Operational Semantics for Model Checking Circus

Jim Woodcock, Ana Cavalcanti, and Leonardo Freitas

Department of Computer Science
University of York, UK
{jim,alcc,leo}@cs.york.ac.uk

Abstract. *Circus* is a combination of Z, CSP, and the refinement calculus, and is based on Hoare & He's *Unifying Theories of Programming*. A model checker is being constructed for the language to conduct refinement checking in the style of FDR, but supported by theorem proving for reasoning about the complex states and data types that arise from the use of Z. FDR deals with bounded labelled transition systems (LTSs), but the *Circus* model checker manipulates LTSs with possibly infinite inscriptions on arcs and in nodes, and so, in general, the success or failure of a refinement check depends on interaction with a theorem prover. An LTS is generated from a source text using an operational interpretation of *Circus*; we present a Structured Operational Semantics for *Circus*, including both its process-algebraic and state-rich features.

1 Introduction

Circus [31, 32, 1, 23, 2, 3] is a state-rich process algebra based on Z [11, 33] and CSP [21], with a refinement calculus for deriving implementations from their specifications. Current work involves constructing a tool-set for supporting the language, including a theorem prover and a model checker. The development of the model checker is inspired by FDR, the model checker for CSP [19, 5]; however, a significant and novel aspect of the *Circus* model checker is the need to address the state-rich aspects of the language. The resulting procedure is refinement checking supported by theorem proving.

In its internal computations, FDR uses finite, labelled transition systems that are derived from source texts using the operational semantics of CSP. In order to construct the *Circus* model checker, we first need to explore the operational semantics of the language, including those state-based features not found in CSP. This leads to transition systems where the diagram is finite, but where the arcs and nodes may carry inscriptions involving infinite data types. This operational semantics must be proved congruent to the denotational semantics of *Circus*, which is different from the set-based presentation of the failures-divergences model used for CSP: it uses the unifying theories of programming (UTP) [10].

We present a Plotkin-style Structured Operational Semantics [17] for *Circus*, also based on UTP and using Z as a metalanguage [33], hence knowledge of Z is assumed. The operational semantics is inspired by the implementation of

CSP_M [24] in FDR [5], but has been adapted and extended to accommodate the state-rich features of *Circus*. The underlying automata theory and its properties have been formalised using the Z/Eves theorem prover [14].

In the next section, we give a brief overview of *Circus* and the UTP. Following this, we present the operational semantics for basic actions, declarations, synchronisation, schema expression, external choice, and interleaving; other operators are omitted for lack of space. In our final section, we discuss our results, put them in context, and look forward to future work.

2 Circus and the UTP

Circus is a forum for exploring the combination of process algebra and model-based abstract data types, and it is distinguished from similar combinations [4, 26] by being based firmly on the notion of refinement. Thus, the process algebra used is CSP and data types are specified in Z, since failure-divergences refinement and schema structuring have both proved their usefulness in describing industrial-scale development. Current experience is showing that when the refinement calculus is extended to include the operators of the process algebra, then its use also scales up to address large-scale architectural issues.

Unifying Theories of Programming [10] provides a single theoretical framework, based on an alphabetised relational calculus, that can be used for unification of many programming language paradigms. A theory in UTP is composed of an alphabet of names, a signature of language constructs, and a set of healthiness conditions. Programs, designs, and specifications are all interpreted as relations between an initial and an intermediate or final observation of behaviour. The following programming theories have all been modelled in the UTP: imperative, reactive, parallel, higher-order, and declarative [10, 30]; object oriented [6–8]; real-time [27]; and mobility [28, 29].

The semantics of *Circus* is defined in the UTP, where Z and the refinement calculus inhabit the theory of designs (pre-post specifications), and where CSP is the embedding of designs in the theory of reactive processes. Thus, everything that one might write in Z, CSP, or the refinement calculus may be freely mixed in a *Circus* specification. This is in contrast with other approaches, where the appropriate Z, VDM, or B specification is interpreted as a communicating abstract data type [4, 26]. The result is a rigid system architecture, which has its advantages: the abstract data type and the process algebra remain orthogonal throughout development, and so can be analysed separately using existing tools. It also has its disadvantages: every program that can be developed will have to adopt this architecture, and clearly many desirable programs do not.

A *Circus* program consists of a network of processes, each with encapsulated data and channels for communication and synchronisation. Within a process, there is a rich state with its attendant operations and process-algebraic behaviours, called *actions*; a distinguished main action defines the behaviour of the process. An action has no encapsulated data: it operates on a data space shared

with other actions inside the same process. Parallel composition defines partitions to avoid the usual problems of reasoning about shared data.

In the UTP denotational semantics, four kinds of observations may be made of *Circus* actions: (i) the *wait* variable distinguishes intermediate states from final ones; (ii) the *okay* variable distinguishes terminating states from non-terminating ones; (iii) the *tr* variables records the trace of past events; and (iv) the *ref* variable describes a set of events that are being refused by the process while it waits. For example, if a process has been started in a state where the events a and b have already occurred (in that order), and the process is waiting to perform the event c (but not a or b), then the following observations will hold: $okay \land \neg\ wait \land okay' \land wait' \land tr' = tr = \langle a, b \rangle \land ref' \subseteq \{a, b\}$. The components of the process state also appear in the alphabet.

Circus programs satisfy all the healthiness conditions for CSP processes found in [10], many of which are familiar from the failures-divergences semantics of CSP [9, 21, 25], and they form a complete lattice ordered by reverse implication. Thus, a process S is refined by another process I with the same alphabet $S \sqsubseteq I$, providing that $[\, S \Leftarrow I \,]$, where the brackets denote universal closure. In this paper, we provide an operational semantics for *Circus* actions.

