Refinement of Actions in Circus

Ana Cavalcanti\textsuperscript{a} Augusto Sampaio\textsuperscript{b} Jim Woodcock\textsuperscript{a}

\textsuperscript{a} Computing Laboratory
University of Kent
Canterbury, England

\textsuperscript{b} Centro de Informática
Universidade Federal de Pernambuco
Recife – PE, Brazil

Abstract
This paper presents refinement laws to support the development of actions in Circus, a combination of Z and CSP adequate to specify the data structures and behavioural aspects of concurrent systems. In this language, systems are characterised as a set of processes; each process is a unit that encapsulates state and reactive behaviour defined by actions. Previously, we have addressed the issue of refining processes. Here, we are concerned with the actions that compose the behaviour of such processes, and that may involve both Z and CSP constructs. We present a number of useful laws, and a case study that illustrates their application.

1 Introduction

Like in other branches of science, specification and programming theories have evolved on the basis of discovering new concepts and paradigms, as well as extending and integrating existing and consolidated ones. The several research efforts dedicated to the field give evidence that combining and unifying theories is a promising direction to make progress. The unifying theories of programming\cite{8} is a tempting invitation to join in.

Circus\cite{14}, an integration of the CSP\cite{7,12} process algebra, and the Z\cite{16} model-based specification language, has been conceived as a unified theory. A Circus program is structured in terms of processes. Each process has a state, described as a Z schema, and a control behaviour expressed by an action in a CSP style. As an action typically embodies state change, Circus allows schema expressions and Dijkstra’s guarded commands\cite{4} as actions, in addition to the usual CSP operators.

A refinement theory for Circus must clearly address both CSP and Z refinement. The refinement of actions in Circus needs to take into account both the CSP notation and the Z operations described as schema expressions. Larger
grain, process level refinement laws are also necessary to allow direct transformations involving process units.

In a related paper [13], we have proposed a strategy for developing distributed Circus applications from centralised specifications. This strategy is backed upon notions of action, process, and data (forwards simulation) refinement, as well as process level refinement laws. The focus of that work is on larger grain transformations, emphasising how changes in the overall structure of the process can be guaranteed to preserve the semantics.

This paper contributes to a systematic development strategy for Circus based on formal refinement, addressing all the language constructs. It complements [13] by proposing laws of actions, including the laws of CSP [12] and of ZRC [3], a refinement calculus for Z. In addition, new laws are necessary to deal with Z schemas as actions.

For example, laws that introduce parallel composition and external choice from Z schema expressions are an entirely new contribution of this work. Even some laws which relate only CSP operators can also be regarded as original contributions, as there seems to be no comprehensive set of refinement laws to support program development in CSP. Furthermore, we extend the forwards simulation laws proposed in [13] to address all the action operators of Circus.

We also illustrate how these laws can be proved from the semantics of Circus. Parts of the development of a distributed, cached-head, ring buffer from a centralised specification are used to illustrate the laws of actions and of forwards simulation.

In the next section, we introduce Circus through an example: a buffer specification that is taken as a reference for the refinements presented in Section 6. In order to make the paper self contained, the refinement notions reported in [13] are summarised in Section 3. A sample of the refinement laws of actions, with emphasis on those for forwards simulation and those that relate Z schemas to CSP operators, is the subject of Sections 4 and 5. The final section summarises the results obtained and discusses related and future work. An appendix gives the list of all laws used in our examples; another one illustrates how our laws can be proved.

2 Circus

A Circus program contains a list of paragraphs that consist of process definitions and their ancillary declarations of channels, types, and global constants. We illustrate such a program in this paper by considering a bounded, reactive, FIFO buffer that is used to store natural numbers. The maximum size of this buffer is a strictly positive constant.

\[
\mid\text{maxbuff} : \mathbb{N}_1
\]

The buffer program takes its inputs and supplies its outputs on two different
typed channels.

\textbf{channel} input, output : \mathbb{N}

A process encapsulates data, and in \textit{Circus} the state is described using the Z notation. The attendant operations are actions, which are specified using Z, CSP operators, and the guarded command language. An unnamed main action at the end of a process description defines its extensional behaviour as a protocol in terms of the actions over the state.

In our example, the process \textit{Buffer} encapsulates two state components: an ordered list of contents and the size of this list.

\textbf{process} Buffer \triangleq \textbf{begin}

\begin{verbatim}
BufferState
buf : seq \mathbb{N}
size : 0 .. \text{maxbuff}

size = \#buf \leq \text{maxbuff}
\end{verbatim}

Initially, the buffer is empty; this is specified as a state initialisation action.

\textit{BufferInit} \triangleq [ BufferState' | buf' = < > \land size' = 0 ]

We need to describe the process’s behaviour on input and output. The buffer accepts an input whenever there is space to store the new value; in this case, the element input is appended to the bounded sequence and the size incremented. We specify this action in two parts, the first, \textit{InputCmd}, describing the action’s effect on the state, and the other, \textit{Input}, describing the constraint on the communication.

\begin{verbatim}
InputCmd
\Delta BufferState
x? : \mathbb{N}

size < \text{maxbuff}

buf' = buf \uplus \{ x? \} \land size' = size + 1
\end{verbatim}

The effect on the state is described by a schema as usual.

\textit{Input} \triangleq size < \text{maxbuff} \& \text{input}?x \rightarrow \textit{InputCmd}

The \textit{Input} action is guarded by \textit{size} < \textit{maxbuff}: if this condition does not hold, \textit{Input} deadlocks. In contrast, if a precondition of a schema action is not satisfied, its execution diverges (aborts), as usual in Z.

The action \textit{input}?x \rightarrow \textit{InputCmd} is a prefixing in the style of CSP. A new input variable \texttt{x} is introduced, and a value input through the channel \texttt{input}.
is assigned to it. Afterwards, the action `InputCmd` is executed; this is the schema action defined above that actually modifies the state.

The `Output` action is enabled, providing the buffer contains something; it outputs the head of the buffer, and updates the size accordingly.

\[
\begin{align*}
\text{OutputCmd} & : \Delta \text{BufferState} \\
size > 0 & \\
\text{buff}' = \text{tail buff} \land size' = size - 1
\end{align*}
\]

\[\text{Output} \triangleq size > 0 \& \text{output!(head buff)} \rightarrow \text{OutputCmd}\]

Finally, the main action initialises the `Buffer` and repeatedly offers the choice of input and output.

\[\bullet \text{BufferInit; } (\mu X \bullet (\text{Input } \Box \text{Output}); X)\]

end

We summarise the complete specification of the buffer process and its declarations in Figure 1.

The example has shown how processes are constructed from actions, but processes may themselves be combined with CSP operators, such as parallel composition. The meaning of a new process constructed in this way is obtained from the conjunction of the constituent process’s states and the appropriate combination of their main actions.

At the level of actions, the `Circus` parallel operator is actually slightly different from that of CSP. Instead of having simply a synchronisation channel set, we also have two sets that partition all the variables in scope: the state components, and the input and local variables. For instance, we write \(A_1 \parallel [v_{s1} \mid C \mid v_{s2}] A_2\) for the parallel composition of \(A_1\) and \(A_2\) synchronising on the channels in the set \(C\). Moreover, \(A_1\) can modify only the variables in \(v_{s1}\) and, similarly, \(A_2\) can modify just the variables in \(v_{s2}\). On the other hand, both \(A_1\) and \(A_2\) have access to the initial value of the variables in \(v_{s1}\) and \(v_{s2}\).

The semantics of `Circus` [15] is based on unifying theories of programming [8]: an alphabetised relational model for imperative programming, concurrency, and communication. We use \(Z\) as the concrete syntax for the relational model, so that a `Circus` program actually denotes a \(Z\) specification. Each process corresponds to a part of that specification characterised by a state definition; in this model, actions are simply operations over this state.

In the unifying theory, distinguished auxiliary variables are used to describe observations relevant to the behaviour being formalised. In the semantics of `Circus`, these variables are included in the state components of a process’s denotation. In addition to the state components in the process’s specification,
there are components to model behaviour: stability from divergence (okay), termination (wait), a history of interaction with the environment (tr), and a set of events that can be refused on the next step (ref). This model is adequate to describe state-based, failures-divergences behaviour, with embedded imperative features: Circus processes.

To illustrate the approach of the unifying theory, consider the semantics of the simple-prefixing operator in the definition $P \triangleq a \to Skip$; we explain $P$'s behaviour by case analysis on the observational variables okay and wait.

