new ones is not a union of proof obligations sets, as in ArcAngel's semantics, but a combination of pairs of proof obligations.

We use a simple approach for expression arguments: we consider that the expressions used as parameters of laws have already been evaluated. This allows us to focus on the semantics of the tactics themselves.

The semantics of **tactic**(n, a) is similar to that of **law** (n, a) presented above. The tactic **skip** produces the singleton list that contains its (refinement cell) argument unchanged.

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

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

The tactic sequence operator uses a construction known as the Kleisli composition [19]. 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, the list of lists thereby obtained is flattened to produce the result.

$$\_\,;\,\_ : (Tactic \times Tactic) \rightarrow Tactic$$

$$\forall t1, t2 : Tactic;\ r : RCell \bullet (t1\ ;\ t2)(r) = \overset{\infty}{\frown}/[RCell](t2 * (t1(r)))$$

For a total function $f : A \rightarrow B$, we have that $f* :$ pfiseq $A \rightarrow$ pfiseq B is the map function that operates on a list by applying f to each of its elements; the operator $\overset{\infty}{\frown}/$ is the distributed concatenation. It is defined as a generic function. Hence, we need to use [*RCell*] to instantiate it for refinement cells. Formal definitions of these operators and others to follow can be found in Appendix A.

The semantics of alternation ($t_1 \mid t_2$), cut ($!t$), recursion ($\mu_T X \bullet t$), **abort**, assertions **succs** t and **fails** t, and pattern matching are similar to those from [35] and are omitted here for conciseness. A full account on the semantics of ArcAngel*C* can be found in [33].

6.2. 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. As an example, we consider the application of tactics t_1 and t_2 to two different components p_1 and p_2 of a program P (we 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 carry out these applications independently. If $t_1\,(p_1) = [r_1, r_2\,]_\infty$ and $t_2\,(p_2) = [s_1, s_2\,]_\infty$, then the combination of these results gives us a list $[(r_1, s_1), (r_1, s_2), (r_2, s_1), (r_2, s_2)\,]_\infty$. The application of the structural combinator yields a list of programs 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.

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

In ArcAngel*C*, as already said, there is one combinator for each *Circus* construct. Since tactics may target different components of a *Circus* program, we have four groups of structural combinators: those that can be applied to either actions or processes, those that can be applied only to actions, those that can be applied only to processes, and those that can be applied to programs.

The reassembling previously mentioned is done by Ω functions. The semantics of the vast majority of the structural combinators of ArcAngel*C* has the following structure.

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

In this template, *structComb* stands for the structural combinator, *tacs* is a single tactic, a pair of tactics, or a sequence of tactics, and *rc* is a refinement cell. The semantics is defined by first applying the tactic(s) to components of the refinement cell's program. The result of their application is represented by *tacApp* above. Next, we map an Ω function to the resulting list. The arguments *args* for the Ω function are static parameters of the program construct that are not affected by the application of tactics.

The naming and typing of the Ω functions are similar. Below, we present a template: essentially, these are generic functions on a type X of *Circus* terms as presented below.

$$\Omega^R_D : X \rightarrow ((X \times D) \rightarrow R) \rightarrow RCell \nrightarrow RCell$$

They receive a term x of type X, a function $f : (X \times D) \rightarrow R$ representing a syntactic constructor, and a refinement cell. D and R are *Circus* syntactic categories (*ActBody*, *ProcPar*, and so on). The result of the application of an Ω function is a refinement cell with the same proof obligations as the cell it takes as its argument. Its cell is of type R; it is the result of the application of f to (x, p), where p is the program structure of the cell in the original refinement cell. For *Circus* binary structural combinators, the argument function has type $f : (X \times D \times D) \rightarrow R$; this is reflected in the name of the function by subscripting the D with a 2 as in $\Omega^R_{D_2}$. We omit the R when it is the same as D.

In what follows we instantiate the above template for the various ArcAngel*C* structural combinators. In each case, we define the appropriate Ω function.

6.2.1. Action combinators

The first Ω function, $\Omega_{ActBody}$, instantiates both D and R to the syntactic category of action bodies, *ActBody*. Its application defines a refinement cell with a basic action as its program, which results from the application of f to (x, a), where a is the action body of the cell in the original refinement cell.

$$[X]$$

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

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

A parametrised action can either have parameters or not. The constructor *BaseAct* is used for those that do not have parameters.

To illustrate the definition, if we have $c : Comm$ (c is a *Circus* communication, and *Comm* the respective type in the syntactic model) we can have the following application of $\Omega_{ActBody}$. We observe that the prefixing construct $\longrightarrow$ is a function $\_ \longrightarrow \_ : (Comm \times ActBody) \to ActBody$.

$$\Omega_{ActBody}\ (c)\ (\longrightarrow)\ (ParActC(BaseAct(\textbf{\textit{Skip}})), pobs) = (ParActC(BaseAct(\longrightarrow(c, \textbf{\textit{Skip}}))), pobs)$$

If we apply $\Omega_{ActBody}$ to a communication c, the function $\longrightarrow$, and a refinement cell with the action ***Skip*** and proof obligations *pobs*, then we get a refinement cell that contains the action $c \longrightarrow$ ***Skip*** as its cell (represented as $ParActC(BaseAct(\longrightarrow(c, \textbf{\textit{Skip}})))$ in our embedding of the syntax) and the proof obligations *pobs*.

Using this Ω function, we define the structural combinator for prefixing. As already said, the combinator $\boxed{\longrightarrow}\ t$ applies to a prefixing $c \longrightarrow A$ (encoded as $\longrightarrow(c, A)$ in our Z embedding of *Circus*). It applies t to A; the possible results are assembled back using the function $\Omega_{ActBody}$ with arguments c and $\longrightarrow$.

$$\boxed{\longrightarrow} : Tactic \to Tactic$$

$$\forall t : Tactic;\ c : Comm;\ a : ActBody;\ pobs : PObs \bullet$$
$$(\boxed{\longrightarrow}\ t)\ (ParActC(BaseAct(\longrightarrow(c, a))), pobs) =$$
$$(\Omega_{ActBody}\ (c)\ (\longrightarrow)) * (t\ (ParActC(BaseAct(a)), pobs))$$

For every action constructor that is defined as a function $f : (X \times ActBody) \to ActBody$, like $\longrightarrow$ above, the semantics of the corresponding combinator is similarly defined: we apply the tactic to the action body and reassemble the original action using the function $\Omega_{ActBody}$. This holds for the combinators for guarded actions ($\boxed{\&}$), recursions ($\boxed{\mu}$), variable blocks ($\boxed{\textbf{\textit{var}}}$), and parametrised actions ($\boxed{\textbf{\textit{val}}}$, $\boxed{\textbf{\textit{res}}}$, and $\boxed{\textbf{\textit{vres}}}$).

The semantics of the combinator for alternation has a more complex definition; it uses a few functions which are explained as needed. To present the definition in a more didactic way, we use an example, but do not present the subtleties of the formalisation of the syntax. We consider the following *RCell*.

$$\left(\left(\begin{array}{l} \textbf{\textit{if}}\ x \geq 0 \longrightarrow x : [\, x \geq 0, x > y \,] \\ [\!]\ x < 0 \longrightarrow x : [\, x < 0, x < z \,] \\ \textbf{\textit{fi}} \end{array} \right), (\langle x \geq 1 \rangle, \langle\,\rangle) \right)$$

and the tactic $\boxed{\textbf{\textit{if}}}$ **law** weak-pre($true$) | **law** assign-intro($x := x + 1$) $\boxed{[\!]}$ **law** assign-intro($x := x - 1$) $\boxed{\textbf{\textit{fi}}}$.

We need to extract the actions from each branch (pair formed by a guard and an action) of the alternation; this is the purpose of the function *extractActions*. The function *buildRCell* constructs an *RCell* with the given action body and proof obligations. To build the cells to which the tactics are to be applied, we use a function *buildRCells* to map *buildRCell* over the extracted list of actions, using the original proof obligations as an argument. Its definition can be found in Appendix C. In our example we have the following result.

$$\begin{array}{l} buildRCells(\langle (x \geq 0, x : [\, x \geq 0, x > y \,]), (x < 0, x : [\, x < 0, x < z \,]) \rangle, (\langle x \geq 1 \rangle, \langle\,\rangle)) \\ = (buildRCell\ (\langle x \geq 1 \rangle, \langle\,\rangle))\ \mathrm{map} \\ \quad (extractActions\ \langle (x \geq 0, x : [\, x \geq 0, x > y \,]), (x < 0, x : [\, x < 0, x < z \,]) \rangle) \\ = (buildRCell(\langle x \geq 1 \rangle, \langle\,\rangle))\ \mathrm{map}\ (\langle x : [\, x \geq 0, x > y \,], x : [\, x < 0, x < z \,] \rangle) \\ = \langle (x : [\, x \geq 0, x > y \,], (\langle x \geq 1 \rangle, \langle\,\rangle)), (x : [\, x < 0, x < z \,], (\langle x \geq 1 \rangle, \langle\,\rangle)) \rangle \end{array}$$

Now, we need to apply each tactic to its corresponding action. We use the function *applyTacsGC*, whose definition can also be found in Appendix C. It takes two lists: the first is a list of tactics and the second is a list of refinement cells. It applies each tactic in the first list to the corresponding refinement cell in the second list. This results in a list of lists of refinement cells. We then use the distributed cartesian product (Π_∞) to get all the possible combinations as sequences of refinement cells in each of the lists. This yields a flat list of sequences in which each element represents a particular outcome of applying the tactics to the programs of the initial refinement cell. In our example, the result is sketched below.

$$\begin{array}{l} \Pi_\infty \left(\begin{array}{l} \langle (\textbf{law}\ \text{weak-pre}(true)\ |\ \textbf{law}\ \text{assign-intro}(x := x + 1)), (\textbf{law}\ \text{assign-intro}(x := x - 1)) \rangle \\ \text{mapl} \\ \langle (x : [\, x \geq 0, x > y \,], (\langle x \geq 1 \rangle, \langle\,\rangle)), (x : [\, x < 0, x < z \,], (\langle x \geq 1 \rangle, \langle\,\rangle)) \rangle \end{array} \right) \\ = \Pi_\infty \left(\left\langle \begin{array}{l} (\textbf{law}\ \text{weak-pre}(true)\ |\ \textbf{law}\ \text{assign-intro}(x := x + 1))\ (x : [\, x \geq 0, x > y \,], (\langle x \geq 1 \rangle, \langle\,\rangle)), \\ (\textbf{law}\ \text{assign-intro}(x := x - 1))\ (x : [\, x < 0, x < z \,], (\langle x \geq 1 \rangle, \langle\,\rangle)) \end{array} \right\rangle \right) \\ = \Pi_\infty \left(\left\langle \begin{array}{l} [\ (x : [\, true, x > y \,], (\langle x \geq 1, x \geq 0 \Rightarrow true \rangle, \langle\,\rangle)), (x := x + 1, \ldots)\ ]_\infty, \\ [(x := x - 1, (\langle x \geq 1, x < 0 \Rightarrow x - 1 < z \rangle, \langle\,\rangle))\ ]_\infty \end{array} \right\rangle \right) \\ = \left[\left\langle \begin{array}{l} (x : [\, true, x > y \,], (\langle x \geq 1, x \geq 0 \Rightarrow true \rangle, \langle\,\rangle)), \\ (x := x - 1, (\langle x \geq 1, x < 0 \Rightarrow x - 1 < z \rangle, \langle\,\rangle)) \end{array} \right\rangle, \langle (x := x + 1, \ldots), (x := x - 1, \ldots) \rangle \right]_\infty \end{array}$$

The next step is to rebuild the alternation. First, we need the list of its guards; for that, we use the function *extractGuards*. We combine this list with each element of the list of *RCell*s we get from the distributed cartesian product above. The

function *insertGuards* takes a list of guards g_i, a list of *RCells* $(a_i, pobs_i)$, and yields a pair with a guarded command and proof obligations. The guarded command associates each guard g_i to the action a_i. The set of proof obligations is the concatenation of the $pobs_i$.