3 Transition Relation

We define a transition relation capturing the operational semantics of *Circus*. An earlier abstract version has been formally mechanised in detail using the Z/Eves theorem prover [22]. All definitions and proof scripts that have guided our implementations are available from [34]. We introduce names, and well-formed Z expressions and predicates: *Name*, *ZExpr*, *ZPred*. Values and types are made from Z expressions: $Value == ZExpr$, and $Type == \mathbb{P}\, ZExpr$.

As usual, our transition relation connects one node to another using an arc; so it is a tertiary relation: $\mathbb{P}(Node \times Arc \times Node)$. The arcs in our relation are labelled with sets of events, and correspond to a communication permitted by a channel type definition; events range over the set Σ, and are channel-name/value pairs: $\Sigma == Name \times Value$ and $Arc == \mathbb{P}\, \Sigma$. These sets may be infinite. An empty arc represents a silent transition: either successful termination or internal progress. This is in contrast to the operational semantics of CSP_M, where two special events are used for silent transitions: ✓ (tick) and τ (tau). The former represents successful termination, whereas the latter represents internal progress. Deadlock in CSP_M is represented by lack of available events, and divergence is represented by an infinite sequence of τ's (a τ-loop in the automaton).

Nodes are configuration/environment pairs. The former contains the process's state and the action yet to be executed; the latter contains various declarations. The state $St \mathrel{\widehat=} ASt \land USt$ contains both observational of the UTP *ASt* and the process state *USt*. In the following definition, $Boolean ::= t \mid f$.

$$ASt \mathrel{\widehat=} [\, okay, wait : Boolean;\ tr : \mathrm{seq}\ \Sigma \,]$$

Refusals are not explicitly recorded; instead, they may be deduced from the outgoing arcs from each node. The schema $Obs \mathrel{\widehat=} [\, \Delta St \mid tr \ \mathsf{prefix}\ tr' \,]$ defines

all allowed observations between before and after states during the evaluation of the semantics; its invariant requires that no process can change the history of past events (a healthiness condition).

As said above, a configuration comprises a state and an action that remains to be executed: $Config == St \times Action$, where

$$
\begin{array}{rcl}
Action & ::= & \Omega \mid \mathsf{Skip} \mid \mathsf{Stop} \mid \mathsf{Chaos} \mid \mathsf{N} \mid \mu\ \mathsf{X} \bullet \mathsf{A} \mid \textbf{let}\ \mathsf{LocalEnv} \bullet \mathsf{A} \\
& \mid & \textbf{var}\ \mathsf{x} : \mathsf{T} \bullet \mathsf{A} \mid \mathsf{c} \rightarrow \mathsf{A} \mid \mathsf{c!v} \rightarrow \mathsf{A} \mid \mathsf{c?x} : \mathsf{P} \rightarrow \mathsf{A} \mid \mathsf{c?x} : \mathsf{P!e} \rightarrow \mathsf{A} \\
& \mid & \mathsf{A}\ ;\ \mathsf{B} \mid \mathsf{SExpr} \mid \mathsf{g}\ \&\ \mathsf{A} \mid \mathsf{A} \sqcap \mathsf{B} \mid \mathsf{A} \Box \mathsf{B} \mid \mathsf{A} \setminus \mathsf{hs} \mid \mathsf{A} \,[\![\mathsf{ns_0} \mid \mathsf{ns_1}]\!]\, \mathsf{B} \\
& \mid & \mathsf{A} \,[\![\mathsf{ns_0} \mid \mathsf{cs} \mid \mathsf{ns_1}]\!]\, \mathsf{B}
\end{array}
$$

Following ideas from CSP_M tools [18, 5], we have included an action Ω to denote a final configuration; it is not part of the user's syntax.

The declaration environment contains the names of channels, variables, actions, and unused names, which partition the given set of names.

Env

$$
\begin{array}{l}
chs, vars, acts, fresh : \mathbb{P}\, Name \\
cType, vType : Name \pfun Type \\
aCtx : Name \pfun Action \\
\hline
\langle chs, vars, acts, fresh \rangle\ \mathsf{partition}\ Name \\
\mathrm{dom}\, cType = chs \land \mathrm{dom}\, vType = vars \land \mathrm{dom}\, aCtx = acts
\end{array}
$$

Now we can define a node as a pair: $Node == Config \times Env$.

The declared type of a channel or variable name is determined by the functions *cType* and *vType*, respectively. The function *aCtx* records the syntax that is associated with an action name. Environments are updated to include new declarations; we give only the function for adding new channel declarations.

$$
\begin{array}{l}
cDecl : Env \times Name \times Type \pfun Env \\
\hline
\mathrm{dom}\, cDecl = \\
\quad = \{\, Env;\ N : Name;\ T : Type \mid N \in fresh \bullet (\theta Env, N, T) \,\} \\
\forall\, Env;\ N : Name;\ T : Type;\ A : Action \mid N \in fresh \bullet \\
\quad cDecl(\theta Env, N, T) \\
\qquad = \theta Env[cType := (cType \oplus \{\, N \mapsto T \,\}), fresh := (fresh \setminus \{\, N \,\})]
\end{array}
$$

In Z/Eves' syntax, substitution of expressions for variables is denoted by ":=", so the function updates exactly two components: *cType* and *fresh*. More generally, we give the semantics of a theta expression $\theta S[x := e]$ within a predicate P using existential quantification and standard renaming, provided y is fresh and e has the same type as x.

$$P\,(\theta S[x := e]) \equiv \exists\, y : \{\, e \,\} \bullet P(\theta S[y/x])$$

We define a transition system only for certain configurations: stable states are those in which *okay* is true ($Stable \mathrel{\widehat=} [\, St \mid okay = t \,]$); and normal states are stable states in which *wait* is false ($Normal \mathrel{\widehat=} [\, Stable \mid wait = f \,]$).