First, suppose that okay is false; in this case, $P$ has been activated in the final state of a preceding process that is diverging. Since divergence is a left-zero for sequential composition, the only thing that $P$ can possibly guarantee is that it leaves the final value of the trace of interaction with the environment as an extension of its initial value: $tr \ prefix tr'$.

Suppose next that okay is true and so $P$’s predecessor is not diverging. There are now two further cases to consider: the predecessor may or may not have terminated; the distinction between these cases is described by the observational variable wait. Suppose that wait is true and so the predecessor has not terminated; then $P$ must have no effect on the observations.

Suppose instead that wait is false and so the predecessor has terminated. There are now two possible states: $P$ itself may or may not have terminated. Suppose that wait', $P$’s final value of this observation, is true and so $P$ has
not terminated. Since \( P \) can do only one event, it obviously has not occurred, so the trace \( tr \) is unchanged, but \( P \) must not be refusing to engage in the event \( a: tr' = tr \wedge a \notin ref' \).

Finally, suppose that \( wait' \) is false and so \( P \) has been observed to have terminated. \( P \) must have added the event \( a \) to the trace: \( tr' = tr \wedge \langle a \rangle \). The final value of the refusal set is irrelevant, since \( P \) has now terminated and can do nothing further. In all the cases that we have discussed where \( okay \) is true, \( P \) leaves the state variables unchanged and does not diverge. In summary, the semantics of \( P \hat{=} a \rightarrow Skip \) is defined as follows.

\[
\begin{align*}
& (\neg okay \wedge tr \prefix tr') \lor \\
& (okay \wedge okay' \wedge v' = v \wedge \\
& ((wait \wedge wait' \wedge tr' = tr \wedge ref' = ref) \lor \\
& (\neg wait \wedge ((wait' \wedge tr' = tr \wedge a \notin ref') \lor \\
& (\neg wait' \wedge tr' = tr \wedge \langle a \rangle)))
\end{align*}
\]

We assume that \( v \) is the (list of) state variable(s).

### 3 Refinement notions

If an implementation is to behave satisfactorily, then every observation that we make of it must be permitted by an agreed specification. So, refinement is simply implication: a process \( P \) satisfies a specification \( S \), providing that \([P \Rightarrow S]\), where the square brackets denote universal quantification over all observations and state variables, which must be the same for both \( P \) and \( Q \).

**Definition 3.1** [Action refinement] Suppose that \( A_1 \) and \( A_2 \) are actions on the same state space. Action \( A_1 \) is refined by action \( A_2 \) if, and only if, every observation of \( A_2 \) is permitted by \( A_1 \) as well: \( A_1 \sqsubseteq_A A_2 \iff [A_2 \Rightarrow A_1] \). \( \square \)

Because encapsulation means that the state of a process is private, we may change its components and their types during refinement [10]. Process refinement (\( \sqsubseteq_P \)) is defined in terms of action refinement of local blocks, since when we hide the local states of processes we are left with two main actions over the same alphabet. In the following, let \( P.st \) and \( P.act \) denote the local state and main action of a process \( P \), respectively.

**Definition 3.2** [Process refinement] We define \( P \sqsubseteq_P Q \) to mean that process \( P \) is refined by process \( Q \) if, and only if,

\[
(\exists P.st; P.st' \bullet P.act) \sqsubseteq_A (\exists Q.st; Q.st' \bullet Q.act)
\]

\( \square \)

Forwards and backwards simulation are well-known techniques for proving the correctness of developments involving local blocks [6,5,16], and a well-established result is that both techniques are needed for completeness. In this paper, we deal only with the technique of forwards simulation.
Definition 3.3 [Forwards simulation] A forwards simulation between actions $A$ and $B$ of processes $P$ and $Q$, with local state $L$, is a relation $R$ satisfying

1. (feasibility) $\forall Q. st \bullet (\exists P. st \bullet R)$

2. (correctness) $\forall P. st; Q. st; Q. st' \bullet R \land B \Rightarrow (\exists P. st' \bullet R' \land A)$

In this case, we write $A \preceq_{P, Q, R, L} B$ and say that the action $B$ simulates the action $A$ according to the simulation $R$ and in a state extended by $L$. When clear from the context, we omit the subscripts. A forwards simulation between $P$ and $Q$ is a forwards simulation between their main actions. The local state $L$ includes input and local variables in scope for $A$ and $B$. $\square$

In Definition 3.3, there is no applicability requirement concerning preconditions, as would usually be found in the definition of forwards simulation; this is because the semantics of actions are total. Furthermore, we do not impose any specific conditions on the initialisation, as it is not necessarily the case that there will be a separate initialisation action: initialisation may be part of the main action or there might not be an explicit initialisation at all.

The next theorem ensures that, if we provide a forwards simulation between processes $P$ and $Q$, then we can substitute $Q$ for $P$ in a program.

Theorem 3.4 (Forwards simulation is sound) When a forwards simulation exists between two processes $P$ and $Q$, we also have that $P \sqsubseteq_P Q$. $\square$

A proof for this theorem can be found in [13], where we present a full exposition of the ideas described in this section.

The laws in the next section provide support to prove that a relation $R$ is indeed a forwards simulation. Using those laws, we can justify proving simulation for schema actions, in much the same way as we do in Z, and for the other primitive actions, and keeping the structure of the actions of the original process $P$ in the definition of the new process $Q$.

4 Forward simulation of actions

The primitive action $\text{Skip}$, $\text{Stop}$, and $\text{Chaos}$ are not affected by forwards simulation. For instance $\text{Skip} \preceq \text{Skip}$, for any $P$, $Q$, $R$, and $L$, which we omit. For schema actions, the provisos are those in the standard data refinement rule for Z, which is a rather pleasing result in terms of using well-established techniques.

Law 4.1 (Schema Expressions)

$\text{ASExp} \preceq \text{CExp}$

provided

$\forall P. st; Q. st \bullet R \land \text{pre AExp} \Rightarrow \text{pre CExp}$

$\forall P. st; Q. st; Q. st' \bullet R \land \text{pre AExp} \land \text{CExp} \Rightarrow$

$(\exists P. st' \bullet R' \land \text{ASExp})$

$\square$

We refrain from presenting the particular case of initialisation operations and
functional data refinement, because the usual results follow, since the more
generic rule present above holds.

As already mentioned, forwards simulation distributes through the other
constructs. Below, we present the rule for an input prefix.

**Law 4.2 (Input prefix distribution)**

\[ c?x \rightarrow A \preceq c?x \rightarrow B \]

provided \( A \preceq B \)

For output prefixing, we need to relate the abstract and the concrete expres-
sions that define the output value. The relevant law is as follows.

**Law 4.3 (Output prefix distribution)**

\[ c!ae \rightarrow A \preceq c!ce \rightarrow B \]

provided

\[ \forall P.st; Q.st; L \bullet R : (ae = ce) \]
\[ A \preceq B \]

The concrete and abstract values have to be equal, with respect to the retrieve
relation. The local state \( L \) includes any additional information inferred from
guards and conditionals in context. More specifically, if \( A \) and \( B \) occur in
an action guarded by \( g \), as in \( g & A \), for instance, then \( L \) includes \( g \) in its
predicate part. (If there are no input and local variables in scope, then \( L \) is a
schema with an empty declaration part and predicate \( g \).)

For guarded actions, we also need to relate the abstract and the concrete
predicates that define the guard.

**Law 4.4 (Guard distribution)**

\[ ag & A \preceq cg & B \]

provided

\[ \forall P.st; Q.st; L \bullet R : (ag \leftrightarrow cg) \]
\[ A \preceq B \]

The proviso is similar to that of Law 4.3.

For the other constructs, mostly we have straightforward distribution laws.
An example is given below for external choice.

**Law 4.5 (External choice distribution)**

\[ A_1 \Box A_2 \preceq B_1 \Box B_2 \]

provided

\[ A_1 \preceq B_1 \]
\[ A_2 \preceq B_2 \]

For parallelism, the issues are discussed in [13]. We do not use that result in
this paper.

For recursion, we also have a simple result.

**Law 4.6 (Recursion distribution)**

\[ \mu X \cdot F(A) \preceq \mu X \cdot F(B) \]

provided \( A \preceq B \)

It is based on this set of laws, that we can conclude that forwards simulation distributes through the structure of arbitrary actions.

5 Algorithmic refinement of actions

Apart from data refining processes, we are also interested in transforming, or algorithmically refining actions. The result below, originally presented in [13], justifies the use of action refinement (Definition 3.1), provided the usual initialisation and applicability theorems hold for the process.