What we need is to apply the function *insertGuards* to each element of the list resulting from the application of *applyTacsGC*. For our example, the list is sketched above. To exemplify this, we present below the application of *insertGuards* to the first element of the list.

$$\begin{array}{l} insertGuards\ \langle x \geq 0, x < 0\rangle \left\langle \begin{array}{l} (x : [\, true, x > y\,], (\langle x \geq 1, x \geq 0 \Rightarrow true\rangle, \langle\,\rangle)), \\ (x := x - 1, (\langle x \geq 1, x < 0 \Rightarrow x - 1 < z\rangle, \langle\,\rangle)) \end{array} \right\rangle \\ = \quad ((\langle x \geq 0, x < 0\rangle, \langle x : [\, true, x > y\,], x := x - 1\rangle), (\langle x \geq 1, x \geq 0 \Rightarrow true, x \geq 1, x < 0 \Rightarrow x - 1 < z\rangle, \langle\,\rangle)) \end{array}$$

The function $\Omega_{ActBody_n}$ uses the function *insertGuards* to rebuild the refinement cell. It takes the sequence of predicates corresponding to the guards, a function from guarded commands to action bodies like ***iffi*** (the Z representation for alternation), a sequence of refinement cells, and defines the resulting refinement cell.

$$\begin{array}{|l} \Omega_{ActBody_n} : (\mathrm{seq}\ Pred) \to (GuardedCommands \to ActBody) \to (\mathrm{seq}\ RCell) \nrightarrow RCell \\ \hline \forall\, guards : \mathrm{seq}\ Pred;\ f : GuardedCommands \to ActBody;\ rcs : \mathrm{seq}\ RCell \bullet \\ \quad \exists\, gcs\_pobs : GuardedCommands \times PObs \mid gcs\_pobs = insertGuards\, guards\, rcs \bullet \\ \quad\quad \Omega_{ActBody_n}\, guards\, f\, rcs = (ParActC(BaseAct(f\ gcs\_pobs.1)), gcs\_pobs.2) \end{array}$$

The pattern used to define $\Omega_{ActBody_n}$ is a generalisation of that previously suggested. The first parameter is a sequence of terms of a type X, namely *Pred*, rather than a single term. Correspondingly, the second parameter, that is, the syntactic constructor embeds such a list (in *GuardedCommands* in the case of our example), and the third parameter is a list of refinement cells. This indicates how the uniformity previously identified is exploited in the definition of the semantics of all ArcAngel*C* combinators. The approach can also be valuable in the definition of refinement-tactic combinators for other languages besides *Circus*.

By instantiating the function parameter of $\Omega_{ActBody_n}$ with ***iffi***, we have the semantics of the structural combinator $\boxed{\textbf{\textit{if}}}\_\boxed{\textbf{\textit{fi}}}$. With the use of ***iffi***, the refinement cells contains an alternation as its program.

$$\begin{array}{|l} \boxed{\textbf{\textit{if}}}\_\boxed{\textbf{\textit{fi}}} : \mathrm{seq}_1\ Tactic \to Tactic \\ \hline \forall\, tacs : \mathrm{seq}_1\ Tactic;\ gcs : GuardedCommands;\ pobs : PObs \bullet \\ \quad (\boxed{\textbf{\textit{if}}}\, tacs\, \boxed{\textbf{\textit{fi}}})\ ((ParActC(BaseAct(\textbf{\textit{iffi}}(gcs))), pobs)) = \\ \quad\quad (\Omega_{ActBody_n}\ (extractGuards\, gcs)\ (\textbf{\textit{iffi}})) * (applyTacsGC(tacs, buildRCells(gcs, pobs))) \end{array}$$

The semantics of the structural combinator maps the application of $\Omega_{ActBody_n}$, with the list of guards extracted using *extractGuards* and the function ***iffi*** as arguments, to the list resulting from the application of the tactics to the corresponding guarded actions wrapped in fresh refinement cells. This follows the same pattern of definition used for the specification of the semantics of all structural combinators.

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

6.2.2. Process combinators

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

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

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

The structural combinator $\boxed{\textbf{beginend}}$ applies to refinement cells that have an explicit process definition as their first element. It applies each of the argument tactics to the corresponding part of the declaration, merges the process paragraphs, and rebuilds the refinement cells. Its definition is omitted here, but like all the other definitions that are not discussed in details, can be found in [33].

6.2.3. Action and process combinators

The structure of the definitions of combinators that can be applied to either actions or processes uses the same pattern we have used for combinators for actions and processes. Since they are used for both actions and processes, however, their definition is a conjunction. Each of the conjuncts has the same structure as the simpler definitions previously presented, but

defines the behaviour for a particular type of argument (using a rebuilding action Ω_A or rebuilding process Ω_P function as appropriate).

$$(structComb_P\ tacs)\ rc = (\Omega_P\ args) * tacApp$$
$$\wedge$$
$$(structComb_A\ tacs)\ rc = (\Omega_A\ args) * tacApp$$

The first definition of a structural combinator that can be applied to processes and actions that we present is that of the structural combinator $\boxed{\widehat{=}}$, which allows the application of a tactic to the body of a process or action definition. It uses two Ω functions: for parametrised actions, $\Omega^{ProcPar}_{ParAct}$, which assembles a process paragraph, and for parametrised processes, $\Omega^{ProgPar}_{ParProc}$, which assembles a program paragraph.

As previously discussed, the structural combinator $\boxed{\widehat{=}}$ can be applied to a parametrised process definition $\textbf{\textit{process}}\ n[gen] \mathrel{\widehat{=}} p$, which is denoted by $\textbf{\textit{process}}((n, gen), p)$ in our embedding of the *Circus* syntax, where $[gen]$ are optional type arguments for generic processes. In this case, it applies the tactic to the process body and reconstructs the process definition, which is a program paragraph. On the other hand, if the combinator is applied to an action definition $n \mathrel{\widehat{=}} a$, which is denoted by $ActDef\ (n, a)$ in our syntactic embedding, it applies the tactic to the action body and reconstructs the action definition, which is a process paragraph.

$$\boxed{\widehat{=}} : Tactic \to Tactic$$

$$\forall t : Tactic;\ n : N;\ gen : \operatorname{seq} N;\ a : ParAct;\ p : ParProc;\ pobs : PObs \bullet$$
$$(\boxed{\widehat{=}}\, t)(ProgC(\langle \textbf{\textit{process}}((n, gen), p)\rangle), pobs) =$$
$$(\Omega^{ProgPar}_{ParProc}\ (n, gen)\ (\textbf{\textit{process}})) * (t\ (ParProcC(p), pobs))$$
$$\wedge\ (\boxed{\widehat{=}}\, t)(ProcParC(\langle ActDef\ (n, a)\rangle), pobs) =$$
$$(\Omega^{ProcPar}_{ParAct}\ n\ ActDef) * (t\ (ParActC(a), pobs))$$

A similar approach is adopted for the combinators for the *Circus* binary operators of sequential composition, external choice, internal choice, parallel composition, and interleaving. The domains of these functions, however, are slightly different: the combinators for sequence, and for external and internal choice are defined as functions of pairs of tactics, and the remaining combinators are defined as functions of triples of tactics.

In the definition of the semantics of all the binary combinators, we use functions $applyTacs_{ProcBody_2}$ and $applyTacs_{ActBody_2}$ that apply each of their two argument tactics to each of the action or process bodies also received as arguments. As already explained, they combine both lists of results, using the distributed cartesian product, and rebuild the refinement cells using the corresponding Ω functions.

To establish a pattern for the definitions of structural combinators that apply to constructs like sequential composition, and of combinators that apply to constructs like parallel composition, the functions that define their semantics accept functions that represent *Circus* constructs that have triples in their domain. Constructs like sequential composition have only two arguments, which are the two actions or processes that are composed in sequence. On the other hand, for constructs like parallel composition, we have one further argument: a triple (ns_1, cs, ns_2) that contains the state partitions ns_1 and ns_2, and the synchronisation channel set cs, in the case of actions, and just a synchronisation set in the case of processes. To reconcile these differences, we use a function *generalise* that transforms functions $((X \times X) \to X)$, which take pairs as arguments, into functions of type $((NIL\_VAL \times X \times X) \to X)$, which take triples as arguments, but ignore the first argument, which is always the value *nil* (of type *NIL_VAL*).

To demonstrate the above, we present the definition of the structural combinator for sequential composition. We generalise the functions $;_A$ and $;_P$ (embedding of the sequential composition for actions and processes, respectively) before using them as arguments to $applyTacs_{ProcBody_2}$ and $applyTacs_{ActBody_2}$.

$$\_\,\boxed{;}\,\_ : (Tactic \times Tactic) \to Tactic$$

$$\forall t1, t2 : Tactic;\ a1, a2 : ActBody;\ p1, p2 : ProcBody;\ pobs : PObs \bullet$$
$$(t1\,\boxed{;}\,t2)(ParProcC(BaseProc(;_P(p1, p2))), pobs) =$$
$$applyTacs_{ProcBody_2}\ (nil, generalise\ (;_P), (p1, p2), pobs, (t1, t2))$$
$$\wedge\ (t1\,\boxed{;}\,t2)(ParActC(BaseAct(;_A(a1, a2))), pobs) =$$
$$applyTacs_{ActBody_2}\ (nil, generalise\ (;_A), (a1, a2), pobs, (t1, t2))$$

The same holds for the definitions of the structural combinators that apply to external choice and internal choice. For parallel composition of processes and actions (denoted by $\|_P$ and $\|_A$, respectively) we have that the first element of the triples are indeed used and hence, we do not need to generalise the functions $\|_P$ and $\|_A$. We use the functions $applyTacs_{ProcBody_2}$ and $applyTacs_{ActBody_2}$ with the appropriate components of the action or process as first argument, and with the constructor functions themselves as second argument.

$$\_[\![\|]\!]\_ : (Tactic \times Tactic) \to Tactic$$

$$\begin{aligned}
&\forall t1, t2 : Tactic;\ a1, a2 : ActBody;\ p1, p2 : ProcBody;\\
&\quad ns1, ns2 : NSExp;\ cs : CSExp;\ pobs : PObs \bullet\\
&\quad (t1 [\![\|]\!] t2)(ParProcC(BaseProc(\|_P\ (cs, p1, p2))), pobs) =\\
&\qquad applyTacs_{ProcBody_2}\ (cs, (\|_P), (p1, p2), pobs, (t1, t2))\\
&\quad \land\ (t1 [\![\|]\!] t2)(ParActC(BaseAct(\|_A\ ((ns1, cs, ns2), a1, a2))), pobs) =\\
&\qquad applyTacs_{ActBody_2}\ ((ns1, cs, ns2), (\|_A), (a1, a2), pobs, (t1, t2))
\end{aligned}$$

The interleaving structural combinator is handled in a similar way.

