Using Circus for Safety-critical Applications

J. C. P. Woodcock\(^1,2\)

Computing Laboratory
University of Kent
Canterbury, UK

Abstract

Circus is a language that unifies Z, CSP, and the refinement calculus, and we describe its application in the development of safety-critical systems. We show the descriptive power of Circus with a fragment of the formalisation of the steam boiler problem. We then use Circus’s refinement calculus to bridge a semantic gap in development, where we eliminate a kind of abstract event.

Keywords: Unifying theories of programming, the Z notation, Communicating Sequential Processes (CSP), the refinement calculus, Circus, synchronisation protocols.

1 Introduction

The Circus notation [31,33,12,13] is being used in the development and verification of safety-critical systems. It is a semantic compound of Z [6,28,29,34], CSP [15,22], the Z refinement calculus [7,11], and the guarded-command language [14], providing a framework in which to discuss the correctness of state-based reactive systems; extensions are under development to include aspects of object orientation [10], time [25,26,27], probability, and lock-step synchrony.

In this paper, we illustrate the use of Circus by discussing a case study: the steam boiler, now a standard benchmark for formal methods [5,1,2]. In particular, we focus on an interesting “semantic gap”. In CSP, an abstraction is sometimes employed in which atomic synchronisations can be system-wide, between many processes, rather than being restricted to only two participants.

\(^1\) The WMF 2003 Formal Methods Europe Lecturer (www.fmeurope.org).
\(^2\) Email: J.C.P.Woodcock@kent.ac.uk
We deal with a simple instance of this phenomenon, which serves to show the power of Circus’s calculational approach to reasoning about reactive systems.

2 Circus

Processes in Circus are reactive and encapsulate state variables; the reactive behaviour is described using CSP, and states and their transitions are described in Z. Circus focuses particularly on techniques for refinement [8,9,24]. An objective is to use Circus to calculate reactive systems in a style reminiscent of the sequential refinement calculi of Back [4], Morgan [19], and Morris [20].

As a simple example of a Circus process, consider the modelling of a bank account. The environment can interact with this process through four channels: to open the account; to check the current balance; to deposit money; and to withdraw money. When the account is opened, an initial deposit is made and an overdraft limit is established. The description starts with the declaration of these four channels, with their types.

```
channel open: N × N; balance, deposit, withdraw : N
```

The process consists of a series of paragraphs that introduce the state and various actions on the state; these declarations are given in a mixture of Z and CSP, and are protected from the outside world by the process’s encapsulation. The extensional behaviour is given by the main action.

```
process BankAcc ⊑
begin

state Acc ⊑ [b : Z; o, f : N | f = o + b]
Init ⊑ [Acc'; init_b?, init_o? : N | b' = init_b? ∧ o' = init_o?]
Deposit ⊑ deposit? v → b := b + v
Deduct ⊑ (a : N • a ≤ f & b := b − a)
Withdraw ⊑ withdraw? v → Deduct(v)
Balance ⊑ balance! b → Skip

main

open? (init_b, init_o) → Init;
µ X • (Deposit □ Withdraw □ Balance); X

end
```
The state of the bank account contains the balance $b$, the overdraft limit $o$, and the funds available $f$, which is always fixed to be the sum of the balance and the overdraft limit. The value of $o$ is a limit, in the sense that $f$ must never be allowed to become negative. The main action ensures that the account is opened before it is used. The environment must supply the initial balance and the overdraft limit, after which the state is updated according to the initialisation schema $Init$. Notice that the variables bound by the communication ($init_b$ and $init_o$) are referred to as inputs to $Init$. Next, the process repeatedly offers the environment a choice of actions: to deposit, to withdraw, or to check the account’s balance. The $Deposit$ action is activated by a communication on the $deposit$ channel; the balance is updated accordingly. $Withdraw$ is similar, except that the action uses another, parametrised action $Deduct$, which contains a guard: if the amount to be deducted is greater than the funds available, then the action deadlocks. This example of reactive behaviour is impossible to express in the Oxford-style of using $Z$.

The semantics of Circus [33] follows Hoare & He’s unifying theories of programming [16]: each construct in the language is given a denotational semantics in a predicative style in an alphabetised extension of Tarski’s relational calculus. The language corresponds to their model in the imperative, reactive subtheory, and provides the setting for an extensive refinement calculus.