Two key functions in the definition of the operational semantics are *enabled*, which gives the set of enabled arcs for a node, and *arcStep*, which returns the set of nodes that can be reached from a given node by following a given arc. These functions are defined piecewise over the syntax of *Circus* actions. Their domains are defined as the nodes where the states are in normal configurations; the domain of *arcStep* insists that we are interested in stepping only through arcs that are enabled.

$$
\begin{array}{|l}
enabled : Node \pfun \mathbb{P}\, Arc \\
arcStep : Node \times Arc \pfun \mathbb{P}\, Node \\
\hline
\mathrm{dom}\, enabled = \{ \mathsf{A} : Action;\ Normal;\ Env \bullet ((\theta St, \mathsf{A}), \theta Env) \} \\
\mathrm{dom}\, arcStep = \{ \mathsf{A} : Action;\ a : Arc;\ Normal;\ Env \mid \\
\qquad\qquad a \in enabled((\theta St, \mathsf{A}), \theta Env) \bullet (((\theta St, \mathsf{A}), \theta Env), a) \}
\end{array}
$$

These functions abstractly define a general theory of automata, where the edges are sets of events (arcs) and the configurations are nodes. Therefore, the operational semantics of *Circus* is given in terms of these semantic functions for each available operator in the BNF syntax. As mentioned before, this is close to the operational semantics of CSP_M in FDR, where similar semantic functions named *inits* and *after* are defined.

There is a relationship between these two functions. The domain of *arcStep* is a relation (a set of pairs), which may be lifted to a set-valued function using relational image. This function is almost exactly *enabled*: we have to remove all pairs that *arcStep* would have mapped to the empty set, since these pairs can have no enabled arcs.

$$\forall\, n : Node \bullet enabled(n) = (\mathrm{dom}(arcStep \rhd \{\emptyset\}))(\!|\ \{n\}\ |\!)$$

The following well-formedness theorem is proved as a consequence of this relationship, and each definition below is proved to respect it.

Theorem 1 (Well-formedness). *An arc a is enabled in node n formed by an action A in a stable before state $(\theta St[okay := t])$ and an environment (θEnv) exactly when it is possible to reach at least one target node through n via a.*

$$
\begin{array}{l}
\forall\ St;\ Env;\ A : Action;\ a : Arc;\ n : Node \bullet \\
\qquad n = ((\theta St, \mathsf{A}), \theta Env) \land okay = t \Rightarrow a \in enabled\, n \Leftrightarrow arcStep(n, a) \neq \emptyset
\end{array}
$$

If the process diverges, the well-formedness theorem is no longer guaranteed.

We have proved this theorem for our underlying abstract automata theory, which is important for the implicit relationship between refusals sets and *enabled*.

An observation is stable whenever it has started ($okay$), and has not diverged ($okay'$). Valid observations are those where the trace history has been preserved (tr prefix tr'), as well as the state invariant. An observation is normal whenever its before state is normal and the after state is stable.

$$
\begin{array}{l}
StableObs \mathrel{\widehat=} Obs \land \Delta Stable \\
NormalObs \mathrel{\widehat=} StableObs \land Normal
\end{array}
$$

A stable observation can make progress initially ($okay \land \neg\, wait$), reach a stable

valid observation in an after state ($okay' \land wait' \land tr$ prefix tr'), but nothing is known about its termination yet ($wait'$ is unconstrained).

In the following sections, we define *enabled* and *arcStep* for a representative subset of *Circus* actions.

4 Basic Actions

Skip has only one possible behaviour—termination—so it has exactly one transition.

$$\forall\, Normal;\ Env \bullet enabled((\theta St, \mathsf{Skip}), \theta Env) = \{\, \emptyset \,\}$$

A *Terminating* observation is normal with $wait'$ false. Silent termination does not change tr. Read-only observations are normal.

$$\begin{array}{l} Terminating \mathrel{\widehat=} [\, NormalObs \mid wait' = f \,] \\ SilentlyTerminating \mathrel{\widehat=} [\, Terminating \mid tr' = tr \,] \\ ReadOnly \mathrel{\widehat=} [\, NormalObs \mid \Xi\, USt \,] \end{array}$$

Since the initial state in the semantics of Skip is normal, the empty arc leads to a final configuration with action Ω in an after state that silently terminates.

$$\begin{array}{l} \forall\, Normal;\ Env \bullet \\ \quad arcStep(((\theta St, \mathsf{Skip}), \theta Env), \emptyset) \\ \qquad = \{\, SilentlyTerminating;\ ReadOnly \bullet ((\theta St', \Omega), \theta Env) \,\} \end{array}$$

For the final configuration $(\theta St, \Omega)$ from a normal before state we have that

$$\forall\, Normal;\ Env \bullet enabled((\theta St, \Omega), \theta Env) = \emptyset$$

and since *enabled* gives the empty set of arcs, so the domain of *arcStep* for Ω is also empty. Finally, we observe the difference between an empty set being *enabled*, and *enabled* returning a singleton set containing just the empty set. The former is related to termination or internal progress; the latter is a final configuration with no outgoing arcs.