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

6.3. Tacticals

Some tactic languages include definitions of meta-tactics, also known as tacticals. An example of a tactical is EXHAUST presented in Section 3; it exhaustively applies a given tactic. Another well-known tactical makes a robust application of a tactic t. This is called *TRY* and is defined as follows.

$$TRY : Tactic \to Tactic$$

$$\forall t : Tactic \bullet TRY\ t = !(t \mid \mathbf{skip})$$

A related tactical receives a list of tactics and tries to apply each of them in order. The first successful tactic application terminates this tactical; if none of the tactics succeeds, the whole tactic skips.

$$TRY\_FIRST : \mathrm{seq}\ Tactic \to Tactic$$

$$\forall ts : \mathrm{seq}\ Tactic \bullet TRY\_FIRST\ ts = \mathbf{if}\ \#ts = 0\ \mathbf{then\ skip\ else}\ (head\ ts) \mid TRY\_FIRST\ (tail\ ts)$$

For *Circus*, we define more specific tacticals, which can be used in the application of the *Circus* refinement calculus. By way of illustration, we present a tactical *JUSTIFIED_BY*, which can be used to automatically apply a given tactic and discharge the generated proof obligations using a given list of tactics. Many tacticals, like *TRY*, are defined in terms of existing tactic constructs. Others, like *TRY_FIRST* and *JUSTIFIED_BY*, are defined in terms of the language model; these are genuine extensions to the tactic language.

Before presenting *JUSTIFIED_BY*, we introduce auxiliary functions that are used in its definition. The first one tries to use a tactic to justify a refinement $A_1 \sqsubseteq_{\mathcal{A}} A_2$. It applies the given tactic to a refinement cell containing A_1 in its cell and no proof obligations. If the resulting list of refinement cells contains an element whose cell is A_2, the result is the proof obligations of the refinement cell with A_2. Otherwise, the application of the tactic has not justified the refinement and the result is the proof obligation $A_1 \sqsubseteq_{\mathcal{A}} A_2$.

$$\_justify\_ : (Tactic \times Refinement) \nrightarrow PObs$$

$$\begin{aligned}
&\forall t : Tactic;\ ref : Refinement \bullet\\
&\quad \exists npobs : PObs \bullet\\
&\qquad t\ justify\ ref = \ \mathbf{let}\ A_1 == (getActions\ ref).1;\ A_2 == (getActions\ ref).2 \bullet\\
&\qquad\qquad \mathbf{if}\ (buildRCell\ npobs\ A_2)\ \mathsf{in}_\infty\ (t\ (buildRCell\ (\langle\rangle, \langle\rangle)\ A_1))\\
&\qquad\qquad \mathbf{then}\ npobs\\
&\qquad\qquad \mathbf{else}\ (\{\}, \langle ref \rangle)
\end{aligned}$$

For a refinement $A_1 \sqsubseteq_A A_2$, the function *getActions* yields (A_1, A_2); finally, $x\ \mathsf{in}_\infty\ xs$ verifies if x is a member of the sequence xs.

The function *justifyl* receives a pair of equally sized sequences of tactics and refinements. It combines each tactic with the corresponding refinement in a pair, and applies the justification function *justify* to each of these pairs. It returns the merge of all the proof obligations generated by these justifications.

$$\_justifyl\_ : ((\mathrm{seq}\ Tactic) \times (\mathrm{seq}\ Refinement)) \to PObs$$

$$\begin{aligned}
&\forall ts : \mathrm{seq}\ Tactic;\ refs : \mathrm{seq}\ Refinement \mid \#ts = \#refs \bullet\\
&\quad ts\ justifyl\ refs = MPObsSeq\ ((_justify_)\ \mathsf{map}\ (ts\ \mathsf{pairwise}\ refs))
\end{aligned}$$

MPObsSeq merges a list of proof obligations. The function pairwise takes a pair of sequences (xs, ys) of the same length, and yields a sequence of pairs (x, y) where x is in xs and y is the corresponding element in ys.

The next function, *justifycell*, takes a sequence of tactics and a refinement cell and tries to justify the refinement proof obligations using the given tactics. The size of the given list of tactics must be equal to the number of refinement proof obligations. It returns a refinement cell with the same cell, but the proof obligations may change: the refinement proof

obligations have either been discharged, and therefore eliminated (possibly giving rise to predicate proof obligations), or left unchanged (if the corresponding tactic is not adequate for proving the refinement).

$$
\begin{array}{|l}
justifycell : (\mathrm{seq}\, Tactic) \rightarrow RCell \nrightarrow RCell \\
\hline
\forall ts : \mathrm{seq}\, Tactic;\ rc : RCell \mid \#ts = \#(getPObsRef\ rc) \bullet justifycell\, ts\, rc = \\
\qquad \textbf{let}\ opobs == (getPObsPred(rc), \langle\rangle);\ npobs == ts\, justifyl\, (getPObsRef\ rc) \bullet \\
\qquad\quad (getCell\, rc, MPObs\, (opobs, npobs))
\end{array}
$$

The functions *getPObsPred* and *getPObsRef* yield the predicate proof obligations and the refinement proof obligations of a given refinement cell, respectively.

Finally, we have the tactical *JUSTIFIED_BY* presented below; it applies a given tactic and tries to discharge the refinement proof obligations raised by this application. It takes a tactic t that must be applied and a sequence of tactics ts that must be used to discharge (or decompose) the refinement proof obligations. It achieves its task by applying ts to each of the refinement cells resulting from the application of t.

$$
\begin{array}{|l}
\_\, JUSTIFIED\_BY\, \_ : (Tactic \times \mathrm{seq}\, Tactic) \nrightarrow Tactic \\
\hline
\forall t : Tactic;\ ts : \mathrm{seq}\, Tactic;\ r : RCell \bullet (t\ JUSTIFIED\_BY\ ts)\, r = (justifycell\, ts) * (t\, r)
\end{array}
$$

With this tactical, and others in [33], we can apply the *Circus* refinement calculus in a more effective and simple way. It is important to observe that the definition of such a tactical is not possible in the context of ArcAngel; it is a feature of ArcAngel*C* and its model, which records the structure of proof obligations.

The semantics of ArcAngel*C* is an extension of that of ArcAngel, for which we already proved 66 laws [30]. Of course, we have made some important changes regarding the structural combinators, and added new tacticals. The proofs of the laws of ArcAngel that do not involve structural combinators, however, are valid for ArcAngel*C*. In addition, the laws involving the combinators are also valid in the new semantics, and the structure of their proofs are the same, since these proofs do not depend on the particular structure of a refinement cell. Finally, since the original proofs are modular, the proofs of properties of the new structural combinators are a simple adaptation of the already existing proofs for the legacy structural combinators.

For added validation, we have mechanised ArcAngel*C* in a theorem prover, and we are using this mechanisation in an industrial case study as we discuss in the next section.

7. Mechanisation of ArcAngel*C*

In [52] we present an implementation of the ArcAngel tactic language for the ProofPower-Z theorem prover, which has been generalised and customised to support ArcAngel*C* as well. In its design, we use a very direct translation of the ArcAngel*C* semantics presented here; in particular, refinement cells in the semantic model are identified with refinement theorems in the implementation model. A faithful representation of partial and infinite lists is achieved through the use of lazy lists at the code level, and recursive ArcAngel*C* tactics are directly supported through recursive ML functions on tactics. Using this mechanisation, the encoding of particular tactics is mostly trivial, and the LCF paradigm adopted by ProofPower-Z effectively guarantees that all refinements constructed via the tactic language implementation are sound with respect to the underlying semantic model of *Circus*, which we present and discuss in detail in [36].

7.1. Overview

To capture the application of ArcAngel*C* tactics, we introduce the notion of a refinement theorem. Such a theorem is of the form $\Gamma \vdash A \sqsubseteq B$, where Γ is a list of assumptions recording the proof obligations, and both A and B are program expressions. The symbol $\sqsubseteq$ here represents refinement between *Circus* actions or processes, but owing to the generality of our design may represent any reflexive and transitive relation.

The application of a refinement law $\Gamma_2 \vdash X \sqsubseteq Y$ to a refinement theorem $\Gamma_1 \vdash A \sqsubseteq B$ produces a new refinement theorem $\Gamma_1, \Gamma_2 \vdash A \sqsubseteq B'$ where B' is a valid refinement of B under the additional provisos Γ_2, inherited from the law. The new theorem is obtained by first matching the left-hand program X of the refinement law against B. This gives an instantiation $\Gamma_2 \vdash B \sqsubseteq B'$ of the law which, by transitivity of refinement, permits the prover to conclude $\Gamma_1, \Gamma_2 \vdash A \sqsubseteq B'$. In summary, in our ProofPower-Z encoding, ArcAngel*C* tactics apply to refinement theorems rather than program expressions (refinement cells in the semantics). The underlying mechanism of tactic application transforms the second operand of a given refinement.

Our approach unifies program transformations in ArcAngel*C* with the design of ProofPower-Z, which is centred on theorem-generating functions. The application of an ArcAngel*C* tactic to a program X can be achieved by first creating an initial refinement theorem $\vdash X \sqsubseteq X$ that is trivially proved by reflexivity of refinement. Next, we apply the encoding of the ArcAngel*C* tactic to this theorem. If successful, it returns a theorem $\Gamma \vdash X \sqsubseteq Y$ encapsulating the transformation of X to Y. This theorem can then be included in the hypotheses of some larger proof where it may be used in either a forward-chaining

or backward-chaining manner: to derive new refinement conjectures or to reduce or solve an existing (refinement) proof goal.

The validity of the refinement is established by the soundness of the primitive inferences of ProofPower-Z's core logic; for that reason, it is independent of our actual implementation of ArcAngel*C* which merely drives the core prover. This protection we do not get in an implementation of ArcAngel based on rewrite systems such as Gabriel [38], since in those the validity of rules and laws is not independently verified.