3 A safety-critical application: the steam boiler

The problem given in [5] is to model a steam boiler, where water is removed from the boiler at a measured rate, and replaced by the action of four pumps delivering a supply from a reservoir of water. Such devices are important components of nuclear reactor systems, amongst other things. The steam boiler’s controller is given the task of maintaining the boiler’s water level within the range $N_1 .. N_2$, and to shut the system down if there is a risk that the water level could go outside the wider bounds $M_1 .. M_2$. If the wider bounds are exceeded, then there is a possibility of flooding the boiler or allowing it to boil dry, either of which might lead to an accident with unacceptable consequences.

Circus has been used to describe the steam boiler in [32]. In that paper, an interesting separation is made between handling messages with complex data from sensors, and the rather simpler finite-state machine that controls the boiler’s operation, in a manner reminiscent of abstract interpretation. This separates the question of validating the controller’s interface from the question of verifying its correctness.

In this paper we use another formalisation based on that of O’Halloran [21]. In his work, the approach to identifying critical requirements is to express
functional requirements as firing rules, of the form “if $a$ then $b$”, where event $a$ enables event $b$, subject to environmental constraints. The implicit inference engine defined by these firing rules is non-monotonic, as it must forget previously inferred facts as the system evolves.

The model of the steam-boiler plant and the inference engine share communication channels that represent communications with actuators and sensors in the real system. The plant consists of a number of components, and we give a brief description of some of them: the water reservoir, the boiler, and a sensor. The reservoir is a model of the body of water that the pumps feed, and that the boiler turns into steam by heating, according to the dynamics of the system. First, the quantity in the body of water, the state variable $L$, is set to an arbitrary initial level between $N_1$ and $N_2$.

```plaintext
process Water ≜
begin main
  var set : P Source, L : N •
  
  set := Source; L ∈ N_1 .. N_2;
  μ X • min < L < max & write?x : set?δ →
    set, L := set \ {x}, L + δ; X
  □

  set = ∅ &∧ min < L < max & read.sensor!L →
    tock →
    set := source; X
  □

  set = ∅ & tock → set := source; X
  □

  L ≤ min & boom → Stop
  □

  L ≥ max & flood → Stop
  □

  emergency_stop & Emergency_Stop
end
```
The process maintains two state variables: a set of water sources, and the water level in the boiler. Time moves forward whenever the clock tocks; within each time period, water is transferred from each source. Once all the sources have been accounted for, the current level is output on the channel read.sensor. If the level drops below the minimum, there is an explosion, signalled by the event boom; if it goes above the maximum, there is another catastrophic event flood. The process shuts down following the emergency stop event.

Next, we model the action of the boiler as a separate process that interacts with the body of water. The boiler consumes $v$ litres of water in each time period, and communicates this fact to Water.

$$\text{Boiler} \equiv \text{begin} \text{write.boiler.}(-v) \to \text{tock} \to \text{Boiler} \text{end}$$

Finally, we model the behaviour of a sensor, which interacts with the body of water, and emits events that abstract the state of the boiler.

$$\text{process Sensor} \equiv \text{begin main}$$

$$\mu X \bullet \text{tock} \to \text{tock} \to \text{tock} \to \text{tock} \to \text{read.sensor?}L : N \to$$

$$L < M_1 \& \text{level.lt}_M_1 \to X$$

$$\square$$

$$M_1 \leq L < N_1 \& \text{level.lt}_N_1 \to \text{tock} \to X$$

$$\square$$

$$N_1 \leq L \leq N_2 \& \text{level.between}_N_1_N_2 \to \text{tock} \to X$$

$$\square$$

$$N_2 < L \leq M_2 \& \text{level.gt}_N_2 \to \text{tock} \to X$$

$$\square$$

$$M_2 < L \& \text{level.gt}_M_2 \to \text{tock} \to X$$

$$\text{end}$$

Every fourth time period, the sensor takes a reading from the water level and enables an appropriate event.

Although we have shown only a small part of the specification of the steam boiler, it is sufficient to demonstrate the approach. The model consists of a number of parallel processes interacting through shared channels and events. We would like to be able to implement this model as a controller for a real steam boiler. A suitable language for this implementation is occam, given
its close relationship with CSP; however, we immediately face a semantic gap: CSP allows the synchronisation of events to be between many processes, but occam restricts this, for efficiency reasons, to just two participants. In the next section, we apply Circus’s refinement calculus to bridge this semantic gap for a specific kind of multi-way synchronisation.