Stop represents a final action (Ω) in a waiting after state, where neither communication nor user state updates have happened. *Waiting* defines read-only observations where the after state is waiting for interaction ($wait'$). *SilentlyWaiting* defines waiting observations where no communication has taken place.

$$\begin{array}{l} Waiting \mathrel{\widehat=} [\, ReadOnly \mid wait' = t \,] \\ SilentlyWaiting \mathrel{\widehat=} [\, Waiting \mid tr' = tr \,] \end{array}$$

Like Skip, the definition of Stop also uses a silent transition through an empty

arc; the final state is given by *SilentlyWaiting* with the original environment.

$$\forall\, Normal;\ Env \bullet enabled((\theta St, \mathsf{Stop}), \theta Env) = \{\emptyset\}$$
$$\forall\, Normal;\ Env \bullet$$
$$arcStep(((\theta St, \mathsf{Stop}), \theta Env), \emptyset) = \{\, SilentlyWaiting \bullet ((\theta St', \Omega), \theta Env)\,\}$$

A deadlocked configuration accepts nothing, whereas a waiting configuration can progress whenever some arc becomes *enabled*. FDR has a similar representation.

Chaos has every possible behaviour, and this is represented as the power set of Σ.

$$\forall\, Normal;\ Env \bullet enabled((\theta St, \mathsf{Chaos}), \theta Env) = \mathbb{P}\ \Sigma$$

An observation is unpredictable whenever we move from a normal before state to an after state where only the minimal constraints hold. Leaving the value of $okay'$ unconstrained allows the possibility of divergence.

$$UnpredictableObs\ \mathrel{\widehat=}\ Normal \land Obs$$

The behaviour after any transition is not entirely arbitrary, even in the presence of divergence: the state invariant will continue to hold and the trace will not be corrupted (the minimal constraints). In the semantics of Chaos, each arc leads back to Chaos in an after state with these two constraints.

$$\forall\, Normal;\ Env;\ a : Arc \bullet$$
$$arcStep(((\theta St, \mathsf{Chaos}), \theta Env), a)$$
$$= \{\, UnpredictableObs \bullet ((\theta St', \mathsf{Chaos}), \theta Env)\,\}$$

Thus, divergence is characterised by an unstable after state ($okay'$ false) that *might* occur after an unpredictable observation. This is different from FDR, where divergence is recorded as a τ-loop in the transition system. These loops are detected by restricting the transition system to τ events, and then calculating the transitive closure [19], where the standard implementation is depth-first search (DFS). Research on a parallel version of FDR using graph pruning to detect divergence is under development [13].

An interesting side-effect of using the UTP characterisation of divergence might give an important performance improvement for the implementation of divergence detection, because no DFS is needed. Instead, a more efficient search such as parallel variations of breadth-first search (BFS) are being analysed. The outcome of this investigation and the parallel implementation of other model-checking algorithms are left as future work.

5 Channel declarations

Channel declarations are not permitted in *Circus* actions, but instead, they are evaluated during the contextual analysis that builds the initial environment. We define the syntax for declaration of channels and actions using a free-type *Decl*.

$$Decl ::= \mathbf{channel}\ \mathsf{N} : \mathsf{T} \mid \mathbf{channel}\ \mathsf{N} \mid \mathsf{N}\ \mathrel{\widehat=}\ \mathsf{A}$$

Next, the function *declare* is defined; it updates an original environment with a

given declaration of a channel.

$$\begin{array}{|l} declare : Env \times Decl \pfun Env \\ \hline \mathrm{dom}\, declare = \{\, Env;\ D : Decl;\ N : Name \mid N \in fresh \bullet (\theta Env, D) \,\} \end{array}$$

This function is partial, since some declarations might not be well-formed. Unlike in CSP, *Circus* channels are strongly typed; thus, a channel declaration includes the new channel name with its declared type in the given environment; it is defined using the function *cDecl* defined in Section 3.

$$\forall\, Env;\ N : Name;\ T : Type \bullet \\ declare(\theta Env, \mathbf{channel}\ \mathsf{N} : \mathsf{T}) = cDecl(\theta Env, N, T)$$

Events are formed from a channel name and a communicated value; but for synchronisation events, where no value is communicated, we define a special value *Synch*: it cannot be referred to by the user. Synchronisation channels are included in environments with the given name and the singleton type $\{\, Synch \,\}$.

$$\forall\, Env;\ N : Name \bullet declare(\theta Env, \mathbf{channel}\ \mathsf{N}) = cDecl(\theta Env, N, \{Synch\})$$

This allows a homogeneous declaration of channel types in the environment.

6 Input Prefixing: c?x : P → A

The enabled arcs of input prefixing contains events formed by the channel name and all values allowed by the declared channel type filtered by predicate P.

$$\forall\, Normal;\ Env \bullet \\ enabled((\theta St, \mathsf{c?x} : \mathsf{P} \to \mathsf{A}), \theta Env) = \{\, \{\, v : cType\ c \mid \mathsf{P} = t \bullet (c, v) \,\} \,\}$$

The transition for a synchronisation is defined using the schema *Communicating*, which requires that: (i) the before and after states are normal; (ii) the state invariant and the trace history are maintained; (iii) no modifications happen in the user state; (iv) the set of possible synchronisations includes the one in question; and (v) the after state trace is extended with the synchronisation event. $Progressing \mathrel{\widehat=} [\, ReadOnly \mid wait' = f \,]$ specifies (i)–(iii). The declaration of available events on input variable *given*?, and the selection of an event using output variable *e*! from *given*? specify (iv). The extension of *tr* specifies (v).