To accommodate angelic execution of tactics, we keep track of all possible outcomes of a tactic; we characterise tactics as functions mapping refinement theorems to lists of refinement theorems. This representation closely resembles the semantic model of ArcAngel*C* presented in Section 6 that models tactics by functions mapping refinement cells to (infinite) lists of refinement cells. Our design is suitable for tactics with finite and infinite behaviours, but extra care is required to cater for tactics that can generate an infinite number of outcomes, or otherwise fail to produce any result due to nontermination of recursive tactics.

The tactic EXHAUST presented on page 796 illustrates this case. Its application has potentially an infinite number of outcomes. Operationally, this results in an infinite list to be computed when EXHAUST(t) is applied to some program. From a computational point of view, this evaluation may not terminate. We need, however, to make the distinction between this application and **abort**, because the application of EXHAUST(t) has a well-defined behaviour as long as we are not attempting to utilise (evaluate) all outcomes.

We, therefore, adopt lazy evaluation when computing the outcomes of tactic applications. Specifically, we introduce a data type `lazylist` that allows us to defer evaluation of tactics until we actually require their results. Since evaluation in SML is strict, lazy evaluation must be simulated by means of additional layers of functions with a spurious argument. For example, evaluation of `t` is deferred in `(fn () => t p)` until we apply the function to an empty tuple (the construction `(fn` *args* `=>` *body*`)` is used in Standard ML for anonymous functions). Lazy versions for most of the functions on lists that have been used to define the semantics of ArcAngel*C* are provided and utilised in the implementation of the various operators.

To support parametrised tactics as well as the **applies to** p **do** t operator, we use a special notion of environment that binds meta-variables to expressions. They are represented by a list of pairs of ProofPower-Z terms, where the first component gives the variable, and the second, the bound expression. Environments are in most cases just propagated to the operands in tactic combinators; the exceptions are **applies to** p **do** t, and law and tactic applications, which actually need to process them.

To conclude, ArcAngel*C* tactics are encoded by functions that map environments and refinement theorems to lazy lists of refinement theorems.

7.2. Implementation details

The basic tactic `TSkip` returns a singleton lazy list containing the program to which the tactic is applied. The implementation of the constructor for law applications, `TLaw`, is more elaborate: it substitutes meta-variables occurring free in the arguments, and moves implications in the conclusion of the law theorem to the assumptions. Laws have to be declared using the `TLawDecl` function; it expects the name of the law, its formal arguments as a list of typed terms, and the corresponding ProofPower-Z theorem. For instance, we consider the refinement law that strengthens the postcondition of a specification statement.

Law 34 (str-post) $w : [pre, post] \sqsubseteq w : [pre, post'] \textbf{ provided } post' \Rightarrow post$

It can be declared with the following command, which identifies its formal parameters.

```
TLawDecl "strPost" [⌜postC' ⦂⊕ ALPHA_PREDICATE⌝] strPost_thm;
```

We assume that the SML constant `strPost_thm` has been initialised to hold the law theorem. The first argument `"strPost"` specifies the name of the declared law in ArcAngel*C*, and the second argument supplies the list of quantified variables used as parameters. The variables are given as (ProofPower-Z) variable terms whose type must be explicitly specified using the '$\overset{\oplus}{\oplus}$' operator of ProofPower-Z for type annotation.

Similarly, we can declare a tactic using the `TTacDecl` function and apply it using `TTactic`. The function `TTacDecl` corresponds to the **Tactic** *name*(*args*) t **end** construct of ArcAngel.

The implementation of the tacticals is also very close to their respective semantic definitions. For example, the SML implementation of sequence is provided by `TSeq` as follows.

```
fun (t1 : AA_TACTIC) TSeq (t2 : AA_TACTIC) : AA_TACTIC =
   (fn env : ENV => (fn p : REF_THM => (lazyflat (lazymap (t2 env) (t1 env p)))));
```

It is a literal translation of the semantics where the distributed concatenation on infinite lists $\overset{\infty}{\frown}$ is encoded by `lazyflat`, and the mapping function $*$ is encoded by `lazymap`. These two SML functions perform operations on lazy lists similar to the semantic functions on infinite lists.

In order to support the structural combinators of ArcAngel*C*, we provide a collection of constructor functions: that is `MakeUnaryTSC`, `MakeBinaryTSC` and `MakeNaryTSC`. They are respectively used to construct structural combinator tactics for arbitrary unary, binary, or *n*-ary operators of the program syntax. Each function requires the operator for which the combinator is defined, the positions of its arguments to which the tactic(s) of the structural combinator are applied, and two monotonicity theorems. The signature of, for example, `MakeBinaryTSC` for creating binary structural combinators is shown below.

```
fun MakeBinaryTSC (area : string, op_tm : TERM, (arg1_pos : int, arg2_pos : int),
   eq_mon_thm : THM, ref_mon_thm : THM);
```

The `area` parameter is used to specify the name of the SML function of the concrete combinator; it is primarily used for error reporting. Next, `op_tm` determines the program operator for which the structural combinator is defined. The following two arguments identify the positions of the operands the two tactics of the structural combinator are applied to. Finally, we require two monotonicity theorems, one for equivalence and one for refinement. Although we do not elaborate on this, our implementation automatically handles both refinement and equivalence-preserving program transformations, explaining the need for two theorems.

For the combinator $\boxed{\sqcap}$, for instance, the refinement theorem has to be of the following shape.

$$\vdash \forall p_1, p_1', p_2, p_2' : T \bullet p_1 \sqsubseteq p_1' \land p_2 \sqsubseteq p_2' \Rightarrow p_1 \sqcap_C p_2 \sqsubseteq p_1' \sqcap_C p_2'$$

where $\_ \sqcap_C \_$ is the semantic function encoding the program operator for internal choice of actions.

The general strategy to mechanise structural combinators of ArcAngel*C* is first to prove the two monotonicity theorems within the semantic model of *Circus*, and then define new constants

```
val TSC_NAME = MakeBinaryTSC (...);
```

by calling the constructor function. Here `NAME` is the name for the combinator tactic; for our example, we use `TSC_Choice`. The constant can then be used as a standard ArcAngel*C* binary tactical.

In this way we can easily define structural combinators for all action and process operators of ArcAngel*C*; the only noteworthy effort is the proof of monotonicity theorems. The configuration of *n*-ary structural combinators requires a more elaborate monotonicity theorem, but in principle follows the same approach.

To illustrate the encoding of tactics we encode the interIntroAndSimpl tactic presented in Section 6.3. This is achieved by the following SML statement.

```
TTacDecl "interIntroAndSimpl" []
   (TAppliesTo ⨾A ⟦C (ns1, cs, ns2) ⟧C Skip_C u ⌝ TDo
      (TLaw "par-inter" []) TSeq (TLaw "inter-unit" []));
```

The new tactic is declared using `TTacDecl` specifying its name and arguments (none in this example). Within the body of the tactic, `TAppliesTo` and `TDo` implement the **applies to** _ **do** _ operator of ArcAngel*C*. Its first argument specifies the pattern against which the right-hand side of the refinement conjecture is matched to determine whether the tactic is applicable. In the pattern, we use two semantic functions in our model of *Circus*, namely $\_ ⟦_C \_ ⟧_C \_$ for parallel composition and $Skip_C$ for ***Skip***. Subsequently, the two sequenced law invocations are performed by the `TLaw` tactical. At this point we assume that the laws par-inter and inter-unit have already been declared via `TLawDecl`, given suitable ProofPower-Z law theorems.

The source code for the tool, including examples, is at www.cs.york.ac.uk/circus/tp/tools.html. It is currently used to mechanise the refinement strategy presented in Section 4.

8. Related work

Backtracking, which is the common implementation strategy for angelic choice, has been recognised as a useful mechanism for the construction of tactics in many tools. ArcAngel*C*'s implicit backtracking manifests itself in the following law: $(t_1 \mid t_2)\ ;\ t_3 = (t_1\ ;\ t_3) \mid (t_2\ ;\ t_3)$. One possible representation of this tactic in ProofPower-Z may use the tactical `ORELSE` to represent alternation, and `THEN` to represent sequencing. However, $(t_1$ `ORELSE` $t_2)$ `THEN` t_3 would not have the same operational behaviour as the ArcAngel*C* tactic $(t_1 \mid t_2)\ ;\ t_3$. The ProofPower-Z tactic would first apply t_1, and if this fails resolve to applying t_2. The choice of either applying t_1 or t_2, however, is not revised if t_3 subsequently turns out to fail. In general, `ORELSE` acts like a cut on alternation; the choice it makes is not a provisional one unless t_1 on its own fails, in which case t_2 is executed. PVS, on the other hand, does provide built-in support for backtracking tactics via its *try* tactical, but failure and backtracking are treated as distinct outcomes of tactic applications [18].

The use of HOL to prove the correctness of refinement rules, and the application of these rules to formalise program refinement was presented by Back and von Wright in [2]. Later, von Wright and his colleagues described the use of HOL in the application of rules from a refinement calculus [48]. In this work, they formalise methods for data refinement as well. The aim of their work was to use HOL to generate the verification conditions for the refinement steps. ML was used to program simple tactics. While ML does have a formal semantics, it is difficult to reason about tactics written in such a general language.

The Program Refinement Tool, PRT [4], was developed at Queensland. It is based on Ergo and originally inherited its tactic language. Later versions include Gumtree [24,25], a tactic language based on Angel.

The association of a notation for formula manipulation and an editor for the development of programs by stepwise refinement in Proxac was presented in [45]. Although the idea seems not to have been taken further, it suggests viewing program transformations as the commands in a programming language for formula manipulation including function composition, conditionals, and a fixed-point operator.

A refinement tool with special emphasis on tactics was presented by Groves et al. in [13,28]. The tool was implemented in Prolog and provides all programming capabilities to the user. A collection of examples showing the Prolog encoding of various common development patterns (like introducing various kinds of loops) was presented by Nickson in his thesis [28]. All these tactics are expressed directly in Prolog, rather than a special tactic language. Hence, the tactics have all the normal Prolog control mechanisms and can do arbitrary computations in deciding what steps to take and in constructing new components, but all modifications to the program being constructed are done by applying refinement rules.

Grundy's refinement tool [14] is based on window inference and on the HOL theorem prover. Since this system is programmable, users can add their own commands to automate refinements that they frequently repeat. Tactics are written using simple ML tacticals. In [3], an extension of this work with a more user-friendly interface is presented. The authors mention that ML can be used to package transformations, but point out themselves that this brings the difficulty of requiring knowledge of HOL and ML.