**Theorem 5.1 (Feasible refinement)** Suppose we have a process \( P \) with actions \( A \) and \( B \). If \( A \sqsubseteq_A B \), then the identity is a forwards simulation between \( A \) and \( B \).

With this result, laws of both CSP and Z, for which we have a refinement calculus [3], are relevant. We concentrate here, however, on the laws that relate Z and CSP constructs, which are novel. All the extra laws needed for our case study in section 5 are presented in Appendix A. The first law we present introduces a choice of guarded actions from a schema expression \( SExp \).

**Law 5.1 (Guard Introduction — Schema Expression)**

\[ SExp \]

\[ \sqsubseteq \]

\[ \Box i \cdot g_i \& SExp \land [State \mid g_i] \]

provided \( \text{pre} \ SExp \Rightarrow \bigvee i \cdot g_i \)

The proviso guarantees that if the precondition of \( SExp \) holds, then at least one of the guards is enabled. In this case, an action associated with an enabled guard \( g_i \) is arbitrarily chosen. The behaviour of this action is given by \( SExp \) itself, and so we know that the original behaviour is attained; however, we conjoin \( SExp \) with a schema that records that \( g_i \) holds. This may be useful in further refining \( SExp \). If the precondition of \( SExp \) does not hold, \( SExp \) diverges and \( \Box i \cdot g_i \& SExp \land [State \mid g_i] \) may block. Therefore, we have a
refinement as well.

If a schema action is expressed as a disjunction, and it is guarded by the precondition of one of the disjuncts, then the other disjuncts can be eliminated.

**Law 5.2 (Schema Disjunction Elimination)**

\[
\text{pre } SExp_1 \& (SExp_1 \lor SExp_2) \\
\subseteq \\
\text{pre } SExp_1 \& SExp_1
\]

This law is clearly a refinement because, in general, \(SExp_1 \lor SExp_2\) allows more nondeterminism than its disjunct \(SExp_1\).

The following law regards the introduction of a sequential composition, from a schema expression. In the provisos, we use new notation. The function \(\alpha\) gives the set of components of a given schema; it can also be applied to a declaration. The function \(FV\) defines the set of free variables of a predicate or expression; \(DFV\) determines the set of dashed free variables of a given predicate; finally, \(UDFV\) gives the set of undashed free variables of a predicate.

**Law 5.3 (Sequence Introduction — Schema Expression)**

\[
[\Delta S_1; \Delta S_2; i? : T | \text{pre}S_1 \& \text{pre}S_2 \& CS_1 \& CS_2] \\
= \\
[\Delta S_1; \Xi S_2; i? : T | \text{pre}S_1 \& CS_1]; [\Xi S_1; \Delta S_2; i? : T | \text{pre}S_2 \& CS_2]
\]

syntactic restrictions

- \(\alpha(S_1) \cap \alpha(S_2) = \emptyset\);
- \(FV(\text{pre}S_1) \subseteq \alpha(S_1) \cup \{i?\}\);
- \(FV(\text{pre}S_2) \subseteq \alpha(S_2) \cup \{i?\}\);
- \(DFV(CS_1) \subseteq \alpha(S_1')\);
- \(DFV(CS_2) \subseteq \alpha(S_2')\);
- \(UDFV(CS_2) \cap DFV(CS_1) = \emptyset\). 

This law applies to a schema action over a state composed of two disjoint sets of components specified in the state schemas \(S_1\) and \(S_2\). The precondition of the action can be expressed as the conjunction of conditions \(\text{pre}S_1\) and \(\text{pre}S_2\) over the different parts of the state and the input variable(s). Also, the updates on the state are also expressed as a conjunction of conditions \(CS_1\) and \(CS_2\) over the final values of the disjoint parts of the state.

The application of this law introduces a sequential composition of schema actions that update the disjoint parts of the state separately. An extra restric-
tion is required: the final value of the state components of \( S_2 \) do not depend on the initial values of those of \( S_1 \), as these are potentially changed by the first action in the sequence.

There are no output variables. If we include them, we have to distinguish their specification and determine which action in the sequence is going to produce the output.

The next law is concerned with the introduction of parallelism, again from a schema expression. It may seem slightly artificial to introduce communication between schema actions. We must have in mind, however, that these laws are used in a stepwise development, where the schemas are further developed and processes are split. So, the introduction of communication is an interesting step towards a more elaborate structure. This point is illustrated in examples, in the next section.

**Law 5.4 (Parallelism Introduction — Schema Expression)**

\[
[\Delta S_1; \Delta S_2; i? : T | CS_1(i?, s_2) \land CS_2] = c?j? : T; j? : T; s? : U | CS_1(j?, s?)
\]

\[
\begin{align*}
&\llbracket\alpha(S_1) \mid \llbracket c\rrbracket \mid \alpha(S_2)\rrbracket \\
&c!l}s_2 \rightarrow [\Delta S_1; \Delta S_2 | CS_2]) \setminus \{\llbracket c\rrbracket\}
\end{align*}
\]

**syntactic restrictions**

- \( \alpha(S_1) \cap \alpha(S_2) = \emptyset \);
- \( s_2 \in \alpha(S_2) \) and \( s_2 \) has type \( U \);
- \( FV(CS_1) \subseteq \alpha(\Delta S_1) \cup \{i?, s_2\} \);
- \( FV(CS_2) \subseteq \alpha(\Delta S_2) \);
- \( c \) is a valid channel of type \( T \times U \).

Like Law 5.3, this law applies to a schema action over states composed by the conjunction of state schemas \( S_1 \) and \( S_2 \) with disjoint sets of components. This action takes an input \( i? \) used to update the state \( S_1 \), but not \( S_2 \). The updates of the state are given by the conjunction of \( CS_1 \) and \( CS_2 \); the former defines the updates on \( S_1 \) and the latter, those on \( S_2 \). The updates on \( S_1 \) depend on the component \( s_2 \) of \( S_2 \), but the updates on \( S_2 \) do not depend on \( S_1 \).

With the application of this law, we introduce a parallel composition, in which the disjoint parts of the state are updated separately by different schema actions, each restricted to the relevant part of the state. The hidden channel \( c \) is used to communicate the input value from one action to another, as well as the state component \( s_2 \) that \( CS_1 \) uses from \( S_2 \). The channel \( c \), of course, needs to have been previously declared and have the appropriate type.

To better understand this law, we must note that \( i \) is necessarily an input variable in scope, as otherwise the original schema action is not well-formed. The value of this variable and of \( s_2 \) is sent by the second action to the first one.
using channel \( c \). This communication introduces new input variables \( j \) and \( s \), that are used by the first action, instead of \( i? \) and \( s_2 \). Since the second action does not make use of \( i? \), it is not in its declaration part.

A proof for a simplified version of this law is presented in the Appendix B. This proof is based on the unifying theory model of Circus.

We present here two laws that transform a sequence into a parallelism. The interesting point about them is that we have to make sure the transformation does not affect either state transformations or communications.

In these laws we use some new notation. The function \( \text{wrtV} \) gives the set of variables written by a given action. The function \( \text{usedV} \) gives the set of used variables: read, but not written.

We observe that in the case of a schema expression, \( \text{wrtV} \) actually gives the set of variables constrained by the schema. The definition is as follows.

\[
\text{wrtV}(SExp) = \{ v' : \text{DFV}(SExp) \mid SExp \neq (\exists v' : T \bullet SExp) \land [v' : T] \bullet v \}
\]

We include the variables \( v \) such that, if we hide \( v' \) in \( SExp \) and include it back in the signature, we obtain a different schema. This means exactly that the final value of \( v \) is not arbitrary; it is constrained by \( SExp \).

Strictly speaking, all variables are potentially written by a schema action, because the presence of the (dashed version of the) state component in itself indicates that it can be updated. Even if it is included in a \( \Xi \)-schema, after expansion, all we have is a declaration part including all state components. The above view, however, is more useful in laws.

It is unfortunate, however, that \( \text{wrtV} \) is not a purely syntactic function, as its calculation for schema expressions involves theorem proving. On the other hand, a tool can take the pragmatic approach of considering the whole of the state components as the set of written variables of a schema action, and request help from the user only if this worst view approach fails.

The function \( \text{usedC} \) gives the set of channels referred in a given action.