$$Communicating \mathrel{\widehat=} \\ [\, Progressing;\ given? : Arc;\ e! : \Sigma \mid e! \in given? \land tr' = tr \frown \langle e! \rangle \,]$$

These two definitions can now be used in the clause for *arcStep*.

$$\forall\, Normal;\ Env \bullet \\ \mathbf{let}\ allowed == \{\, v : cType\ c \mid \mathsf{P} = t \bullet (c, v) \,\} \bullet \\ arcStep(((\theta St, \mathsf{c?x} : \mathsf{P} \to \mathsf{A}), \theta Env), allowed) \\ = \{\, Communicating[given? := allowed] \bullet \\ \mathbf{let}\ lEnv == ((\mathsf{x}, \mathrm{ran}\ allowed), second\ e!) \bullet \\ ((\theta St', (\mathbf{let}\ lEnv \bullet \mathsf{A})), \theta Env) \,\}$$

The state is updated according to the *Communicating* schema where the given

events are those allowed; however, the communicated value must be available for the evaluation of the following action A. This is achieved by introducing a new local variable x implicitly declared through the special syntax (**let** *lEnv* • A), where the type is just that of c. Its value is that communicated: *second* e!.

The evaluation of the local environment for input communication is similar to that for variable declarations. The only difference is that the implicitly declared variable x must have a value from the communication that just took place.

$$\forall\, Normal;\ Env \bullet$$
$$enabled(\,(\,\theta St, (\,\textbf{let}\ \mathsf{lEnv} == ((\mathsf{x},\mathsf{T}),\mathsf{v}) \bullet \mathsf{A}\,)\,), \theta Env\,)$$
$$= enabled((\theta St, \mathsf{A}), vDecl(\theta Env, \mathsf{x}, \mathsf{T}))$$

The function *vDecl* extends the environment to include x; its definition is omitted for lack of space. For *arcStep*, we do something similar. We enrich the state in order to evaluate the action; removing the local variable from the environment afterwards ensures that the scope is indeed local. This is achieved using *vRemove*.

$$\forall\, Normal;\ Env;\ a : Arc \bullet$$
$$arcStep(((\theta St, (\textbf{let}\ \mathsf{lEnv} == ((\mathsf{x},\mathsf{T}),\mathsf{v}) \bullet \mathsf{A})), \theta Env), a)$$
$$= \textbf{let}\ ExtSt \mathrel{\widehat=} [\mathsf{x}, \mathsf{x}' : \mathsf{T} \mid \mathsf{x}' = \mathsf{x} = \mathsf{v}\,] \bullet$$
$$\{\, A' : Action;\ UnpredictableObs;\ Env' \mid$$
$$((\theta St', A'), \theta Env') \in$$
$$arcStep(((\theta(St \wedge ExtSt), \mathsf{A}), vDecl(\theta Env, \mathsf{x}, \mathsf{T})), a) \bullet$$
$$((\theta(St' \setminus (\mathsf{x}, \mathsf{x}')), (\textbf{let}\ lEnv == ((\mathsf{x},\mathsf{T}),\mathsf{x}') \bullet A'),$$
$$vRemove(\theta Env', \mathsf{x}))\,\}$$

Schemas cannot be written in **let** clauses as shown above. For clarity, we have used this notation, but in Z/Eves *ExtSt* has to be defined separately. In calculating the semantics of a *Circus* program, this results in a proliferation of small schemas that need to be introduced, and nested scope has to be eliminated in advance. This does not lead to problems when reasoning about the semantics.

7 Schema Expression: SExpr

Successful evaluation of schema expressions is represented with a silent transition via an empty arc *enabled*; however, as schema expressions can diverge if executed outside their preconditions, we allow any arc to be enabled.

$$\forall\, Normal;\ Env \bullet$$
$$enabled((\theta St, \mathsf{SExpr}), \theta Env) = \mathbb{P}\ \Sigma$$

Provided the precondition holds, a schema expression successfully terminates silently performing the operation in the user state. This leads to a final configuration that is terminating on the same environment.

$$\forall\, Normal;\ Env \mid \text{pre}\ \mathsf{SExpr} \bullet$$
$$arcStep(((\theta St, \mathsf{SExpr}), \theta Env), \emptyset)$$
$$= \{SilentlyTerminating \mid \mathsf{SExpr} \bullet ((\theta St', \Omega), \theta Env)\,\}$$

When the precondition does not hold, evaluation of schema expressions leads to

an after state with unpredictable observations, where the only guarantees are that the state invariant holds, and the trace history is not forgotten.

$$\begin{array}{l} \forall\, Normal;\ Env;\ a : Arc \mid \neg\ \mathrm{pre}\ \mathsf{SExpr} \bullet \\ \quad arcStep(((\theta St, \mathsf{SExpr}), \theta Env), a) \\ \quad\quad = \{\, UnpredictableObs \bullet ((\theta St', \mathsf{SExpr}), \theta Env) \,\} \end{array}$$

There is an implicit contextual analysis assumed on the unpredictable case. In order to calculate pre SExpr, both input (?) and output (!) variables on the schema expression must be in context in the given environment.