The Red refinement tool was developed by Gardiner and Vickers [46]. It was implemented in Prolog and contains three interactive windows. The first window shows the current state of the program being developed. The second displays the proof obligations. Finally, the third window presents the refinement operations performed to obtain the program. The authors also developed a language to describe the refinement tree [47]. Basically, for every construct of the target programming language, there is a corresponding construct in the transformation language. This idea was the origin of Angel's *structural combinators* [22]. This was not, however, a language to describe tactics. Instead, it was used to describe refinements. There were no constructs for alternation or recursion. A survey of further early refinement tools can be found in [5].

In [39], Owre and Shankar present a language for defining PVS proof strategies and describe how it can be used to define advanced proof strategies. The tactic language is based on Common Lisp and allows PVS users to construct proofs and define strategies at a higher level than that using PVS's primitive proof commands. A formal semantics for this language is left as future work.

Delahaye proposes a new tactic language $\mathcal{L}_{tac}$ for Coq in [10]. The language has a functional core (based on the Objective Caml programming language) with recursion and matching operators for Coq terms and proof contexts. Delahaye's language can be used in tactic definitions and proof scripts. The definition of a formal semantics for $\mathcal{L}_{tac}$ is not the aim of [10] and, as far as we know, it is not available.

We are currently developing a tool, *C*Refine [15,37], that is based on the work presented in [50,38] to provide automated support for the application of the *Circus* refinement calculus in an environment more friendly than that of a theorem prover. In the near future, we intend to include support for tactics written in ArcAngel*C*; using this extension, one may then specify refinement tactics like those presented in this paper, and apply them just like refinement laws. In *C*Refine, however, the validity of the refinement laws is not established by the soundness of the primitive inferences of a theorem prover's core logic as it is in the work presented here. Instead, the refinement laws, which have already been manually proved in [32], are implemented and tested exhaustively before they can be used within the tool.

9. Conclusions

We have presented ArcAngel*C*, a refinement-tactic language that extends ArcAngel and can be used in the formalisation of refinement strategies for concurrent state-rich programs in *Circus*. ArcAngel*C* tactics can be used as single transformation rules, and hence, shorten developments, and automate verification strategies. In particular, we have formalised the first two phases of a refinement strategy proposed in [6]; it is used to verify Ada programs with respect to Simulink diagrams using *Circus*, and has been applied to industrial applications. The formalisation of the verification strategy as a tactic gives a clear route to automation.

Despite being a small example, the case study discussed here was proposed to us by QinetiQ, and its implementation is representative of the architectural pattern used for the development of their safety-critical applications in avionics. The verification process adopted by QinetiQ already uses Z and CSP independently to check different aspects of these systems, namely, functionality and scheduling, separately. *Circus* and the refinement strategy that we formalise allows the verification of those aspects as part of a single formal argument. The refinement technique has been developed in conjunction with QinetiQ. With the use of *Circus*, we have managed to enlarge the set of properties and systems that can

be checked, without increasing the proof burden, and therefore, the costs. QinetiQ intends to use our strategy (and tools) in the verification of some of their safety-critical systems.

We have defined the semantics of ArcAngel*C* formally using Z as a meta-language. It is based on the notion of a goal modelled by a refinement cell (an element of the type *RCell*). This is a pair with a program as its first element and a pair of sequences of proof obligations as its second element: the first sequence contains standard proof obligations, and the second those stated as further refinements. The application of a tactic to an *RCell* returns a list of *RCell*s containing the possible output programs with their corresponding proof obligations. We handle 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 addition, based on this semantic model, we can use ArcAngel*C* to handle the justification of the proof obligations stated as refinements. For that, we rely on the order of the proof obligations generated.

For structural combinators, we have identified a pattern in their semantic definitions that can be applied for the definition of the semantics of tactic languages for other refinement notations, besides *Circus*.

In [29], we have shown the soundness of algebraic laws for reasoning about ArcAngel tactics. We covered most of the laws that had been previously proposed for Angel. With these laws, the strategy that had been proposed to reduce finite Angel tactics to a normal form could be applied to ArcAngel tactics. As future work, we intend to prove that these laws are also valid in the context of ArcAngel*C*. Since our semantics of ArcAngel*C* is a natural extension of the semantics of ArcAngel, it is possible that these proofs will be relatively simple. They remain, however, still to be done.

Correctness of tactics in the sense of achieving the required program derivations is a challenging issue. It is, however, perhaps no more challenging than writing correct programs. In this paper, we have provided a formal semantics for ArcAngel*C*, the "programming language" for tactics of refinement. This semantics can support the justification of a number of laws that can be used to reason about tactics. In particular, the implementation of ArcAngel*C* that we discuss in this paper is a basis for reasoning in that way, since it encodes the semantics of ArcAngel*C*. As said, we, however, leave work on laws of tactics to the future.

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

Acknowledgements

CNPq supports the work of Marcel Oliveira: grant 550946/2007-1, 573964/2008-4, 620132/2008-6, and 476836/2009-3.

Appendix A. Infinite lists

We present in this appendix the model for infinite lists adopted here; basically, we reproduce the definitions previously presented in [21]. The set of the finite and partial sequences of members of X is defined as

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

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

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

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

$$[X]$$
$$\mathsf{chain} : \mathbb{P}(\mathbb{P}(\mathsf{pfseq}[X]))$$
$$\forall\, c : \mathbb{P}(\mathsf{pfseq}[X]) \bullet c \in \mathsf{chain} \Leftrightarrow (\forall\, x, y : c \bullet x \leq y \lor y \leq x)$$

A new constructor is defined below; finite and partial lists will be modelled by chains of pfseqs, which have the required list as their maximum element. All these chains must be prefix closed (as defined by pchain) so that each finite, partial or infinite list is denoted by a unique chain.

$$[X]$$
$$\mathsf{pchain} : \mathbb{P}(\mathsf{chain}[X])$$
$$\forall c : \mathsf{chain}[X] \bullet c \in \mathsf{pchain} \Leftrightarrow (\forall x : c;\ y : \mathsf{pfseq}[X] \mid y \leq x \bullet y \in c)$$

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

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

The definitions of the functions used in this paper are the object of the material in the rest of this appendix. The function *pfmap* maps the function f to the second element of x.

$$[X, Y]$$
$$\_\mathsf{pfmap}\_ : ((X \to Y) \times \mathsf{pfseq}[X]) \to \mathsf{pfseq}[Y]$$
$$\forall xs : \mathsf{seq}\, X;\ f : X \to Y;\ pf : PF \bullet f\ \mathsf{pfmap}\ (pf, xs) = (pf, (f \circ xs))$$

The map function $*$ maps a function f to each element of a possibly infinite list.

$$[X, Y]$$
$$\_ * \_ : ((X \to Y) \times \mathsf{pfiseq}[X]) \to \mathsf{pfiseq}[Y]$$
$$\forall c : \mathsf{pfiseq}[X];\ g : X \to Y \bullet g\ *\ c = \{x : c \bullet g\ \mathsf{pfmap}\ x\}$$

The distributed concatenation returns the concatenation of all the elements of a possibly infinite list of possibly infinite lists. The function $\bigsqcup_\infty$ gives the maximum element of a finite chain.

$$[X]$$
$$\overset{\infty}{\frown}/ : \mathsf{pfiseq}[\mathsf{pfiseq}[X]] \to \mathsf{pfiseq}[X]$$
$$\forall s : \mathsf{pfiseq}[\mathsf{pfiseq}[X]] \bullet \overset{\infty}{\frown}/ s = \bigsqcup_\infty \{c : s \bullet \overset{\infty}{\frown}/ c\}$$

It uses the function $\overset{\infty}{\frown}/$, which is the distributed concatenation for $\mathsf{pfseq}(\mathsf{pfiseq}\, X)$.

$$[X]$$
$$\overset{\infty}{\frown}/ : \mathsf{pfseq}[\mathsf{pfiseq}[X]] \to \mathsf{pfiseq}[X]$$
$$\overset{\infty}{\frown}/(\mathsf{finite}, \langle\rangle) = \{\}$$
$$\overset{\infty}{\frown}/(\mathsf{partial}, \langle\rangle) = \{(\mathsf{partial}, \langle\rangle)\}$$
$$\forall g : \mathsf{pfiseq}[X] \bullet \overset{\infty}{\frown}/(\mathsf{finite}, \langle g\rangle) = g \wedge \overset{\infty}{\frown}/(\mathsf{partial}, \langle g\rangle) = g \overset{\infty}{\frown} \{(\mathsf{partial}, \langle\rangle)\}$$
$$\forall gs, hs : \mathsf{seq}(\mathsf{pfiseq}[X]) \bullet$$
$$\overset{\infty}{\frown}/(\mathsf{finite}, gs \frown hs) = (\overset{\infty}{\frown}/(\mathsf{finite}, gs)) \overset{\infty}{\frown} (\overset{\infty}{\frown}/(\mathsf{finite}, hs))$$
$$\wedge\ \overset{\infty}{\frown}/(\mathsf{finite}, gs \frown hs) = (\overset{\infty}{\frown}/(\mathsf{finite}, gs)) \overset{\infty}{\frown} (\overset{\infty}{\frown}/(\mathsf{partial}, hs))$$

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

$$[X]$$
$$\_ \overset{\infty}{\frown} \_ : (\mathsf{pfiseq}[X] \times \mathsf{pfiseq}[X]) \to \mathsf{pfiseq}[X]$$
$$\forall a, b : \mathsf{pfiseq}[X] \bullet a \overset{\infty}{\frown} b = \{x : a;\ y : b \bullet x \frown y\}$$

where the function $\frown$ is the concatenation function for $\mathsf{pfseq}\, X$ defined as

$$[X]$$
$$\_ \,^\frown\, \_ : (\mathrm{pfseq}[X] \times \mathrm{pfseq}[X]) \to \mathrm{pfseq}[X]$$
$$\begin{aligned}
&\forall gs, hs : \mathrm{seq}\, X;\ s : \mathrm{pfseq}[X] \bullet \\
&\quad (\mathsf{finite}, gs) \,^\frown\, (\mathsf{finite}, hs) = (\mathsf{finite}, gs \frown hs) \\
&\quad \land\ (\mathsf{finite}, gs) \,^\frown\, (\mathsf{partial}, hs) = (\mathsf{partial}, gs \frown hs) \\
&\quad \land\ (\mathsf{partial}, gs) \,^\frown\, s = (\mathsf{partial}, gs)
\end{aligned}$$

The function $^\circ$ applies a possibly infinite list of functions to a single argument.

$$[X, Y]$$
$$\_ \,^\circ\, \_ : (\mathrm{pfiseq}[X \nrightarrow Y] \times X) \to \mathrm{pfiseq}[Y]$$
$$\begin{aligned}
&\forall fs : \mathrm{pfiseq}[X \nrightarrow Y];\ f : X \nrightarrow Y;\ x : X \bullet \\
&\quad \bot[X \nrightarrow Y] \,^\circ\, x = [\,]_\infty \land [\,]_\infty \,^\circ\, x = [\,]_\infty \land (f :_\infty fs) \,^\circ\, x = (f\ x) :_\infty (fs \,^\circ\, x)
\end{aligned}$$