**Law 5.5 (Parallelism Introduction — Sequential Composition 1)**

\[
A_1; A_2(e) \]

\[
\square ((A_1; c!e \to \text{Skip}) \parallel \text{wrtV}(A_2) \mid \{c\} \mid \text{wrtV}(A_2) \parallel c?y \to A_2(y)) \setminus \{c\}
\]

**syntactic restrictions**
- \( c \) is a valid channel of type \( T \);
- \( c \notin \text{usedC}(A_1) \cup \text{usedC}(A_2) \);
- \( y \notin \text{FV}(A_2) \).

**provided**
- \( \text{wrtV}(A_1) \cap \text{usedV}(A_2) = \emptyset \);
- \( \text{FV}(e) \cap \text{wrtV}(A_2 \text{ before } e) = \emptyset \).

This is our first law to transform a sequence into a parallelism. To preserve the
execution order, it introduces a communication through a new hidden channel $c$ with a new input variable $y$. The type of $c$ has to be compatible with that of the communicated value $e$. The communication removes direct access of $A_2$ to the expression $e$. Even though the order of execution is preserved, in a parallelism both actions have access to the initial value of the variables. Therefore, a proviso is needed: the variables changed by the first action are not used by the second one.

This is a refinement law, not an equality. To understand the reason, suppose $A_1$ is a schema action that leaves the value of a state component $v$ unconstrained, and that $A_2$ uses this value. In the sequence, $A_2$ uses the arbitrary value of $v$, and in the parallelism, $A_2$ takes the initial value of $v$.

The sets of partition variables is defined as $\text{wrt}V(A_2)$ and $\text{wrt}V(A_2)$, respectively, where the first is the set of all state components that are not written by $A_2$. We observe that $\text{wrt}V(A_1)$ and $\text{wrt}V(A_2)$ is not adequate, as we need to ensure coverage of the whole set of state components. Also, $\text{wrt}V(A_1)$ and $\text{wrt}V(A_2)$ is not appropriate because $A_2$ has to be given priority to change the value of the variables it modifies. As an example, suppose $A_1$ changes a state component $x$ to 1, and $A_2$ changes it to 2; the final value of $x$ has to be 2. It is important to notice that the proviso guarantees that $A_2$ does not use the variable $x$, but it may change it.

Finally, we need to guarantee that the value of $e$ is not changed by $A_2$ before it is actually used. The function $\text{before}$ gives an action that captures the behaviour of its action argument before it has to evaluate the given expression. A worst case view is taken in the definition of $\text{before}$, which is by recursion on the structure of actions.

For example, using the notation $SExp[e]$ to represent the fact that the expression $e$ occurs in the (predicate part of the) schema expression $SExp$, we have the definition below. We use similar notation for expressions, predicates, actions in general, and others.

$$SExp[e] \quad \text{before} \quad e = \text{Skip}$$

If the expression $e$ occurs in the schema, the resulting action is $\text{Skip}$: the action that occurs before $e$ has to be evaluated is $\text{Skip}$.

In general, for any action $A$, if $e$ does not occur in it, the result is that action itself.

$$A \quad \text{before} \quad e = A,$$

if $e$ does not occur in $A$

This covers the actions $\text{Skip}$, $\text{Stop}$, and $\text{Chaos}$, for example.

For prefixing, the definition is as follows.

$$(c?x \rightarrow A) \quad \text{before} \quad e = c?x \rightarrow A,$$

if $x \in FV(e)$

$$(c?x \rightarrow A) \quad \text{before} \quad e = c?x \rightarrow (A \quad \text{before} \quad e),$$

otherwise

If a free variable of $e$ is reintroduced locally as an input variable, $e$ cannot occur in its scope, which we keep. Otherwise, we proceed to the prefixed action $A$.

The definition for the other constructs is not very illuminating. Perhaps,
except for sequence.

\((A_1[e]; A_2) \text{ before } e = A_1[e] \text{ before } e\)

\((A_1; A_2) \text{ before } e = A_1; (A_2 \text{ before } e), \text{ if } e \text{ does not occur in } A_1\)

If \(e\) occurs in the first action, the second one does not need to be considered.

Typically, with the application of Law 5.5 we want to avoid direct access of \(A_2\) to a state component. So, we consider the particular case in which \(e\) is a variable. However, the generality above is necessary because we work with structured variables, like sequences, and we do not want to communicate the entire component.

If we have just a variable, we can have less restrictive provisos.

**Law 5.6 (Parallelism Introduction — Sequential Composition 2)**

\[ A_1(x); A_2(x) \]

\[ \sqsubseteq ((A_1(x); c!x \rightarrow \text{Skip}) \]

\[ \left[ \text{wrt} V(A_2) \mid \{ c \} \mid \text{wrt} V(A_2) \right] \]

\[ (c?y \rightarrow A_2(y))) \setminus \{ c \} \]

**syntactic restrictions**

- \(c\) is a valid channel of type \(T\);
- \(c \notin \text{used} C(A_1) \cup \text{used} C(A_2)\);
- \(y \notin \text{FV}(A_2)\).

**provided** \(\text{wrt} V(A_1) \cap \text{used} V(A_2) = \{ x \}\)

Since the access of \(A_2\) to \(x\) is completely removed, and the local input variable \(y\) is used instead, we do not need to worry about the value of \(x\) being changed in \(A_2\) before it is used, as in Law 5.5. Simply, \(x\) is not used or changed.

If all communications between two actions occur sequentially through channels \(c_1\) and \(c_2\), we can use only one channel \(c_3\) to communicate the same values.

**Law 5.7 (Channel Combination)**

\[ (A_1[c_1.\text{com}_1 \rightarrow c_2.\text{com}_2 \rightarrow B_1]) \]

\[ \left[ \text{vs}_1 \mid \{ c_1, c_2 \} \mid \text{vs}_2 \right] \]

\[ A_2[c_1.\text{com}_3 \rightarrow c_2.\text{com}_4 \rightarrow B_2]) \setminus \{ c_1, c_2 \} \]

\[ = (A_1[c_3.\text{com}_1, \text{com}_2 \rightarrow B_1]) \]

\[ \left[ \text{vs}_1 \mid \{ c_3 \} \mid \text{vs}_2 \right] \]

\[ A_2[c_3.\text{com}_3, \text{com}_4 \rightarrow B_2]) \setminus \{ c_3 \} \]

**syntactic restrictions**

- all occurrences of \(c_1\) and \(c_2\) in \(A_1\) and \(A_2\) are as explicitly stated;
- \(c_3\) is a valid channel of the appropriate type.

Obviously, the new channel has to have the appropriate type to communicate
pairs of values: the first element is the value originally communicated by $c_1$, and the second, the value communicated by $c_2$.

The next law extends a channel synchronisation set.

**Law 5.8 (Channel Extension 1)**

$$A_1 \ll [\mathit{vs}_1 | \mathit{cs} | \mathit{vs}_2] \mathit{A}_2 = A_1 \ll [\mathit{vs}_1 | \mathit{cs} \cup \{c\} | \mathit{vs}_2] \mathit{A}_2$$

provided $c \notin \mathit{usedC}(\mathit{A}_1) \cup \mathit{usedC}(\mathit{A}_2)$

The new channel has to be indeed new, in the sense that it is not used in the actions in parallel.

The next law is a more elaborate version of the previous one, where the new channel is used to communicate a value $e$ from one of the parallel actions to the other.

**Law 5.9 (Channel Extension 2)**

$$A_1 \ll [\mathit{vs}_1 | \mathit{cs} | \mathit{vs}_2] \mathit{A}_2(e) = (c!e \to A_1 \ll [\mathit{vs}_1 | \mathit{cs} \cup \{c\} | \mathit{vs}_2] c?x \to A_2(x)) \setminus \{c\}$$

syntactic restrictions
- $c$ is a valid channel of the appropriate type;
- $c \notin \mathit{usedC}(\mathit{A}_1) \cup \mathit{usedC}(\mathit{A}_2)$;
- $x \notin \mathit{FV}(\mathit{A}_2)$.

provided $\mathit{FV}(e) \cap \mathit{wrtV}(\mathit{A}_2 \text{ before } e) = \emptyset$

The idea is that the access of $\mathit{A}_2$ to $e$ is removed. As in Law 5.5, we have to make sure that the value of $e$ is not changed before it is actually used in $\mathit{A}_2$.

Several other action laws of Circus can be found in Appendix A. The examples in the next section give a better intuition on their usefulness.

### 6 Examples

In this section, we consider the refinement of the buffer presented in Section 2. This case study is also considered in [13], but here we give the details of the refinement of the actions using the laws presented in the previous section. In [13], the focus is on laws for refining processes.