4 A simple protocol for multi-way synchronisation

Suppose that we have a collection of parallel processes indexed over $I$, that each process repeatedly executes some individual transaction, represented by the event $t.i$, and that we synchronise the transactions by alternating them with the globally shared event $m$. This is modelled by the Circus process:

$$\parallel_{i : I} (\mu X \cdot m \rightarrow t.i \rightarrow X) \setminus \{m\}$$

This is the parallel combination of a collection of processes, where the $i$-th process alternates forever between engaging in the multi-way synchronisation $m$ and the independent event $t.i$. Notice that every process participates in the multi-way synchronisation, whereas only the $i$-th process participates in the independent event $t.i$. Also notice that $m$ is hidden from the environment. Thus, crucially for this development, we know the identities of all of $m$’s participants. If the membership were dynamic, then we would need to develop a protocol to manage its membership.

We now use Circus’s refinement calculus to derive a protocol that is equivalent to this system, but where there is no multi-way synchronisation. Our first step is to convert the $i$th process into an action system [3]. An action system is a process consisting of a loop containing the choice between events guarded by firing conditions in terms of the process’s state. Following each event, there is an action that updates the state.

At present, the flow of control in our process is described by the operators of Circus’s process algebra; when we convert to an action system, we record the current control position explicitly in the new variable $p$, the program counter.

$$\parallel_{i : I} (\mu X \cdot m \rightarrow t.i \rightarrow X) \setminus \{m\}$$

$$= \{ \text{action-system-conversion} \}$$
\[
\left\{ \parallel \{m\} i : I \bullet \right. \\
\text{\textbf{var} } p : PC := m \bullet \\
\mu X \bullet p = m \& m \rightarrow p := t; \ X \\
\left. \square \ \\
p = t \& t.i \rightarrow p := m; \ X \right) \setminus \{m\}
\]

Our next step is to widen the scope of the local variables; to avoid confusing their values, we index each of the variables as an element of the new array-valued variable \( p \). The choice of locality of scope is familiar to programmers.

\[
= \{ \text{parallel-state} \} \\
\left( \begin{array}{c}
\text{array } p : I \rightarrow PC \bullet \\
( \parallel \{m\} i : I \bullet \\
p[i] := m; \\
\mu X \bullet p[i] = m \& m \rightarrow p[i] := t; \ X \\
\left. \square \ \\
p[i] = t \& t.i \rightarrow p[i] := m; \ X \right) \\
\end{array} \right) \setminus \{m\}
\]

The initialisation of the position counters can be moved out from the individual processes to form a simultaneous assignment that takes place after the declaration of the array, but before the execution of any of the individual processes, as an interleaving of individual assignments.

\[
= \{ \text{parallel-assignment} \} \\
\left( \begin{array}{c}
\text{array } p : I \rightarrow PC \bullet \\
( \parallel i : I \bullet p[i] := m); \\
( \parallel \{m\} i : I \bullet \\
\mu X \bullet p[i] = m \& m \rightarrow p[i] := t; \ X \\
\left. \square \ \\
p[i] = t \& t.i \rightarrow p[i] := m; \ X \right) \\
\end{array} \right) \setminus \{m\}
\]

Now the parallel composition can be flattened into a sequential action system. The event \( m \), shared between the individual processes, gives rise to an action
whose guard is the conjunction of all the individual processes’ guards. The events \( t.i \) may occur independently.

\[
= \{ \text{action-system-parallel} \}
\]

\[
\begin{array}{l}
\text{array } p : I \rightarrow PC \bullet \\
( \|\| i : I \bullet p[i] := m ) ; \\
\mu X \bullet (\forall i : I \bullet p[i] = m ) & m \rightarrow \\
( \|\| i : I \bullet p[i] := t ) ; X \\
\square \\
\square i : I \bullet p[i] = t & t.i \rightarrow p[i] := m ; X \\
\end{array}
\]

\[
\{ m \}
\]

The operation of hiding the event \( m \) distributes through the array declaration, the outermost sequential composition, and the fixed-point operator. Since the event \( m \) doesn’t occur in the simultaneous assignment, hiding \( m \) is vacuous.