8 External Choice: A □ B

External choice has the arcs of both actions initially enabled. This includes empty arcs meaning either internal progress or termination, and visible communication on nonempty arcs.

$$\begin{array}{l} \forall\, Normal;\ Env;\ \mathsf{A}, \mathsf{B} : Action \bullet \\ \quad enabled((\theta St, \mathsf{A} \,\Box\, \mathsf{B}), \theta Env) \\ \quad\quad = enabled((\theta St, \mathsf{A}), \theta Env) \cup enabled((\theta St, \mathsf{B}), \theta Env) \end{array}$$

External choice is rather complex with respect to progress on the transition system. Intuitively, there are many cases to consider: visible communication, silent termination, internal progress, and the possibility of deadlock or divergence on either action, and deadlock on both actions. We analyse the cases separately.

Firstly, visible communication ($tr' \neq tr$) happens only when a prefixing is communicating. This communication represents the choice being resolved; it is formally defined by the schema *Choosing* as a normal observation that changes the trace. That is, from a normal before state ($okay \wedge \neg\ wait$) it reaches a stable after state ($okay'$) with valid observations (tr prefix tr'), where the trace has been extended ($tr' \neq tr$).

$$Choosing \ \widehat{=}\ [\, NormalObs \mid tr' \neq tr \,]$$

Whenever a visible communication happens, the choice is resolved to the following action that arises from either A or B. A first definition for *arcStep*, considering the case in which A is chosen, is as follows.

$$\begin{array}{l} \forall\, Normal;\ Env;\ a : Arc \bullet \\ \quad arcStep(((\theta St, \mathsf{A} \,\Box\, \mathsf{B}), \theta Env), a) \\ \quad\quad = \{\, C : Action;\ Choosing \mid \\ \quad\quad\quad ((\theta St', C), \theta Env) \in arcStep(((\theta St, \mathsf{A}), \theta Env), a) \bullet \\ \quad\quad\quad\quad ((\theta St', C), \theta Env) \,\} \cup \end{array}$$

That is, from a normal before state, A leads to C on a stable after state according to the schema *Choosing* on the same environment.

Silent termination on either action also resolves the choice. The difference is that silent termination always leads to the final action Ω.

$$\{ Terminating \mid ((\theta St', \Omega), \theta Env) \in arcStep(((\theta St, \mathsf{A}), \theta Env), a) \bullet ((\theta St', \Omega), \theta Env) \} \cup$$

This models the fact that termination cannot be refused. It is a direct consequence of the denotational semantics of *Circus*. This approach is also taken in Roscoe's CSP [21] and FDR [24, 5]. Alternatively, Hoare's CSP [9] forbids the choice of termination in an external choice, and Schneider's CSP [25] requires cooperation with the external environment when termination is offered in an external choice.

Now we consider internal progress in either action, say A again. Action $(\mathsf{A} \Box \mathsf{B})$ leads to $(\mathsf{A}' \Box \mathsf{B})$ in an after state that is ready for further progress $(\mathsf{A}' \neq \Omega)$, provided that A leads to A' in an after state as defined by the schema $SilentlyProgressing \mathrel{\widehat=} [\, Progressing \mid tr' = tr \,]$.

$$\{ A' : Action;\ SilentlyProgressing \mid A' \neq \Omega \land ((\theta St', A'), \theta Env) \in arcStep(((\theta St, \mathsf{A}), \theta Env), a) \bullet ((\theta St', \mathsf{A}' \Box \mathsf{B}), \theta Env) \} \cup$$

Internal (silent) progress happens on the resolution of internal choice, evaluation of variable declaration, action call, and so forth. Additionally, although after states observed due to internal progress are the same as those observed due to successful termination, the ambiguity is cleared because we insist that A' is different from final action Ω.

The possibility of deadlock in either action is defined next. Whenever action A leads to action A' in an after state that is silently waiting, deadlock might occur if the *enabled* arcs of A' is the empty set because A' is refusing every possible event.

$$\{ A' : Action;\ SilentlyWaiting \mid ((\theta St', A'), \theta Env) \in arcStep(((\theta St, \mathsf{A}), \theta Env), a) \bullet ((\theta St', \mathsf{A}' \Box \mathsf{B}), \theta Env) \} \cup$$

Whenever either action is already deadlocked $(\mathsf{Stop} \Box \mathsf{B})$, the choice is resolved to the remaining action, since the deadlocked action will have no arcs *enabled* (*arcStep* on the right-hand side is empty, and so the result is also empty). Of course, when both actions of the choice are deadlocked, so is the external choice.

Next, we need to consider divergence in either action, say A once more. If A leads to A' on an unpredictable after state, the external choice might be divergent; the result is A' in a possibly divergent state.

$$\{ A' : Action;\ UnpredictableObs \mid ((\theta St', A'), \theta Env) \in arcStep(((\theta St, \mathsf{A}), \theta Env), a) \bullet ((\theta St', \mathsf{A}'), \theta Env) \}$$

Putting all these cases together for both actions of the choice, we get the complete definition for *arcStep* of external choice.

9 Interleaving: $\mathsf{A} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \, \mathsf{B}$

Interleaving synchronises only on successful termination. In our semantics, successful termination happens whenever we reach a final configuration with Ω in an after state according to the observations of *Terminating*. An empty arc represents termination. Therefore, this is the only place where the original refusals (or acceptances) sets need to be readjusted. An empty arc can be allowed initially only if both actions are willing to terminate successfully. One possibility for the semantics is similar to that used in CSP_M [24].

$$\begin{array}{l}
\forall\, Normal;\ Env \bullet \\
\quad enabled((\theta St, \mathsf{A} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \, \mathsf{B}), \theta Env) \\
\quad\quad = (enabled((\theta St, \mathsf{A}), \theta Env) \cup enabled((\theta St, \mathsf{B}), \theta Env)) \setminus \{\, \emptyset \,\} \\
\quad\quad\quad \cup\, enabled((\theta St, \mathsf{A}), \theta Env) \cap enabled((\theta St, \mathsf{B}), \theta Env) \cap \{\, \emptyset \,\}
\end{array}$$

However, since an *enabled* empty arc also represents other transitions such as internal progress, this would wrongly enforce synchronisation in this case as well.