#### 6.1 Data refinement of the abstract buffer

At first, we carry out a data refinement to introduce a cache and a ring to record the buffered elements. In a non-empty buffer, the cache stores its head.
The ring is an array, whose two ends are regarded to be joined: a circular array. We use the indexes bottom and a top to determine the segment of the array that is in use. This segment records, at the concrete level, the tail of the abstract buffer.

A new constant maxring bounds the size of the ring: one less than maxbuff. The elements of the state of the concrete state are as expected.

```
process CBuffer ≜ begin
  CBufferState
  size : 0..maxbuff
  ringsize : 0..maxring
  cache : N
  top, bot : 1..maxring
  ring : seq N
  ringsize mod maxring = (top - bot) mod maxring
  ringsize = max{0, size - 1}
  #ring = maxring
```

The size of the ring, ringsize, may be computed from the positions of the top and bot indexes. Nevertheless, there is a subtlety, when top and bot coincide, a confusion arises as to whether the ring is full or empty. Therefore, it is necessary to add the equation that relates ringsize and size.

With the introduction of this new state, we need to provide a new description for the buffer main action. To establish that CBuffer is a refinement of the process Buffer presented in Section 2, we have to prove that the new main action is related to that of Buffer by forward simulation (Theorem 3.4).

Instead of proposing a new action from scratch, we consider the schema actions of Buffer and rely on the fact that forwards simulation distributes through the action constructors (Laws of Section 4). The actions of CBuffer have the same structure as those of Buffer, but use the new schema actions.

For this forwards simulation, the retrieve relation is as shown below. We use a shift operator: \( n ≪ a \) shifts the (circular) array \( a \) by \( n \) positions. For the sake of conciseness, we omit the simple inductive definition of this operator.

```
RetrBuffer
BufferState
CBufferState
buff = (1..size) ◁ ((cache) ⊕ ((bot - 1) ≪ ring))
```

Informally, if we shift the circular buffer so that bot occurs at position 1, and restrict the ring to the first size - 1 elements, then we have the tail of the abstract buffer. The head of the buffer, when it is non-empty, is directly represented by the cache.
The first schema action is the initialisation BufferInit. Initially, the buffer is empty and so has zero size; for the concrete initialisation, we choose some suitable values for top and bot.

\( \text{CBufferInit} \triangleq \left\{ \text{CBufferState}' \mid \text{size}' = 0 \land \text{bot}' = 1 \land \text{top}' = 1 \right\} \)

To prove that this new initialisation is related to BufferInit by forward simulation, we need to apply Law 4.1, which considers schema actions. In this case, however, the provisos are simplified because initialisation schemas do not include the state components that represent the before state and have true as precondition. We actually obtain the standard proviso for refinement of initialisations in \( Z \). All we have to prove is that

\[ \forall \text{BufferState}; \text{CBufferState}; \text{CBufferState}'; \bullet \]

\[ \text{RetrBuffer} \land \text{CBufferInit} \Rightarrow (\exists \text{CBufferState}'; \bullet \text{RetrBuffer}' \land \text{BufferInit}) \]

This is a simple proof: the one-point rule, and the fact that \( (\emptyset, s) = \langle \rangle \) for any sequence \( s \), can be used to reduce \( \exists \text{CBufferState}'; \bullet \text{RetrBuffer}' \land \text{BufferInit} \) to true. As this is on the right-hand side of the implication above, the proof-obligation is discharged.

The concrete input action corresponding to InputCmd has to consider whether the buffer is empty or not. If it is empty, then the input must be kept in the cache; if it is non-empty, then it must be passed on to the appropriate ring cell. When the input is cached, the top and bot indexes do not change.

\[ \text{CacheInput} \]

\[ \begin{array}{c}
\Delta \text{CBufferState} \\
x? : \mathbb{N}
\end{array} \]

\[ \begin{array}{c}
\text{size} = 0 \\
\text{size}' = 1 \land \text{cache}' = x? \\
\text{bot}' = \text{bot} \land \text{top}' = \text{top}
\end{array} \]

When the input is passed on to the ring, the corresponding value is stored and the top index advances.

\[ \text{StoreInput} \]

\[ \begin{array}{c}
\Delta \text{CBufferState} \\
x? : \mathbb{N}
\end{array} \]

\[ \begin{array}{c}
0 < \text{size} < \text{maxbuff} \\
\text{size}' = \text{size} + 1 \land \text{cache}' = \text{cache} \\
\text{bot}' = \text{bot} \land \text{top}' = (\text{top} \mod \text{maxring}) + 1 \\
\text{ring}' = \text{ring} \oplus \{\text{top} \mapsto x?\}
\end{array} \]

The overall state change caused by an input to the buffer can be captured by
the disjunction of the above two schemas.

\[ C_{\text{InputCmd}} \supseteq \text{CacheInput} \lor \text{StoreInput} \]

Again, we can justify this step with an application of Law 4.1. The proof-obligations are simple, if long; again this is data refinement as in Z. We observe that the precondition of \( C_{\text{InputCmd}} \) is the disjunction of the preconditions of \( \text{CacheInput} \) and \( \text{StoreInput} \), and amounts to \( \text{size} < \text{maxbuff} \).

Since \( C_{\text{InputCmd}} \) simulates \( \text{InputCmd} \), we can apply Laws 4.4 and 4.2 to obtain the following simulation of \( \text{Input} \). The structure of \( \text{Input} \) is preserved and \( \text{InputCmd} \) is replaced with \( C_{\text{InputCmd}} \).

\[ C_{\text{Input}}_0 \supseteq \text{size} < \text{maxbuff} \land \text{input}?x \rightarrow C_{\text{InputCmd}} \]

In this case, the guard is not changed and the first proviso of Law 4.4 is trivial. In the next development step (Section 6.2) this action is refined to refer directly to the \( \text{CacheInput} \) and \( \text{StoreInput} \) operations.

The refinement of \( \text{Output} \) is similar and, for brevity, we present only the resulting concrete action. As for the input, there is a case analysis for output. The output always comes from the cache, which must be replaced if the ring is non-empty. In the case that the ring is empty, we have \( \text{size} = 1; \text{size} \) is reset; nothing else changes.

<table>
<thead>
<tr>
<th>NoNewCache</th>
<th>( \Delta C_{\text{BufferState}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{size} = 1</td>
<td></td>
</tr>
<tr>
<td>\text{size}' = 0 \land \text{cache}' = \text{cache}</td>
<td></td>
</tr>
<tr>
<td>\text{bot}' = \text{bot} \land \text{top}' = \text{top} \land \text{ring}' = \text{ring}</td>
<td></td>
</tr>
</tbody>
</table>

If the ring is non-empty, then a new element (obtained from the ring) is stored in the cache; \( \text{bot} \) must be advanced.

<table>
<thead>
<tr>
<th>StoreNewCache</th>
<th>( \Delta C_{\text{BufferState}} )</th>
</tr>
</thead>
<tbody>
<tr>
<td>\text{size} &gt; 1</td>
<td></td>
</tr>
<tr>
<td>\text{size}' = \text{size} - 1 \land \text{cache}' = \text{ring}[\text{bot}]</td>
<td></td>
</tr>
<tr>
<td>\text{bot}' = (\text{bot} \mod \text{maxring}) + 1 \land \text{top}' = \text{top}</td>
<td></td>
</tr>
<tr>
<td>\text{ring}' = \text{ring}</td>
<td></td>
</tr>
</tbody>
</table>

The overall state change caused by an output from the buffer can be captured by the disjunction of the above two schemas.

\[ C_{\text{OutputCmd}} \supseteq \text{NoNewCache} \lor \text{StoreNewCache} \]

Law 4.1 can be used to justify that \( C_{\text{OutputCmd}} \) simulates \( \text{OutputCmd} \).
Laws 4.4 and 4.3 justify that $\text{COutput}_0$ below simulates $\text{Output}$.

$$C\text{Output}_0 \equiv \text{size} > 0 \& \text{output}!(\text{cache}) \rightarrow C\text{OutputCmd}$$

The output expression $\text{head buff}$ is replaced with $\text{cache}$. This is justified by $\text{RetrBuffer}$, which amounts to $\text{buff} = (1..\text{size}) \ll ((\text{cache}) \sim ((\text{bot} - 1) \ll \text{ring}))$ which implies that $\text{head buff}$ is $\text{cache}$, since, in the context of the communication, $\text{size} > 0$.

Finally, the main action of the centralised ring buffer also has the same structure of that of the original $\text{Buffer}$. We simply replace the original input and output actions with those presented above.