\[
= \{ \text{hiding-distribution} \}
\]

\[
\begin{array}{l}
\text{array } p : I \rightarrow PC \bullet \\
( \|\| i : I \bullet p[i] := m ) ; \\
\mu X \bullet (\forall i : I \bullet p[i] = m ) & m \rightarrow \\
( \|\| i : I \bullet p[i] := t ) ; X \\
\square \\
\square i : I \bullet p[i] = t & t.i \rightarrow p[i] := m ; X \\
\end{array}
\]

\[
\{ m \}
\]

Hiding also distributes through the action system. The hidden event’s guard is either true or false. If the guard is false, then its internalised action is irrelevant. If the guard is true, then the hidden action may occur; but the other actions may or may not be available.

\[
= \{ \text{hiding-conditional-external-choice-distribution-2} \}
\]
array \ p : I \rightarrow PC \bullet
\begin{align*}
( & \parallel i : I \bullet p[i] := m) ; \\
\mu X \bullet ( & \forall i : I \bullet p[i] = m) \land \\
( & \parallel i : I \bullet p[i] := t) ; X \\
\neg ( & \forall i : I \bullet p[i] = m) \land \Box i : I \bullet p[i] = t \land t.i \rightarrow \\
& p[i] := m ; X \\
\end{align*}

The recursive process is not an action system, but can be transformed into one. First, we push the guard \( (\forall i : I \bullet p[i] = m) \) through the external and internal choices, and notice that \( g \& Stop \) is simply \( Stop \).

\begin{align*}
= \{ & \text{guard-external-choice-distribution, guarded-Stop} \} \\
array \ p : I \rightarrow PC \bullet
\begin{align*}
( & \parallel i : I \bullet p[i] := m) ; \\
\mu X \bullet ( & \forall i : I \bullet p[i] = m) \land ( \parallel i : I \bullet p[i] := t) ; X \\
\neg ( & \forall i : I \bullet p[i] = m) \land \Box i : I \bullet p[i] = t \land t.i \rightarrow \\
& p[i] := m ; X \\
\end{align*}
\end{align*}

Notice that the process \( (\Box i : I \bullet p[i] = t \land t.i \rightarrow p[i] := m) \), which is to be found in the second external choice, has the guard \( (\exists i : I \bullet p[i] = t) \). This says that there is an individual process waiting to do its independent transaction. But the outer guard says that every process is waiting to do
the multi-way synchronisation. These two guards are contradictory, so if we combine them, we get \( false \ & \ p[i] := m \), which may again be simplified to the deadlocked process \( Stop \).

\[
\begin{align*}
= & \{ \text{predicate calculus, guard-left-zero} \} \\
array \ p : I \rightarrow PC \bullet \\
& ( || i : I \bullet p[i] := m ) ; \\
& \mu X \bullet ( \forall i : I \bullet p[i] = m ) \ & ( || i : I \bullet p[i] := t ) ; \ X
\end{align*}
\]

(\( Stop \cap Stop \)) is, once again, simply \( Stop \), which is a unit for external choice.

\[
\begin{align*}
= & \{ \text{internal-choice-idempotence, external-choice-unit} \} \\
array \ p : I \rightarrow PC \bullet \\
& ( ||| i : I \bullet p[i] := m ) ; \\
& \mu X \bullet ( \forall i : I \bullet p[i] = m ) \ & ( ||| i : I \bullet p[i] := t ) ; \ X
\end{align*}
\]

The result of this transformation is now an action system; the second action’s guard can be simplified, using the existing reasoning about the guard.

\[
\begin{align*}
= & \{ \text{guard simplification} \}
\end{align*}
\]
\[
\text{array } \ p : I \rightarrow PC \cdot \\
( \|\| \ i : I \bullet p[i] := m ); \\
\mu X \bullet ( \forall i : I \bullet p[i] = m ) \& ( \|\| i : I \bullet p[i] := t ); \ X \\
\square \\
\square i : I \bullet p[i] = t \& t. i \rightarrow p[i] := m; \ X
\]

Notice that the initialisation sets all the elements of \( p \) to \( m \). This results in the first guard being enabled and the second being disabled; the first action then sets all the elements of \( p \) to \( t \); these two assignments can be merged.