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

$$[X]$$
$$\Pi_\infty : \mathrm{seq}(\mathrm{pfiseq}[X]) \to \mathrm{pfiseq}[\mathrm{seq}\, X]$$
$$\begin{aligned}
&\forall xss : \mathrm{seq}(\mathrm{pfiseq}[X]) \bullet \\
&\quad \Pi_\infty\, xss = \\
&\qquad \mathbf{if}(\#xss = 0)\ \mathbf{then}\ [\,]_\infty \\
&\qquad \mathbf{else\ if}(\#xss = 1)\ \mathbf{then}\ \mathsf{e2l}\ *\ (head\, xss) \\
&\qquad\quad \mathbf{else}(head\, xss)\ \mathsf{seqcons}_\infty\ (\Pi_\infty (tail\, xss))
\end{aligned}$$

The function e2l receives an element x and returns a singleton that has x as its only element.

$$[X]$$
$$\mathsf{e2l} : X \to \mathrm{seq}\, X$$
$$\forall x : X \bullet \mathsf{e2l}\, x = \langle x \rangle$$

The function seqcons takes a sequence xs and a sequence of sequences xxs. It defines a sequence of sequences: for every x from xs, and sequence ys from xxs the resulting sequence contains a sequence $x : ys$.

$$[X]$$
$$\_\mathsf{seqcons}\_ : (\mathrm{seq}\, X \times \mathrm{seq}(\mathrm{seq}\, X)) \to \mathrm{seq}(\mathrm{seq}\, X)$$
$$\begin{aligned}
&\forall x : X;\ xs : \mathrm{seq}\, X;\ xss : \mathrm{seq}(\mathrm{seq}\, X) \bullet \\
&\quad \langle\rangle\ \mathsf{seqcons}\ xss = \langle\rangle \\
&\quad \land\ (x\ :\ xs)\ \mathsf{seqcons}\ xss = \{i : 1..(\#xss) \bullet i \mapsto x\ :\ xss(i)\} \frown (xs\ \mathsf{seqcons}\ xss)
\end{aligned}$$

The function $\mathsf{seqcons}_\infty$ lifts the function seqcons to possibly infinite lists.

$$[X]$$
$$\_\mathsf{seqcons}_\infty\_ : (\mathrm{pfiseq}[X] \times \mathrm{pfiseq}[\mathrm{seq}\, X]) \to \mathrm{pfiseq}[\mathrm{seq}\, X]$$
$$\begin{aligned}
&\forall xs : \mathrm{pfiseq}[X];\ xss : \mathrm{pfiseq}[\mathrm{seq}\, X] \bullet \\
&\quad [\,]_\infty\ \mathsf{seqcons}_\infty\ xss = [\,]_\infty \\
&\quad \land\ \mathsf{max}(xs\ \mathsf{seqcons}_\infty\ xss) = \\
&\qquad \mathbf{let}\ sc == ((\mathsf{max}\, xs).2)\ \mathsf{seqcons}\ ((\mathsf{max}\, xss).2) \bullet \\
&\qquad\quad \mathbf{if}((\mathsf{max}\, xss).1 = \mathsf{partial}) \\
&\qquad\quad \mathbf{then}(\mathsf{partial}, (\langle head((\mathsf{max}\, xs).2)\rangle)\ \mathsf{seqcons}\ ((\mathsf{max}\, xss).2)) \\
&\qquad\quad \mathbf{else}((\mathsf{max}\, xs).1, sc)
\end{aligned}$$

The max operator returns the sequence that is denoted by a pchain in the partial/finite case.

$$[X]$$
$$\mathsf{max}\_ : \mathrm{pfiseq}[X] \nrightarrow \mathrm{pfseq}[X]$$
$$\forall x : \mathrm{pfiseq}[X] \mid x \in \mathbb{F}(\mathrm{pfseq}[X]) \bullet \mathsf{max}(x) \in x \land (\forall z : x \bullet z \leq \mathsf{max}(x))$$

For every element x of a certain type X and xs a (possible infinite) sequence of elements of the same type, $x\ \mathsf{in}_\infty\ xs$ if, and only if, x is a member of the sequence xs.

$$
\begin{array}{|l}
\hline
[X] \\
\_\mathsf{in}_\infty\_ : X \leftrightarrow \mathrm{pfiseq}[X] \\
\hline
\forall x : X;\ xs : \mathrm{pfiseq}[X] \bullet \\
\quad (x\ \mathsf{in}_\infty\ xs) \Leftrightarrow (\exists ys, zs : \mathrm{pfiseq}[X] \bullet xs = ys \overset{\infty}{\frown} [x]_\infty \overset{\infty}{\frown} zs) \\
\hline
\end{array}
$$

More information about all these definitions and others can be found in [21].

Appendix B. Laws of refinement

This appendix lists all the *Circus* refinement laws mentioned in this paper in alphabetical order of their labels. They have all been previously published and justified. The side conditions of some of them involve meta-functions: the function *usedV* identifies the set of used variables (read, but not written) in the given actions or processes; the function *wrtV* gives the set of variables that are written by a given action; the function *usedC* returns a set of all channels mentioned in an action; the function *initials* gives a set containing all the events in which an action is initially willing to synchronise; the function α determines the set of components of a given schema; finally, *FV* is a function that defines the free variables of a predicate or expression.

Law 1 (*assign-intro*). $w : [pre, post] \sqsubseteq_{\mathcal{A}} x := e$ **provided** $pre \Rightarrow post[e/x]$

In the next law, we use $\mathbf{L}(n)$ to denote the fact that the **L**ocal action definitions may include references to the action n; the same holds for the **M**ain **A**ction $\mathbf{MA}(n)$. Later references to $\mathbf{L}(A)$ and $\mathbf{MA}(A)$ are the result of substituting the body A of n for some or all occurrences of n in **L** and **MA**.

Law 2 (*copy-rule-action*).

$$
\begin{array}{l}
\textbf{\textit{begin}}\ (\textbf{\textit{state}}\ S)\ (n \mathrel{\widehat=} A)\ \mathbf{L}(n) \bullet \mathbf{MA}(n)\ \textbf{\textit{end}} \\
= \\
\textbf{\textit{begin}}\ (\textbf{\textit{state}}\ S)\ (n \mathrel{\widehat=} A)\ \mathbf{L}(A) \bullet \mathbf{MA}(A)\ \textbf{\textit{end}}\,.
\end{array}
$$

In the law below, we use α_P to denote the interface of the process P.

Law 3 (*hid-contract*). $(P\ \alpha_P) \setminus cs = (P\ \alpha_P) \setminus cs'$. **where** $cs' = cs \cap \alpha_P$.

Law 4 (*hid-idem*). $A \setminus cs = A$ **provided** $usedC(A) \cap cs = \varnothing$.

Law 5 (*hid-inter-dist*). $(A_1 \llbracket ns_1 \mid ns_2 \rrbracket A_2) \setminus cs_2 = (A_1 \setminus cs_2) \llbracket ns_1 \mid ns_2 \rrbracket (A_2 \setminus cs_2)$.

Law 6 (*hid-par-dist*). $(A_1 \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket A_2) \setminus cs_2 = (A_1 \setminus cs_2) \llbracket ns_1 \mid cs_1 \mid ns_2 \rrbracket (A_2 \setminus cs_2)$.
provided $cs_1 \cap cs_2 = \varnothing$.

Law 7 (*hid-rec-dist*). $(\mu X \bullet A) \setminus cs = \mu X \bullet A \setminus cs$.

Law 8 (*hid-seq-dist*). $(A_1 \fatsemi A_2) \setminus cs = (P_1 \setminus cs) \fatsemi (P_2 \setminus cs)$.

Law 9 (*hid-step*). $(c \longrightarrow A) \setminus cs = A \setminus cs$. **provided** $c \in cs$.

Law 10 (*inter-comm*). $A_1 \llbracket ns_1 \mid ns_2 \rrbracket A_2 = A_2 \llbracket ns_2 \mid ns_1 \rrbracket A_1$.

Law 11 (*inter-index*).

$$
\begin{array}{l}
\Big|\Big|\Big| x : \{v_1, \ldots, v_n\} \bullet \llbracket ns(x) \rrbracket A(x) \\
= \\
A(v_1) \llbracket ns(v_1) \mid \bigcup \{x : \{v_2, \ldots, v_n\} \bullet ns(x)\} \rrbracket (\ldots (A(v_{n-1}) \llbracket ns(v_{n-1}) \mid ns(v_n) \rrbracket A(v_n)))\,.
\end{array}
$$

Law 12 (*inter-unit*). $A \llbracket ns_1 \mid ns_2 \rrbracket \textbf{\textit{Skip}} = A$.

Law 13 (*inter-seq-assign*). $v_1 := e_1 \llbracket ns_1 \mid ns_2 \rrbracket v_2 := e_2 = v_1 := e_1 \fatsemi v_2 := e_2$
provided $v_1 \notin \{v_2\} \cup FV(e_2)$; $v_1 \in ns_1$; *and* $v_2 \in ns_2$.

Law 14 (*inter-seq-extract-snd*). $(A_1 \fatsemi A_2) \llbracket ns_1 \mid ns_2 \rrbracket A_3 = (A_1 \llbracket ns_1 \mid ns_2 \rrbracket A_3) \fatsemi A_2$
provided $usedC(A_2) = \varnothing$; $usedV(A_2) \cap wrtV(A_3) = \varnothing$;
$wrtV(A_1) \subseteq ns_1 \cup ns_1'$; *and* $wrtV(A_2) \subseteq ns_1 \cup ns_1'$.

Law 15 (*inter-unused-name*). $A_1 \llbracket \{x\} \cup ns_2 \mid ns_3 \rrbracket A_2 = A_1 \llbracket ns_2 \mid ns_3 \rrbracket A_2$
provided $\{x, x'\} \cap wrtV(A_1) = \varnothing$

Law 16 (*join-blocks*). $\textbf{\textit{var}}\ x : T_1 \bullet \textbf{\textit{var}}\ y : T_2 \bullet A = \textbf{\textit{var}}\ x : T_1;\ y : T_2 \bullet A$.

We use $CPars_i(P)$ to denote paragraphs that may refer to the process P. For example, $CPars_1\ ((P_1\ \alpha_P) \setminus cs)$ denotes paragraphs that may refer to $(P_1\ \alpha_P) \setminus cs$. The law below replaces these references with references to $P_2\ (\alpha_P \setminus cs)$. In addition, we use $PPars_i$ and $QPars_i$ to denote sequences of process paragraphs.