We cannot exploit the observational variables, as we do not yet know the possible after states of the enabling configuration for either action. Instead, the differentiation of these cases must be in the *arcStep* function. Soundness is guaranteed by the well-formedness theorem.

$$a \in enabled((\theta St, \mathsf{A}), \theta Env) \Leftrightarrow arcStep(((\theta St, \mathsf{A}), \theta Env), a) \neq \emptyset$$

Therefore, the enabled arcs of interleaving are those enabled on either action, and the distinction on distributed termination is left to *arcStep*.

$$\begin{array}{l}
\forall\, Normal;\ Env;\ \mathsf{A}, \mathsf{B} : Action \bullet \\
\quad enabled((\theta St, \mathsf{A} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \, \mathsf{B}), \theta Env) = \\
\quad\quad enabled((\theta St, \mathsf{A}), \theta Env) \cup enabled((\theta St, \mathsf{B}), \theta Env)
\end{array}$$

In the case of distributed termination, both A and B reach Ω through an empty arc in a final configuration with *Terminating* observations.

$$\begin{array}{l}
\forall\, Normal;\ Env;\ a : Arc \bullet \\
\quad arcStep(((\theta St, \mathsf{A} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \, \mathsf{B}), \theta Env), a) \\
\quad\quad = \{\, Terminating \mid \\
\quad\quad\quad ((\theta St', \Omega), \theta Env) \in (arcStep(((\theta St, \mathsf{A}), \theta Env), \emptyset) \\
\quad\quad\quad\quad\quad\quad \cap\, arcStep(((\theta St, \mathsf{B}), \theta Env), \emptyset)\,) \bullet \\
\quad\quad\quad\quad ((\theta St', \Omega), \theta Env) \,\}
\end{array}$$

In the absence of divergence, an enabled event is accepted by $(\mathsf{A} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \, \mathsf{B})$ whenever it is accepted by either A or B. We define the schema *Interleaving*0, which describes the observations allowed when A makes its independent progress; the schema *Interleaving*1 is defined similarly. We need three versions of the

state: (i) the before state shared by both actions and the resulting interleaving; (ii) the after state of the action being evaluated independently; and (iii) the after state of the interleaving.

$$
\begin{array}{|l}
\hline
\textit{Interleaving}0 \\
\textit{UnpredictableObs} \\
\textit{UnpredictableObs}[okay_0/okay', wait_0/wait', tr_0/tr', userVars_0/userVars'] \\
ns? : \mathbb{P}\, Name \\
\hline
okay' = okay_0 \land wait' = wait_0 \land tr' = tr_0 \\
\theta USt' = \theta USt \\
\hline
\end{array}
$$

UnpredictableObs describes valid observations (tr prefix tr') from a normal before state ($okay \land \neg\ wait$), where either divergence ($\neg\ okay'$), visible communication ($tr' \neq tr$), internal progress ($\neg\ wait' \land tr' = tr \land \Xi\ USt$), or waiting ($wait' \land tr' = tr \land \Xi\ USt$) are possible on the after sate.

After state variables of A are 0-subscripted to distinguish them from the after state variables of the interleaving. In *Interleaving*1, we use 1 as a subscript to distinguish the after state.

We have the case where independent progress is made on one action (say A). Action $(\mathsf{A} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \,\mathsf{B})$ reaches $(\mathsf{A'} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \,\mathsf{B})$ whenever A leads to A′ through the arc a.

$$
\begin{array}{l}
\forall\, Normal;\ Env;\ a : Arc \bullet \\
\quad arcStep(((\theta St, \mathsf{A} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \,\mathsf{B}), \theta Env), a) \\
\qquad = \{\, A' : Action;\ Interleaving0[\mathsf{ns_0}/ns?] \mid \\
\qquad\quad ((\theta St_0, A'), \theta Env) \in arcStep(((\theta St, \mathsf{A}), \theta Env), a) \bullet \\
\qquad\qquad ((\theta St', \mathsf{A'} \,|||[\, \mathsf{ns_0} \mid \mathsf{ns_1} \,]||| \,\mathsf{B}\ ), \theta Env) \,\}
\end{array}
$$

The effect on the after state of the interleaving is defined according to observations allowed by the *Interleaving* schema with appropriate substitution for the input name set.

10 Discussion

Our *Circus* model checker will permit the checking of certain kinds of infinite state processes using an algorithm inspired by FDR. We require the LTS to have a finite diagram bounded in size, but inscriptions on the nodes and arcs can involve infinite states and transitions. To see how our operational semantics compresses the graph of an LTS, consider the following two examples.

The process $c?x : \mathbb{N} \to SKIP$ communicates a natural number and then terminates: the CSP_M LTS branches infinitely; the *Circus* LTS has just a single arc to a node that is followed by termination.