\[
= \{ \text{ computation } \}
\]

\[
\text{array } \ p : I \rightarrow PC \cdot \\
( \|\| i : I \bullet p[i] := m ); \\
\left( ( \forall i : I \bullet p[i] = m ) \& ( \|\| i : I \bullet p[i] := t ); \ Z \right) \\
\square \\
\square i : I \bullet p[i] = t \& t. i \rightarrow p[i] := m; \ Z
\]

where

\[
Z \triangleq \mu X \bullet ( \forall i : I \bullet p[i] = m ) \& ( \|\| i : I \bullet p[i] := t ); \ X \\
\square \\
\square i : I \bullet p[i] = t \& t. i \rightarrow p[i] := m; \ X
\]

A useful assertion is based on a consequence of the simultaneous assignment:

\[
( ( \|\| i : I \bullet p[i] := m ) \Rightarrow ( \forall i : I \bullet p[i] = m ) )
\]

The assertion is designed to match the first guard and to contradict the second.

\[
= \{ \text{ assignment-assertion-introduction } \}
\]
```plaintext
array p : I → PC •
( ⊤ i : I • p[i] := m );
( ∀ i : I • p[i] = m )⊥;
( ∀ i : I • p[i] = m ) & ( ⊤ i : I • p[i] := t );  Z
□
□ i : I • p[i] = t & t.i → p[i] := m;  Z

where
Z ≡ µ X • ( ∀ i : I • p[i] = m ) & ( ⊤ i : I • p[i] := t );  X
□
□ i : I • p[i] = t & t.i → p[i] := m;  X

Now we can use the assertion to simplify the external choice. First, we distribute the assertion to the two branches of the choice. Next we simplify the resulting guards; the first becomes true and the second false.

= {  assertion-external-choice, predicate calculus, guard-unit, guard-zero  }

array p : I → PC •
( ⊤ i : I • p[i] := m );
( ∀ i : I • p[i] = m )⊥;
( ⊤ i : I • p[i] := t );
µ X • ( ∀ i : I • p[i] = m ) & ( ⊤ i : I • p[i] := t );  X
□
□ i : I • p[i] = t & t.i → p[i] := m;  X

The assertion can be absorbed back into the assignment, which may then be safely deleted, since it is followed by another assignment in sequence.

= {  assignment-assertion-introduction, assignment-sequence  }
```
array \ p : I \rightarrow PC \\
\quad ( \| i : I \bullet p[i] := t); \\
\quad \mu X \bullet (\forall i : I \bullet p[i] = m) \& (\| i : I \bullet p[i] := t); \ X \\
\quad \square \\
\quad \square i : I \bullet p[i] = t \& t.i \rightarrow p[i] := m; \ X

We can’t easily repeat this unfolding with the other guard, since its action is an external choice. However, we can exploit this asymmetry further by applying a law from the fixed-point calculus known as the diagonal rule, which allows us to separate the two recursive calls, naming them individually.

= \{ \ diagonal-rule \ \}

array \ p : I \rightarrow PC \\
\quad ( \| i : I \bullet p[i] := t); \\
\quad \mu Y \bullet \\
\quad \quad \mu X \bullet (\forall i : I \bullet p[i] = m) \& \\
\quad \quad \quad (\| i : I \bullet p[i] := t); \ X \\
\quad \quad \quad \square \\
\quad \quad \quad \square i : I \bullet p[i] = t \& t.i \rightarrow p[i] := m; \ X

Clearly, we could have applied the diagonal rule in another way; so why did we choose the first recursion? Because we can apply another fixed-point law: rolling. Notice that the recursion named Y is preceded by an assignment, and all recursive calls are also preceded by the same assignment.

= \{ \ rolling \ \}

array \ p : I \rightarrow PC \\
\quad ( \| i : I \bullet p[i] := t); \\
\quad \mu Y \bullet \\
\quad \quad (\| i : I \bullet p[i] := t); \\
\quad \quad \mu X \bullet (\forall i : I \bullet p[i] = m) \& Skip \\
\quad \quad \quad \square \\
\quad \quad \quad \quad i : I \bullet p[i] = t \& t.i \rightarrow p[i] := m; \ X

Now we’d like to localise the elements of \ p. First, we shift the declaration of
the array into the recursion, localising its scope.