Law 17 (*join-proc-hid*)**.**

$$
\begin{array}{l}
CPars_1\ ((P_1\ \alpha_P) \setminus cs) \\
P_1 \mathrel{\widehat=} \textbf{\textit{begin}}\ PPars_1\ \textbf{\textit{state}}\ P\_State \mathrel{\widehat=} SExp_P\ PPars_2 \bullet PAct\ \textbf{\textit{end}} \\
CPars_2\ ((P_1\ \alpha_P) \setminus cs) \\
\quad = \\
CPars_1\ ((P_2\ (\alpha_P \setminus cs))) \\
P_2 \mathrel{\widehat=} \textbf{\textit{begin}}\ PPars_1\ \textbf{\textit{state}}\ P\_State \mathrel{\widehat=} SExp_P\ PPars_2 \bullet PAct \setminus cs\ \textbf{\textit{end}} \\
CPars_2\ ((P_2\ (\alpha_P \setminus cs)))
\end{array}
$$

provided $P_1 \notin \alpha(CPars_1\ (P_2)) \cup \alpha(CPars_2\ (P_2))$; *and* $P_2 \notin \alpha(CPars_1\ (P_1)) \cup \alpha(CPars_2\ (P_1))$.

Law 18 (*join-proc-par*)**.**

$$
\begin{array}{l}
CPars_1\ (P \parallel Q) \\
P \mathrel{\widehat=} \textbf{\textit{begin}}\ PPars_1\ \textbf{\textit{state}}\ P\_St \mathrel{\widehat=} SExp_P\ PPars_2 \bullet PAct\ \textbf{\textit{end}} \\
CPars_2\ (P \parallel Q) \\
Q \mathrel{\widehat=} \textbf{\textit{begin}}\ QPars_1\ \textbf{\textit{state}}\ Q\_St \mathrel{\widehat=} SExp_Q\ QPars_2 \bullet QAct\ \textbf{\textit{end}} \\
CPars_3\ (P \parallel Q) \\
= \\
P\_Q \mathrel{\widehat=} \quad \textbf{\textit{begin}} \quad \textbf{\textit{state}}\ P\_Q\_St \mathrel{\widehat=} P\_St \wedge Q\_St \\
\qquad\qquad\qquad PPars_1 \\
\qquad\qquad\qquad PPars_2 \\
\qquad\qquad\qquad QPars_1 \\
\qquad\qquad\qquad QPars_2 \\
\qquad\qquad\qquad \bullet PAct\ [\![\ \alpha(P\_St) \mid usedC(PAct) \cap usedC(QAct) \mid \alpha(Q\_St)\ ]\!]\ QAct\ \textbf{\textit{end}} \\
CPars_1\ (P\_Q) \\
CPars_2\ (P\_Q) \\
CPars_3\ (P\_Q)
\end{array}
$$

provided $P \notin \alpha(Q) \cup \bigcup_i \alpha(CPars_i\ (P\_Q));\ Q \notin \alpha(P) \cup \bigcup_i \alpha(CPars_i\ (P\_Q));$
$\alpha(P\_St) \cap \alpha(Q\_St) = \varnothing;\ (\alpha(PPars_1) \cup \alpha(PPars_2)) \cap (\alpha(QPars_1) \cup \alpha(QPars_2)) = \varnothing.$

Law 19 (*par-comm*)**.** $A_1\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A_2 = A_2\ [\![\ ns_2 \mid cs \mid ns_1\ ]\!]\ A_1.$

Law 20 (*par-hid-dist*)**.**

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

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

Law 21 (*par-inter*)**.** $A_1\ [\![\ ns_1 \mid cs \mid ns_2\ ]\!]\ A_2 = A_1\ [\![\ ns_1 \mid ns_2\ ]\!]\ A_2.$
provided $usedC(A_1, A_2) \cap cs = \varnothing.$

Law 22 (*par-inter-2*)**.** $A_1\ [\![\ ns_2 \mid ns_2\ ]\!]\ A_2 = A_1\ [\![\ ns_2 \mid \varnothing \mid ns_2\ ]\!]\ A_2.$

Law 23 (*par-out-inp-inter-exchange*)**.**

$$
\begin{array}{l}
(A_1;\ c_1 \longrightarrow \textbf{\textit{Skip}})\ [\![\ ns_1 \mid \{\!|\ c_1\ |\!\}\ \mid ns_2\ ]\!]\ ((c_1 \longrightarrow A_2\ [\![\ ns_3 \mid ns_4\ ]\!]\ c_2 \longrightarrow A_3);\ A_4) \\
= \\
(A_1;\ c_1 \longrightarrow A_2\ [\![\ ns_1 \cup ns_3 \mid ns_4\ ]\!]\ c_2 \longrightarrow A_3);\ A_4
\end{array}
$$

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

Law 24 (*par-out-inp-inter-exchange-n*).

$$
\begin{array}{l}
(B_1 \mathbin{;} (\Big|\!\Big|\!\Big| i : 1\,..\,n \bullet [\![ns_i]\!] (c_i!v_i \longrightarrow \textbf{\textit{Skip}})) \\
\quad [\![ns_1 \mid \{\!| c_1, \ldots, c_n |\!\} \mid ns_2]\!] \\
(((\Big|\!\Big|\!\Big| i : 1\,..\,n \bullet [\![ns_i]\!] (c_i?x \longrightarrow A_i(x))) [\![ns_3 \mid ns_4]\!] B_2) \mathbin{;} B_3) \\
= \\
(B_1 \mathbin{;} ((\Big|\!\Big|\!\Big| i : 1\,..\,n \bullet [\![ns_i]\!] (c_i!v_i \longrightarrow A_i(v_i))) [\![ns_1 \cup ns_3 \mid ns_4]\!] B_2)) \mathbin{;} B_3
\end{array}
$$

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

Law 25 (*par-seq-step*). $(A_1 \mathbin{;} A_2) [\![ns_1 \mid cs \mid ns_2]\!] A_3 = A_1 \mathbin{;} (A_2 [\![ns_1 \mid cs \mid ns_2]\!] A_3)$
provided $usedC(A_1) = \varnothing$, $usedV(A_3) \cap wrtV(A_1) = \varnothing$; *and* $wrtV(A_1) \subseteq ns_1 \cup ns_1'$.

Law 26 (*par-seq-step-2*).

$$\textbf{\textit{var}}\ d \bullet (A_1 \mathbin{;} A_2) [\![ns_1 \mid cs \mid ns_2]\!] (A_1 \mathbin{;} A_3) = \textbf{\textit{var}}\ d \bullet A_1 \mathbin{;} (A_2 [\![ns_1 \mid cs \mid ns_2]\!] A_3)$$

provided $usedC(A_1) \subseteq cs$; *and* $wrtV(A_1) \subseteq \alpha(d)$.

Law 27 (*prefix-seq-assoc*). $c \longrightarrow (A_1 \mathbin{;} A_2) = (c \longrightarrow A_1) \mathbin{;} A_2$
provided $FV(A_2) \cap \alpha(c) = \varnothing$.

The reference to $\mathbf{L}(\_)$ denotes the fact that declarations of x (and x') in schemas, which were used to put the local variable x of the main action into scope, may now be removed, as x is a state component.

Law 28 (*prom-var-state*).

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

Law 29 (*prom-var-state-2*).

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

Law 30 (*rec-sync*).

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

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

Law 31 (*seq-assoc*). $A_1 \mathbin{;} (A_2 \mathbin{;} A_3) = (A_1 \mathbin{;} A_2) \mathbin{;} A_3$.

Law 32 (*seq-left-unit*). $A = \textbf{\textit{Skip}} \mathbin{;} A$.

Law 33 (*seq-right-unit*). $A = A \mathbin{;} \textbf{\textit{Skip}}$.

Law 34 (*str-post*). $w : [pre, post] \sqsubseteq w : [pre, post']$ **provided** $post' \Rightarrow post$.

Law 35 (*var-exp-par*).

$$(\textbf{\textit{var}}\ d : T \bullet A_1) [\![ns_1 \mid cs \mid ns_2]\!] A_2 = (\textbf{\textit{var}}\ d : T \bullet A_1 [\![ns_1 \mid cs \mid ns_2]\!] A_2)$$

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

Law 36 (*var-exp-par-2*).

$$(\textbf{\textit{var}}\ d \bullet A_1) [\![ns_1 \mid cs \mid ns_2]\!] (\textbf{\textit{var}}\ d \bullet A_2) = (\textbf{\textit{var}}\ d \bullet A_1 [\![ns_1 \mid cs \mid ns_2]\!] A_2).$$

Law 37 (*var-exp-rec*). $\mu X \bullet (\textbf{\textit{var}}\ x : T \bullet F(X)) = \textbf{\textit{var}}\ x : T \bullet (\mu X \bullet F(X))$
provided *x is initialised before use in F*.

Law 38 (*var-exp-seq*). $A_1 \mathbin{;} (\textbf{\textit{var}}\ x : T \bullet A_2) \mathbin{;} A_3 = (\textbf{\textit{var}}\ x : T \bullet A_1 \mathbin{;} A_2 \mathbin{;} A_3)$
provided $\{ x, x' \} \cap (FV(A_1) \cup FV(A_3)) = \varnothing$.

Appendix C. Extra definitions

The function *buildRCells* maps the function *buildRCell* over a list of actions, using the given proof obligations as argument.

$$buildRCells : (\mathrm{seq}_1(Pred \times ActBody) \times PObs) \to \mathrm{seq}\, RCell$$

$$\forall\, gcs : \mathrm{seq}_1(Pred \times ActBody);\ pobs : PObs \bullet buildRCells\,(gcs, pobs) = (buildRCell\, pobs)\ \mathsf{map}\ (extractActions\, gcs)$$

The function *applyTacsGC* takes two lists: the first is a list of tactics and the second is a list of refinement cells. It applies each tactic in the first list to the corresponding refinement cell in the second list. This results in a list of lists of refinement cells. We then use the distributed cartesian product ($\prod_\infty$) to get all the possible combinations as sequences of refinement cells in each of the lists. This yields a flat list of sequences in which each element represents a particular outcome of applying the tactics to the programs of the initial refinement cell.

$$applyTacsGC : (\mathrm{seq}_1\, Tactic) \times (\mathrm{seq}_1\, RCell) \to \mathrm{pfiseq}[\mathrm{seq}\, RCell]$$

$$\forall\, tacs : \mathrm{seq}_1\, Tactic;\ rcs : \mathrm{seq}_1\, RCell \bullet applyTacsGC\,(tacs, rcs) = \prod_\infty (tacs\ \mathsf{mapl}\ rcs)$$