- $\text{CBufferInit}; \mu X \bullet (\text{CInput}_0 \sqcap \text{COutput}_0); X$

This step can be justified applying Law 4.6 and a distribution law of forwards simulation through sequential composition.

### 6.2 Decompose input and output actions

In the previous step, we have structured the state change resulting from an input to, or output from, the buffer in terms of separate operation schemas. This reflected a case analysis on whether the $\text{ring}$ (or rather just the $\text{cache}$) component needed to be accessed.

Nevertheless, the $\text{CInput}_0$ as well as the $\text{COutput}_0$ actions still refer to the compound operation which combines the two cases. This was intentional in the previous development step, where we keep an explicit correspondence between the abstract and concrete operations, in order to allow a simpler justification of the data refinement.

Here we perform a simple design step to promote the case analysis from the operations on the state to the control behaviour of the actions. We show only the refined actions and give the lemma that justifies the refinement later.

Input is enabled when the buffer is not full; in that case, the behaviour depends on whether the buffer is empty (in which case the corresponding state change is captured by $\text{CacheInput}$) or not (captured by $\text{StoreInput}$).

$$\text{CInput}_1 \equiv$$

$$\text{size} < \text{maxbuff} \& \text{input}?x \rightarrow$$

$$\text{size} = 0 \& \text{CacheInput}$$

$$\sqcap$$

$$\text{size} > 0 \& \text{StoreInput}$$

The output action is enabled when there is something in the buffer; the sub-
sequent behaviour depends on whether the ring is empty or not.

\[ C_{\text{Output}1} = \begin{cases} & \text{size} > 0 \& \text{output!cache} \rightarrow \text{size} > 1 \& \text{StoreNewCache} \\ & \Box \text{size} = 1 \& \text{NoNewCache} \end{cases} \]

The following lemma formalises the refinement of the input action.

**Lemma 6.1 (Refinement of \( C_{\text{Input}0} \))**

\( C_{\text{Input}0} \sqsubset A C_{\text{Input}1} \)

**Proof**

\[ LHS \]
\[ \sqsubset A \{ \text{Law 5.1} \} \]
\[ \text{size} < \text{maxbuff} \& \text{input}?x \rightarrow \text{size} = 0 \& \text{CInputCmd} \]
\[ \Box \text{size} > 0 \& \text{CInputCmd} \]
\[ \sqsubset A \{ \text{Law 5.2} \} \]

\[ RHS \]

The refinement of the output action is captured by an analogous lemma whose proof follows from exactly the same two laws used above.

6.3 Parallelisation of the input and output actions

In this step, we refine the actions with the aim of obtaining two independent sets of paragraphs: one that accesses exclusively the ring, and another that accesses the remaining components. This is part of a development strategy where each partition of the state space together with its actions gives rise to a new process. The purpose here is to illustrate how such a decomposition can be justified by formal refinement.

The \( \text{StoreInput} \) operation modifies top and ring. Decomposing it into two operations has to deal with the fact that the operation concerned with updating the ring needs the input \((x?)\) and the value of top. To solve this problem, we introduce communication. We use the channels below.

\[ \text{channel} \quad \text{write}, \text{read} : (1 \ldots \text{maxring}) \times \mathbb{N} \]

They are hidden in the design and implementation.

We write the state of the buffer as two separate schemas; each one is a
state space for a set of process paragraphs.

\[
\begin{align*}
\text{ControllerState} & \quad \text{size} : 0 \ldots \text{maxbuff} \\
& \quad \text{ringsize} : 0 \ldots \text{maxring} \\
& \quad \text{cache} : \mathbb{N} \\
& \quad \text{top, bot} : 1 \ldots \text{maxring} \\
\text{ringsize} &= \max\{0, \text{size} - 1\} \\
\text{ringsize mod maxring} &= (\text{top} - \text{bot}) \mod \text{maxring}
\end{align*}
\]

\[
\begin{align*}
\text{RingState} & \triangleq [\text{ring} : \text{seq} \mathbb{N} | \#\text{ring} = \text{maxring}] \\
\text{BufferState} & \triangleq \text{ControllerState} \land \text{RingState}
\end{align*}
\]

Here, we split the actions StoreInput and StoreNewCache presented in the previous section into actions on either ControllerState or RingState. For StoreInput, we aim at the actions below.

\[
\begin{align*}
\text{StoreInputController} & \quad \Delta \text{ControllerState} \\
& \quad \Delta \text{RingState} \\
& \quad 0 < \text{size} < \text{maxbuff} \\
& \quad \text{size}' = \text{size} + 1 \land \text{cache}' = \text{cache} \\
& \quad \text{bot}' = \text{bot} \land \text{top}' = (\text{top} \mod \text{maxring}) + 1
\end{align*}
\]

\[
\begin{align*}
\text{StoreRingCmd} & \quad \Delta \text{RingState} \\
& \quad \Delta \text{ControllerState} \\
& \quad i? : 1 \ldots \text{maxring} \\
& \quad y? : \mathbb{N} \\
& \quad \text{ring}' = \text{ring} \oplus \{i? \mapsto y?\}
\end{align*}
\]

The lemma in the sequel establishes that the input action of the previous section, CInput₁, can be refined to the following action.

\[
\begin{align*}
\text{CInput} & \triangleq \text{size} = 0 \land \text{input}?x \rightarrow \text{CacheInput} \\
& \quad \square 0 < \text{size} < \text{maxbuff} \land \text{input}?x \rightarrow \\
& \quad \quad (\text{write}?i?y \rightarrow \text{StoreRingCmd} \\
& \quad \quad \quad [\alpha(\text{RingState}) \mid \{\text{write}\} \mid \alpha(\text{ControllerState})] \\
& \quad \quad \quad \text{write}.\text{top}!x \rightarrow \text{StoreInputController} \setminus \{\text{write}\})
\end{align*}
\]

The proof of the following lemma illustrates the application of the laws for
introducing parallelism. We also reference some simple laws of CSP which are presented in Appendix A. From the identification of each law which annotates a refinement step it is clearly whether it is presented in Section 5 or in Appendix A.

Lemma 6.2 (Refinement of $C_{\text{Input}}_1$) \quad $C_{\text{Input}}_1 \sqsubseteq_A C_{\text{Input}}$

Proof

LHS

\[ \sqsubseteq_A \{ \text{Law A.1} \} \]

\[ \begin{align*}
\text{size} < \text{maxbuff} & \quad \text{size} = 0 & \text{input}\!?:x \rightarrow \text{CacheInput} \\
\Box & \text{size} > 0 & \text{input}\!?:x \rightarrow \text{StoreInput}
\end{align*} \]

\[ \sqsubseteq_A \{ \text{Laws A.2, A.3} \} \]

\[ \begin{align*}
\text{size} = 0 & \quad \text{input}\!?:x \rightarrow \text{CacheInput} \\
\Box & \text{0} < \text{size} < \text{maxbuff} & \text{input}\!?:x \rightarrow \text{StoreInput}
\end{align*} \]

\[ \sqsubseteq_A \{ \text{Law 5.4} \} \]

RHS

We proceed in much the same way for the output action. We replace $\text{StoreNewCache}$ with an action on the state components of the controller.

\[ \text{StoreNewCacheController} \]

\[ \begin{align*}
\Delta & \text{ControllerState} \\
\Delta & \text{RingState} \\
x? & : \mathbb{N}
\end{align*} \]

\[ \begin{align*}
\text{size} > 1 & \\
\text{size}' & = \text{size} - 1 & \text{cache}' & = x? \\
\text{bot}' & = (\text{bot mod maxring}) + 1 & \text{top}' & = \text{top}
\end{align*} \]

With this, we can refine the output action of the previous section as follows.

\[ C_{\text{Output}} \triangleq \text{size} > 1 \quad \text{&} \quad \text{output!}\text{cache} \rightarrow \]

\[ \begin{align*}
\text{(read}\!?:i!\text{ring}[i] & \rightarrow \text{Skip} \\
\| & \text{\{\alpha (\text{RingState}) | \{} \text{read} \} | \alpha (\text{ControllerState})\}}
\end{align*} \]

\[ \begin{align*}
\Box & \text{size} = 1 \quad \text{&} \quad \text{output!}\text{cache} \rightarrow \text{NoNewCache}
\end{align*} \]

The lemma below gives the formal justification.
Lemma 6.3 (Refinement of $COutput_1$)