\[
\begin{align*}
\{ \text{declaration-recursion} \} \\
\mu Y \cdot \left( \begin{array}{c}
\text{array } p : I \rightarrow PC \\
( || i : I \cdot p[i] := t); \\
\mu Y \cdot \mu X \cdot (\forall i : I \cdot p[i] = m) & \& \text{Skip} \\
\boxdot \\
\boxdot i : I \cdot p[i] = t & \& t.i \rightarrow p[i] := m; X \\
\end{array} \right) ; Y
\end{align*}
\]

The inner recursive process is actually an action system (actually, it is a degenerate kind of action system, with an action that is not event-guarded). More than this, it matches the right-hand side of the law that collapses a parallel combination of action systems into a single, sequential one. Using this observation as an inspiration, we start a series of transformations that mirror the ones at the start of the derivation, this time introducing parallelism.

\[
\begin{align*}
\{ \text{action-system-parallel} \} \\
\mu Y \cdot \left( \begin{array}{c}
\text{array } p : I \rightarrow PC \\
( || i : I \cdot p[i] := t); \\
\mu X \cdot \mu Y \cdot (\forall i : I \cdot p[i] = m) & \& \text{Skip} \\
\boxdot \\
p[i] = t & \& t.i \rightarrow p[i] := m; X \\
\end{array} \right) ; Y
\end{align*}
\]

We are heading for an application of the action system conversion law; before this is applicable, we must push the initialisation of each element of the array into the individual action systems.

\[
\{ \text{parallel-assignment} \}
\]
\[
\mu Y \bullet \left( \begin{array}{c}
\text{array } p : I \rightarrow PC \\
\| \| i : I \\
p[i] := t; \\
\mu X \bullet p[i] = m \& Skip \\
\square \\
p[i] = t \& t. i \rightarrow p[i] := m; \; X
\end{array} \right) ; \; Y
\]

Now we push the declaration of the array into the individual action systems.

\[
= \{ \text{parallel-state} \}
\mu Y \bullet \left( \begin{array}{c}
\| \| i : I \\
\begin{array}{c}
\text{var } p := t \\
\mu X \bullet p = m \& Skip \\
\square \\
p = t \& t. i \rightarrow p := m; \; X
\end{array}
\end{array} \right) ; \; Y
\]

At last, we can apply the action system conversion law.

\[
= \{ \text{action-system-conversion} \}
\mu Y \bullet ( \| \| i : I \bullet ( \mu X \bullet t. i \rightarrow Skip ) ) ; \; Y
\]

The inner-most recursion has now disappeared, and so we remove the innermost fixed-point operator, which is redundant.

\[
= \{ \text{fixed-point-constant} \}
\mu Y \bullet ( \| \| i : I \bullet t. i \rightarrow Skip ) ; \; Y
\]

And at last, we have reached our objective: a simple protocol that ensures that the transactions are synchronised, but that does not use an explicit multi-way synchronisation. It is now at the code-level of \textsc{occam}.

5 Conclusions and related work

We have shown how \textsc{Circus} can be used to describe and reason about safety-critical systems. In particular, we have shown how to use \textsc{Circus}'s refinement
calculus to develop a program that implements an abstract multi-way synchronisation using only pairwise synchronisations. The example is simple, but it is representative of our style of using calculation to develop reactive programs.

Our implementation actually retains a special kind of multi-way synchronisation: distributed co-termination of the interleaved processes. This can easily be avoided with a different development pattern.

A more interesting problem occurs when the multi-way synchronisation is part of an external choice, and our solution is not applicable in such a situation. We have calculated efficient two-phase commit protocols to deal with these kinds of synchronisation patterns. Although these programs are much more complex than the one calculated in this paper, the same development strategy is used. The abstract program is reduced to normal form, which, since it is sequential, contains no multi-way synchronisations. This normal form is then partitioned into new parallel processes that implement a protocol for synchronising individual transactions.

Our colleague, Peter Welch, has implemented a range of these protocols in occam and in JCSP, a Java class library for CSP/occam primitives. These implementations give a feel for the cost of implementing the multi-way abstraction. On a 1GHz commodity processor, it takes about 10µs to synchronise ten processes using the two-phase commit protocol.

Our work exploits the simple properties of CSP action systems, which are clearly related to those of Back and Kurki-Suonio [3]. Our work differs in that we are interested in rich state and reactive behaviour, using action systems as a convenient stepping-stone during transformations.