References

[1] R.D. Arthan, P. Caseley, Colin O'Halloran, A. Smith, Clawz: Control Laws in Z, in: ICFEM, 2000, pp. 169–176.
[2] R.J.R. Back, J. von Wright, Refinement concepts formalised in higher order logic, Formal Aspects of Computing 2 (1990) 247–274.
[3] M. Butler, J. Grundy, T. Långbacka, R. Rukš enas, J. von Wright, The refinement calculator: proof support for program refinement, in: L. Groves, S. Reeves (Eds.), Formal Methods Pacific'97: Proceedings of FMP'97, Springer-Verlag, Wellington, New Zealand, 1997, pp. 40–61.
[4] D. Carrington, I. Hayes, R. Nickson, G. Watson, J. Welsh, A program refinement tool, Formal Aspects of Computing 10 (2) (1998) 97–124.
[5] David Carrington, Ian Hayes, Ray Nickson, Geoffrey Watson, Jim Welsh, A review of existing refinement tools, Technical Report 94-8, Software Verification Research Centre, The University of Queensland, 1994.
[6] A.L.C. Cavalcanti, P. Clayton, Verification of control systems using *Circus*, in: Proceedings of the 11th IEEE International Conference on Engineering of Complex Computer Systems, IEEE Computer Society, 2006, pp. 269–278.
[7] A.L.C. Cavalcanti, P. Clayton, C. O'Halloran, Control law diagrams in *Circus*, in: J. Fitzgerald, I.J. Hayes, A. Tarlecki (Eds.), FM 2005: Formal Methods Symposium, in: Lecture Notes in Computer Science, vol. 3582, Springer-Verlag, 2005, pp. 253–268.
[8] A.L.C. Cavalcanti, A.C.A. Sampaio, J.C.P. Woodcock, A refinement strategy for *Circus*, Formal Aspects of Computing 15 (2–3) (2003) 146–181.
[9] A.L.C. Cavalcanti, J.C.P. Woodcock, ZRC—a refinement calculus for Z, Formal Aspects of Computing 10 (3) (1999) 267–289.
[10] David Delahaye, A tactic language for the system coq, in: M. Parigot, A. Voronkov (Eds.), Logic for Programming and Automated Reasoning, in: Lecture Notes in Computer Science, vol. 1955, Springer, Berlin, Heidelberg, 2000, pp. 377–440.
[11] C. Fischer, How to combine Z with a process algebra, in: J.P. Bowen, A. Fett, M.G. Hinchey (Eds.), ZUM'98: The Z Formal Specification Notation, 11th International Conference of Z Users, in: Lecture Notes in Computer Science, vol. 1493, Springer Verlag, 1998, pp. 5–23.
[12] M. Gordon, R. Milner, C. Wadsworth, Edinburgh LCF, in: Lecture Notes in Computer Science, vol. 78, Springer-Verlag, 1979.
[13] L. Groves, R. Nickson, M. Utting, A tactic driven refinement tool, in: C.B. Jones, R.C. Shaw, T. Denvir (Eds.), 5th Refinement Workshop, Workshops in Computing, Springer-Verlag, 1992, pp. 272–297.
[14] J. Grundy, A window inference tool for refinement, in: C.B. Jones, R.C. Shaw, T. Denvir (Eds.), 5th Refinement Workshop, Workshops in Computing, Springer-Verlag, 1992, pp. 230–254.
[15] A.C. Gurgel, C.G. de Castro, M.V.M. Oliveira, Tool support for the *Circus* refinement calculus, in: E. Börger, M. Butler, J.P. Bowen, P. Boca (Eds.), ABZ Conference, in: Lecture Notes in Computer Science, vol. 5238, Springer-Verlag, 2008, p. 349.
[16] C.A.R. Hoare, Communicating Sequential Processes, Prentice-Hall, 1985.
[17] Brian R. Hunt, Ronald L. Lipsman, Jonathan M. Rosenberg, A Guide to MATLAB: For Beginners and Experienced Users, Cambridge University Press, New York, NY, USA, 2001.
[18] F. Kirchner, C. Muñoz, **PVS**#: streamlined tacticals for **PVS**, Electronic Notes in Theoretical Computer Science 174 (11) (2007) 47–58.
[19] Saunders Mac Lane, Categories for the Working Mathematician, in: Graduate Texts in Mathematics, Springer, 1991.
[20] B.P. Mahony, J.S. Dong, Blending object-Z and timed CSP: an introduction to TCOZ, in: K. Torii, K. Futatsugi, R.A. Kemmerer (Eds.), The 20th International Conference on Software Engineering, ICSE'98, IEEE Computer Society Press, 1998, pp. 95–104.
[21] A. Martin, Infinite lists for specifying functional programs in Z, Technical Report, University of Queensland, Queensland, Australia, March 1995.
[22] A.P. Martin, P.H.B. Gardiner, J.C.P. Woodcock, Tactic semantics and reasoning, Technical Report, Oxford University Computer Laboratory, Oxford, December 1993, Draft version.
[23] A.P. Martin, P.H.B. Gardiner, J.C.P. Woodcock, A tactical calculus, Formal Aspects of Computing 8 (4) (1996) 479–489.
[24] Andrew Martin, Ray Nickson, Mark Utting, A tactic language for Ergo, Technical Report 97-16, Software Verification Centre, Department of Computer Science, The University of Queensland, February 1997.
[25] Andrew Martin, Ray Nickson, Mark Utting, Improving Angel's parallel operator: Gumtree's approach, Technical Report 97-15, Software Verification Centre, School of Information Technology, The University of Queensland, December 1997.
[26] C. Morgan, Programming from Specifications, Prentice-Hall, 1994.
[27] C. Morgan, P.H.B. Gardiner, Data refinement by calculation, Acta Informatica 27 (6) (1990) 481–503.
[28] Raymond George Nickson, Tool support for the refinement calculus, Ph.D. Thesis, Victoria University of Wellington, 1994.
[29] M.V.M. Oliveira, Tactics of refinement, Technical report, Centro de Informática, Universidade Federal de Pernambuco, Pernambuco, Brazil, December 2000. Available at http://www.cs.york.ac.uk/~marcel/gabriel/.
[30] M.V.M. Oliveira, ArcAngel: tactics examples and their usage, Technical report, Centro de Informática, Universidade Federal de Pernambuco, Pernambuco, Brazil, December 2002. Availabel at http://www.cs.york.ac.uk/~marcel/gabriel/.
[31] M.V.M. Oliveira, ArcAngel: a tactic language for refinement and its tool support, Master's Thesis, Centro de Informática, Universidade Federal de Pernambuco, Pernambuco, Brazil, 2002. Available at http://www.ufpe.br/sib/.
[32] M.V.M. Oliveira, Formal derivation of state-rich reactive programs using Circus, Ph.D. Thesis, Department of Computer Science, University of York, 2006. YCST-2006/02.
[33] M.V.M. Oliveira, ArcAngel*C*, Technical report, Departamento de Informática e Matemática Aplicada - Universidade Federal do Rio Grande do Norte, Natal, Brazil, February 2007. Available at http://www.dimap.ufrn.br/~marcel/.

[34] M.V.M. Oliveira, A.L.C. Cavalcanti, ArcAngel*C*: a refinement tactic language for *Circus*, Electronic Notes in Theoretical Computer Science 214C (2008) 203–229.
[35] M.V.M. Oliveira, A.L.C. Cavalcanti, J.C.P. Woodcock, ArcAngel: a tactic language for refinement, Formal Aspects of Computing 15 (1) (2003) 28–47.
[36] M.V.M. Oliveira, A.L.C. Cavalcanti, J.C.P. Woodcock, A UTP semantics for *Circus*, Formal Aspects of Computing (2008) doi:10.1007/s00165-007-0052-5.
[37] M.V.M. Oliveira, A.C. Gurgel, C.G. de Castro, CRefine: support for the *Circus* refinement calculus, in: Antonio Cerone, Stefan Gruner (Eds.), 6th IEEE International Conferences on Software Engineering and Formal Methods, IEEE Computer Society Press, 2008, pp. 281–290.
[38] M.V.M. Oliveira, M. Xavier, A.L.C. Cavalcanti, Refine and Gabriel: support for refinement and tactics, in: Jorge R. Cuellar, Zhiming Liu (Eds.), 2nd IEEE International Conference on Software Engineering and Formal Methods, IEEE Computer Society Press, 2004, pp. 310–319.
[39] Sam Owre, N. Shankar, Writing PVS proof strategies, in: Myla Archer, Ben Di Vito, César Muñoz (Eds.), Design and Application of Strategies/Tactics in Higher Order Logics, STRATA 2003, number CP-2003-212448 in NASA Conference Publication, NASA Langley Research Center, Hampton, VA, September 2003, pp. 1–15.
[40] Lawrence C. Paulson, Logic and Computation: Interactive Proof with Cambridge LCF, Cambridge University Press, New York, NY, USA, 1987.
[41] ProofPower. Available at http://www.lemma-one.com/ProofPower/index/index.html.
[42] A.C.A. Sampaio, J.C.P. Woodcock, A.L.C. Cavalcanti, Refinement in *Circus*, in: L. Eriksson, P.A. Lindsay (Eds.), FME 2002: Formal Methods—Getting IT Right, in: Lecture Notes in Computer Science, vol. 2391, Springer-Verlag, 2002, pp. 451–470.
[43] J.M. Spivey, The Z Notation: A Reference Manual, 2nd edition, Prentice-Hall, 1992.
[44] H. Treharne, S. Schneider, Using a process algebra to control B operations, in: K. Araki, A. Galloway, K. Taguchi (Eds.), Proceedings of the 1st International Conference on Integrated Formal Methods, Springer, 1999, pp. 437–456.
[45] J.L.A. van de Snepscheut, Mechanised support for stepwise refinement, in: Jürg Gutknecht (Ed.), Programming Languages and System Archtectures, in: Lecture Notes in Computer Science, vol. 782, Springer, Zurich, Switzerland, 1994, pp. 35–48.
[46] T. Vickers, An overview of a refinement editor, in: 5th Australian Software Engineering Conference, Sydney, Australia, May 1990, pp. 39–44.
[47] T. Vickers, A language of refinements, Technical Report TR-CS-94-05, Computer Science Department, Australian National University, 1994.
[48] J. von Wright, J. Hekanaho, P. Luostarinen, T. Långbacka, Mechanizing some advanced refinement concepts, Formal Methods in System Design 3 (1993) 49–81.
[49] J.C.P. Woodcock, J. Davies, Using Z—Specification, Refinement, and Proof, Prentice-Hall, 1996.
[50] M.A. Xavier, A.L.C. Cavalcanti, A.C.A. Sampaio, Type Checking *Circus* Specifications, in: A.M. Moreira, L. Ribeiro (Eds.), SBMF 2006: Brazilian Symposium on Formal Methods, 2006, pp. 105–120.
[51] M. Saaltink, The Z/EVES System, in: J.P. Bowen, M.G. Hinchey, D. Till (Eds.), ZUM'97: The Z Formal Specification Notation, in: Lecture Notes in Computer Science, vol. 1212, Springer-Verlag, 1997, pp. 72–85.
[52] F. Zeyda, M.V.M. Oliveira, A.L.C. Cavalcanti, Supporting ArcAngelin ProofPower, in: J. Derrick, E. Boiten (Eds.), REFINE 2009, in: Eletronic Notes in Theoretical Computer Science, vol. 259, Elsevier, 2009, pp. 225–243.