$COutput_1 \sqsubseteq_A COutput$

Proof

$LHS$

$\sqsubseteq_A \{\text{Law A.1}\}$

size > 0 &

size > 1 & output!cache $\rightarrow$ StoreNewCache

$\square$ size = 1 & output!cache $\rightarrow$ NoNewCache

$\sqsubseteq_A \{\text{Laws A.2, A.3}\}$

size > 1 & output!cache $\rightarrow$ StoreNewCache

$\square$ size = 1 & output!cache $\rightarrow$ NoNewCache

$\sqsubseteq_A \{\text{Law 5.5}\}$

size > 1 & output!cache $\rightarrow$

$(\text{Skip}; \text{read}_2!\text{ring}[\text{bot}] \rightarrow \text{Skip}$

\[\alpha(\text{RingState}) \mid \{ \text{read}_2 \} \mid \alpha(\text{ControllerState})]\]

$\text{read}_2?x \rightarrow \text{StoreNewCacheController} \setminus \{ \{ \text{read}_2 \} \}$

$\square$ size = 1 & output!cache $\rightarrow$ NoNewCache

$\sqsubseteq_A \{\text{Laws A.4, 5.9}\}$

size > 1 & output!cache $\rightarrow$

$((\text{read}_1?i \rightarrow \text{read}_2!\text{ring}[i] \rightarrow \text{Skip}$

\[\alpha(\text{RingState}) \mid \{ \text{read}_1, \text{read}_2 \} \mid \alpha(\text{ControllerState})]\]

$\text{read}_1!\text{bot} \rightarrow \text{read}_2?x \rightarrow$

$\text{StoreNewCacheController} \setminus \{ \{ \text{read}_1 \} \} \setminus \{ \{ \text{read}_2 \} \}$

$\square$ size = 1 & output!cache $\rightarrow$ NoNewCache

$\sqsubseteq_A \{\text{Laws A.5, 5.7}\}$

$RHS$

$\square$

The rest of the development can be found in [13]. Our emphasis here, as already mentioned, was in the development of the actions. In [13] we proceed with the development at the level processes and end up with a parallel composition of processes: a controller and a ring.
7 Related and future work

An action system consists of a set of state variables and a set of guarded actions on these variables; the behaviour of the action system is the repeated execution of enabled actions. This is a general model for parallelism. A model for concurrency with shared variables is obtained by partitioning the actions amongst different processes. A model for distributed systems is obtained by partitioning the variables amongst the processes. The emphasis is on the state of an action system, with interaction described through the interference of shared variables.

Back and Sere [1] describe the combination of the refinement calculus and action systems in the derivation of parallel and distributed algorithms. They start from a purely sequential algorithm and proceed by stepwise refinement. Most steps are accomplished as sequential refinements, with parallelism being introduced through the decomposition of atomic actions.

The main difference between the action system approach and Circus is due to the very basic nature of the action system formalism in comparison with process algebra. Control flow in an action system is simple: select an enabled guard and execute it. This gives a very flat structure where auxiliary variables are needed to guarantee the proper sequencing of actions. In Circus, control flow is described using the process algebraic operators of CSP, and as a result, a rich set of laws are available for process and action refinement that have no direct correspondence in action systems.

The two approaches are formally linked: Woodcock and Morgan [9,17] show how to calculate the failures-divergences semantics of an action systems and provide complete techniques for data refinement. Butler [2] extends this work to include internal actions and unbounded nondeterminism. With these links, we may be able to take inspiration from the rules related to decomposition in Back and Sere’s work to propose further laws for Circus.

Olderog [11] introduces a design calculus for occam-like communicating programs that allows for the stepwise development of correct programs. The programs are given an imperative trace-readiness semantics, and specifications are given in terms of language-theoretic and assertional techniques. The program and specification semantics are uniformly presented in a predicative style similar in spirit to that of unifying theories of programming. In fact, both works have roots in the Esprit ProCoS project. The design rules of [11] can be another source of inspiration for further refinement laws for Circus actions.

We are conducting a series of formal developments of concurrent programs using Circus, both in academia, as case studies, and in industry, as part of a commercial project. This work is leading to the discovery of new refinement laws and design rules. As we gain more experience from their practical use, we hope that the set of Circus laws will become comprehensive.

The verification of the laws of Circus is a major task that is currently underway. The proofs are being carried out by hand and peer review, but may
be formalised mechanically later on. We have recently completed a mecha-
nisation of a major part of the semantic metalanguage in both Z/Eves and
ProofPowerZ. This provides the possibility of machine-checking the proofs,
although such an exercise is very labour-intensive.

We have also recently started work on tools for Circus. A parser is complete,
and we are now working on a model-checker for Circus refinement. A tool to
support the application of the laws presented here and the others that are to
come is also in our plans.

Acknowledgements

This work is partially supported by the EPSRC grant GR/R43211/01 on
“Refinement calculi for sequential and concurrent programs” and the QinetiQ
grant CU009-019346 on “Advances in formal modelling and concurrency as
represented by the Circus language”. The work of Ana Cavalcanti and Augusto
Sampaio is partially supported by CNPq: grants 520763/98-0 and 521039/95-
9. We are grateful to Arthur Hughes for his suggestions.

A  Laws

Here we introduce some additional laws of actions used to justify the refine-
ments presented in Section 6. All these laws are valid for CSP and express
very simple and standard relationships among the CSP operators.

The first two laws state that both prefix and guard distribute through
external choice.

Law A.1 (Prefix/External choice — Distribution)

\[ c \rightarrow \Box i \bullet g_i \land A_i = \Box i \bullet g_i \land c \rightarrow A_i \]

**syntactic restriction** \( FV(g_i) \cap \alpha(c) = \emptyset \), for all \( i \)

provided \( \lor i \bullet g_i \)

The proviso is needed to ensure that at least one guard is valid, so that in the
right-hand side action the communication does take place.

Law A.2 (Guard/External choice — Distribution)

\[ g \land (A \Box B) = (g \land A) \Box (g \land B) \]

The following law states that nested guards can be combined by taking
their conjunction.
Law A.3 (Guard combination)
\[ g_1 \land (g_2 \land A) = (g_1 \land g_2) \land A \]

A well-known property is that \textit{Skip} is the unity of sequential composition.

Law A.4 (Sequential Composition — Unit)
\[ \text{Skip}; \ A = A = A; \ \text{Skip} \]

Nested hidden sets of channels can be combined.

Law A.5 (Hide combination)
\[ (A \setminus cs_1) \setminus cs_2 = A \setminus (cs_1 \cup cs_2) \]

B Proof of a law

In this section, we describe the proof of the following simplified version of Law 5.4, where we have synchronisation events, instead of communications, and the schema actions are total.

\[ Op_1 \land Op_2 = ((a \rightarrow Op_1) \llbracket \text{wrt} V(A_1) \mid \{a\} \mid \text{wrt} V(A_2) \rrbracket (a \rightarrow Op_2)) \setminus \{a\} \]

This result establishes that the conjunction of schemas \textit{Op}_1 and \textit{Op}_2, which are assumed to be total and act on disjoint state spaces, can be implemented in parallel. The channel \textit{a} does not have a declared type, and therefore is a synchronisation event.

This result is interesting because, as illustrated in our examples, it may be convenient to perform a synchronisation before executing the operations. We conduct the proof in the context of the unifying theory.

The semantics of parallel composition merges the observational and state variables of parallel actions. Suppose \textit{A}_1 and \textit{A}_2 are parallel actions with disjoint states, synchronising on every event, then the semantics of their parallel composition is defined as follows.

\[ \llbracket \textit{A}_1 \mid \text{wrt} V(A_1) \mid \text{used} C(A_1) \cup \text{used} C(A_2) \mid \text{wrt} V(A_2) \rrbracket \text{A}_2 \]

We name \textit{owr} the following set of observational variables.

\[ \text{owr} \equiv \{\text{okay, wait, ref}\} \]

Above, we use a slight abuse of notation to rename in the semantics of \textit{A}_1.
and $A_2$ the dashed versions of each of these variables to prefix them with a 1 and a 2. This renaming allows us to differentiate between each action’s final values of these variables; it remains for us to merge these values to produce the joint final values, and this is carried out by the merge operation, $N$. There is no need to differentiate the final values of the trace and the state: since the actions are synchronising on every event, the actions will produce identical traces; since the actions are working in separate partitions, the conjunction of the partitions is a satisfactory final state. In the unifying theories, this style of semantics is known as parallel by merge.

$$N \triangleq okay' = (1.okay \land 2.okay) \land wait' = (1.wait \lor 2.wait) \land$$

$$tr' = tr \land ref' = (1.ref \lor 2.ref) \land II \Sigma$$

The parallel composition has the product state and its trace is a trace of both actions. The definition of $N$ above specifies that the parallel composition is divergence-free if both actions are; it is waiting if either action is; and it refuses an event if either action could refuse it.