Acknowledgement

The author is grateful to the following: the Programme Committee of the Brazilian Workshop on Formal Methods, WMF’2003, for their invitation to present this paper; and to Formal Methods Europe for their financial support for the lecture. The Circus project is generously supported by QinetiQ Malvern and Bedford. The author has benefited from extensive technical discussions with Ana Cavalcanti, Alistair McEwan, Colin O’Halloran, and Peter Welch.

References


Some laws of Circus

This section contains a list of the laws used in the derivation. Each of these laws has been proved correct in the unifying theory of programming [16].

**Law: action-system-conversion**

\[
( \mu X \bullet x \rightarrow y \rightarrow X ) = \left( \begin{array}{l}
\text{var}\; p := x \bullet \\
\mu X \bullet p = x & p := y; X \\
\quad \square \\
p = y & p := x; X
\end{array} \right)
\]

**Law: action-system-parallel**
\[
\begin{align*}
\left( \|_{\{i\}} i : I \bullet \\
\mu X \bullet g_i \& x \to A_i; \ X \\
\bigg) = \\
\bigg( \big| \left( \forall i : I \bullet g_i \right) \& x \to \\
\bigg) &= \\
\bigg( \big| i : I \bullet A_i \big); \ X \\
\bigg)
\end{align*}
\]

**Law:** assertion-external-choice-distribution
\[
c_\bot; (P \square Q) = (c_\bot; P) \square (c_\bot; Q)
\]

**Law:** assignment-assertion-introduction
\[
(x := e) = (x := e; (x = e)_\bot)
\]

**Law:** assignment-sequence
\[
(x := e; x := f) = (x := f)
\]

**providing** \(x\) is not free in \(f\)

**Law:** computation
\[
\mu F = F(\mu F)
\]

**Law:** declaration-recursion
\[
\textit{var} \ x \bullet (\mu X \bullet x := e; F(X)) = \mu X \bullet (\textit{var} \ x := e; F(X))
\]

**Law:** fixed-point-constant
\[
(\mu X \bullet K) = K
\]

**providing** \(X\) is not free in \(K\)

**Law:** fixed-point-diagonal
\[
\mu X \bullet F(X, X) = \mu X \bullet (\mu Y \bullet F(X, Y))
\]

**Law:** fixed-point-rolling
\[
G(\mu X \bullet F(G(X))) = \mu X \bullet G(F(X))
\]
\textbf{Law: guard-external-choice-distribution} \\
\quad g \& (P \boxdot Q) = (g \& P) \boxdot (g \& Q)

\textbf{Law: guard-left-unit} \\
\quad \text{true} \& P = P

\textbf{Law: guard-left-zero} \\
\quad \text{false} \& P = \text{Stop}

\textbf{Law: guard-right-zero} \\
\quad g \& \text{Stop} = \text{Stop}

\textbf{Law: hiding-conditional-external-choice-distribution-2} \\
\begin{align*}
\left( b \& x \rightarrow P \right) \\Box \{x\} &= \left( b \& (P \\Box \{x\} \rightarrow Q \\Box \{x\}) \right) \\
\left( c \& y \rightarrow Q \right) \\Box \{x\} &= \left( \neg b \land c \& y \rightarrow Q \\Box \{x\} \right)
\end{align*}

\textbf{Law: hiding-distribution} \\
\begin{align*}
(\text{var} \ x \bullet P) \\backslash S &= (\text{var} \ x \bullet P \\backslash S) \\
(P ; Q) \\backslash S &= (P \\backslash S ; Q \\backslash S) \\
(\mu X \bullet F(X)) \\backslash S &= (\mu X \bullet F(X) \\backslash S) \\
(x := e) \\backslash S &= (x := e)
\end{align*}

\textbf{Law: internal-choice-idempotence} \\
\quad (P \sqcap P) = P

\textbf{Law: parallel-assignment} \\
\begin{align*}
(\parallel i : I \bullet (x[i] := e_i ; P_i)) &= (\parallel i : I \bullet x[i] := e_i) ; (\parallel i : I \bullet P_i)
\end{align*}

\textbf{Law: parallel-state} \\
\begin{align*}
\parallel i : I \bullet (\text{var} \ x : T \bullet P_i) &= \text{array} \ x : I \rightarrow T \bullet (\parallel i : I \bullet P_i[x[i]/x])
\end{align*}