We now consider $\rho_1$ and $\rho_2$, total relations over disjoint states with sets of components $vs_1$ and $vs_2$. These relations are a representation of schema operations in the unifying theory.

Each relation may be promoted into an action by describing that its effect on the observational variables is the same as that of $\text{Skip}$: it preserves the values of the observations; if its predecessor has terminated, then the initial value of the refusal set is irrelevant.

$$\text{Skip}(\rho) \triangleq \Pi < wait > (\exists ref \bullet \Pi^{\text{Obs}} \land \rho)$$

This is captured above in the relational calculus with a conditional on the value of $wait$. If $wait$ is true, then the predecessor has terminated and the behaviour is described by the identity on all observational and state variables, $\Pi$. If $wait$ is false, then the behaviour is described by $\exists ref \bullet \Pi^{\text{Obs}} \land \rho$, where $\Pi^{\text{Obs}}$ is the identity over the observational variables only.

We now consider the parallel composition of $\text{Skip}(\rho_1)$ and $\text{Skip}(\rho_2)$. The synchronisation set $C$ is irrelevant, since these actions never carry out any communications. We use $\Pi^C$ to denote the identity on the state variables.

\[
\text{Skip}(\rho_1) \llbracket vs_1 \mid C \mid vs_2 \rrbracket \text{Skip}(\rho_2) \\
= \{ \text{by definition of parallel operator} \} \\
(\text{Skip}(\rho_1)[1.\text{owr}'/\text{owr}'] \land \text{Skip}(\rho_2)[2.\text{owr}'/\text{owr}']) ; N \\
= \{ \text{by definition of Skip(\rho)} \} \\
(\Pi[1.\text{owr}'/\text{owr}'] < wait > (\exists ref \bullet \Pi^{\text{Obs}}[1.\text{owr}'/\text{owr}'] \land \rho_1)) \land \\
(\Pi[2.\text{owr}'/\text{owr}'] < wait > (\exists ref \bullet \Pi^{\text{Obs}}[2.\text{owr}'/\text{owr}'] \land \rho_2)) ; N
\]
\[\begin{align*}
\{(P_1 \triangleright b \triangleright Q_1) \land (P_2 \triangleright b \triangleright Q_2) & = (P_1 \land P_2) \triangleright b \triangleright (Q_1 \land Q_2) \} \\
((\Pi\{1.\text{owr}'/\text{owr}'\} \land \Pi\{2.\text{owr}'/\text{owr}'\}) & \triangleright \text{wait} \triangleright \\
((\exists \text{ref} \cdot \Pi^{\text{Obs}\{1.\text{owr}'/\text{owr}'\}} \land \rho_1) \land \\
(\exists \text{ref} \cdot \Pi^{\text{Obs}\{2.\text{owr}'/\text{owr}'\}} \land \rho_2)) \land \\
N & = \{(P \triangleright b \triangleright Q); \ R = (P; \ R) \triangleright b \triangleright (Q; \ r) \} \\
((\Pi\{1.\text{owr}'/\text{owr}'\} \land \Pi\{2.\text{owr}'/\text{owr}'\}) & \triangleright \text{wait} \triangleright \\
((\exists \text{ref} \cdot \Pi^{\text{Obs}\{1.\text{owr}'/\text{owr}'\}} \land \rho_1) \land \\
(\exists \text{ref} \cdot \Pi^{\text{Obs}\{2.\text{owr}'/\text{owr}'\}} \land \rho_2)) \land \\
N & = \text{by definition of } \Pi \text{ and } N, \text{ and predicate calculus} \\
1.\text{okay}' = \text{okay} \land 2.\text{okay}' = \text{okay} \land 1.\text{wait}' = \text{wait} \land \\
2.\text{wait}' = \text{wait} \land \text{tr}' = \text{tr} \land 1.\text{ref}' = 2.\text{ref}' = \text{ref} \land \Pi' \land \\
\text{okay}' = (1.\text{ok} \land 2.\text{ok}) \land \text{wait}' = (1.\text{wait} \lor 2.\text{wait}) \land \\
\text{tr}' = \text{tr} \land \text{ref}' = (1.\text{ref} \lor 2.\text{ref}) \land \Pi' \land \\
\triangleright \text{wait} \triangleright \\
1.\text{okay}' = \text{okay} \land 2.\text{okay}' = \text{okay} \land 1.\text{wait}' = \text{wait} \land \\
2.\text{wait}' = \text{wait} \land \text{tr}' = \text{tr} \land \rho_1 \land \rho_2; \\
\text{okay}' = (1.\text{okay} \land 2.\text{okay}) \land \text{wait}' = (1.\text{wait} \lor 2.\text{wait}) \land \\
\text{tr}' = \text{tr} \land \text{ref}' = (1.\text{ref} \lor 2.\text{ref}) \land \Pi' \land \\
= \text{by definition of } \Pi \text{ and predicate calculus} \\
\Pi \triangleright \text{wait} \triangleright \\
1.\text{okay}' = \text{okay} \land 2.\text{okay}' = \text{okay} \land 1.\text{wait}' = \text{wait} \land \\
2.\text{wait}' = \text{wait} \land \text{tr}' = \text{tr} \land \rho_1 \land \rho_2; \\
\text{okay}' = (1.\text{okay} \land 2.\text{okay}) \land \text{wait}' = (1.\text{wait} \lor 2.\text{wait}) \land \\
\text{tr}' = \text{tr} \land \text{ref}' = (1.\text{ref} \lor 2.\text{ref}) \land \Pi' \land \\
\end{align*}\]
\( = \{ \text{by definition of sequential composition and predicate calculus} \} \)

\[
\begin{align*}
\mathcal{I} & \triangleleft \text{wait} \triangleright \\
(\exists \text{ref}_{0}, \text{ref}_{0} \bullet) & \quad \text{okay}' = \text{okay} \land \text{wait}' = \text{wait} \land tr' = tr \land \\
\text{ref}' &= (1.\text{ref}_{0} \cup 2.\text{ref}_{0}) \land \rho_{1} \land \rho_{2} \\
\end{align*}
\]

\( = \{ \text{by set theory and predicate calculus} \} \)

\[
\begin{align*}
\mathcal{I} & \triangleleft \text{wait} \triangleright \\
(\exists \text{ref} \bullet) & \quad \text{okay}' = \text{okay} \land \text{wait}' = \text{wait} \land \text{tr}' = \text{tr} \land \text{ref}' = \text{ref} \land \rho_{1} \land \rho_{2} \\
\end{align*}
\]

\( = \{ \text{by definition of } \mathcal{I}^{\text{Obs}} \} \)

\[
\begin{align*}
\mathcal{I} & \triangleleft \text{wait} \triangleright (\exists \text{ref} \bullet \mathcal{I}^{\text{Obs}} \land \rho_{1} \land \rho_{2}) \\
\end{align*}
\]

\( = \{ \text{by definition of } \text{Skip}(\rho) \} \)

\( \text{Skip}(\rho_{1} \land \rho_{2}) \)

Now we can prove the law correct, using this result and some simple laws of the unifying theory similar to those of CSP processes.

\[
((a \rightarrow \text{Skip}(\rho_{1})) \parallel vs_{1} | \{a\} | vs_{2} || (a \rightarrow \text{Skip}(\rho_{2}))) \setminus \{a\}
\]

\( = \{ \text{by a step law for parallelism} \} \)

\[
(a \rightarrow (\text{Skip}(\rho_{1}) \parallel vs_{1} | \{a\} | vs_{2} || \text{Skip}(\rho_{2}))) \setminus \{a\}
\]

\( = \{ \text{by a property of hiding} \} \)

\[
(\text{Skip}(\rho_{1}) \parallel vs_{1} | \{a\} | vs_{2} || \text{Skip}(\rho_{2})) \setminus \{a\}
\]

\( = \{ \text{there are no communications in } \text{Skip}(\rho) \text{ and } \text{Skip}(\rho) \} \)

\( \text{Skip}(\rho_{1}) || vs_{1} | \{a\} | vs_{2} || \text{Skip}(\rho_{2}) \)

\( = \{ \text{by the result above} \} \)

\( \text{Skip}(\rho_{1} \land \rho_{2}) \)

The other laws can be proved in a similar manner.
References