The CSP_M process $P(i) = a!i \to P(i{+}1)$ outputs the natural numbers, start-

ing at i. The parametrised process $P(0)$ has an infinite number of nodes, each indexed with a natural number, and a long thin LTS with transitions between successor nodes. The state-based process $\mathbf{var}\ i := 0 \bullet (\ \mu\, X \bullet a!i \rightarrow i := i + 1;\ X\ )$ has the same behaviour, but without the infinite graph. ¿From its start node, the declaration enriches the environment with the variable i. From this node, there is a single transition labelled with the set of events $\{\ i : \mathbb{N} \bullet a.i\ \}$, followed by two transitions in sequence representing the assignment and recursive call. This makes a total of four nodes and four transitions. Both processes can be written in *Circus*, but the state-based style encourages the second.

The *Circus* model-checking algorithm tries to establish similarity between two LTSs: an implementation and its putative specification. It can confirm refinement or generate counterexamples for systems with modest data types, but in general it requires the proof of verification conditions to distinguish the outcome of model checking attempts and to compute counterexamples.

The verification conditions may be easily decidable, as would be the case when the programs involved are data independent in the sense of Lazić and Roscoe [21]. Other programs give rise to infinite state machines, but with bounded arcs, like the data flow example in [9]. In such cases, certain checks can be made with economical effort. For example, freedom from deadlock can be checked using the LTS, with the possibly infinite nodes giving rise to verification conditions that all partial functions have been applied within their domains (Z/Eves' *domain checks* [22]), which can be made automatic when appropriate preconditions are present in the Z specification.

Most model checking attempts fail, as a user debugs both the specification and the implementation, and we envisage a similar pattern with our tool. At first, many verification conditions are generated, which the user must scrutinise and judge. As the cycle of attempts continues, a pattern emerges, and similar verification conditions are generated in individual attempts. It is now worthwhile developing an appropriate theory and tuning its automation so that the stable set of verification conditions are discharged mechanically. New verification conditions appear in subsequent attempts, and most are dealt with by the theory. In this way, as the debugging converges to a correct refinement, the level of automation converges with it.

One of our guiding principles is to take our own medicine in building the tool, and so to develop crucial parts of the program using formal specification and refinement. Indeed, the formal model in UTP makes precise the connection between model checking and theorem proving, and this use of formalism is important for credibility as well as for soundness. The operational semantics presented in this paper is one of the departure points for the formal derivation of the algorithms used in the tool. Publication gives an opportunity for public scrutiny of the tool's development, as well as making its specification and algorithms available for other tool builders.

Our operational semantics is inspired by that for CSP_M used in FDR. Our most important contribution is the treatment of infinite constructions such as schema expressions and other state-related features of *Circus*. Our operational

and denotational semantics are presented in a uniform theoretical framework, making their proof of congruence much easier. Finally, the two operational semantics differ in various details, particularly to do with silent transitions, distributed termination, and divergence.

The two functions *enabled* and *arcStep* are used to define a transition relation between configurations (states and action pairs) $(s, P) \rightarrow (t, Q)$. As described in [10], this may be interpreted as saying that an implementation that is required to execute P in state s is permitted to execute the shorter action Q in state t. This gives us an independent correctness criterion for the operational semantics: $(s \ ; \ P)$ must be refined by $(t \ ; \ Q)$, where $(s \ ; \ P)$ is the program P started in state s. In this context, the state is represented by an assignemnt.

We have used the Z/Eves theorem prover to analyse the soundness of our description by proving refinement using the denotational semantics. This check for soundness is not yet complete, since there is no mature version of UTP embedded in a theorem prover yet. Nevertheless, research on this front is well advanced: the works in [15, 16] describe deep embeddings of the UTP in the theorem provers Z/Eves [14] and ProofPowerZ [12]. Eventually, this will enable us to mechanise the proof of the correctness of all of our operational semantics with respect to *Circus*'s denotational semantics.

Acknowledgements

We are grateful to QinetiQ Malvern for their long-term support of the *Circus* project, to the Royal Society for an Industry Fellowship, and jointly to the Universities of Kent and York for a *Circus* studentship. We are thankful to Peter Mosses for several illuminating discussions.

References

1. A.L.C. Cavalcanti, A.C.A. Sampaio, and J.C.P. Woodcock. Refinement of actions in *Circus*. *REFINE 2002. Electronic Notes in Theor. Comp. Sci.* **70**(3) 2002.
2. A.L.C. Cavalcanti, A.C.A. Sampaio, and J.C.P. Woodcock. A refinement strategy for *Circus*. *Formal Aspects of Computing* **15**(2–3):146–181 2003.
3. A.L.C. Cavalcanti and J.C.P. Woodcock. Predicate transformers in the semantics of *Circus*. *IEE Proceedings Software* **150**(2):85–94 2003.
4. C. Fischer. Combining CSP and Z. *Technical Report*. Univ. Oldenburg. 1996.
5. Michael Goldsmith. *FDR2 User's Manual version 2.67*. FSEL. May 2000.
6. He Jifeng, Zhiming Liu, and Xiaoshan Li. A Relational Model for Object-Oriented Programming. *Tech. Rep. 231*. UNU/IIST, P. O. Box 3058, Macau, May 2001.
7. He Jifeng, Zhiming Liu, and Xiaoshan Li. Towards a Refinement Calculus for Object Systems. *Procs ICCI2002* pp.69–77. IEEE Computer Society Press 2002.
8. He Jifeng, Zhiming Liu, and Xiaoshan Li. Modelling Object-oriented Programming with Reference Type and Dynamic Binding. *Tech. Rep.* **280**. UNU/IIST. 2003.
9. C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall 1985.
10. C.A.R. Hoare and He J. *Unifying Theories of Programming*. Prentice Hall 1998.

11. *Information Technology — Z Formal Specification Notation — Syntax, Type System and Semantics*. ISO/IEC 13568:2002.
12. Lemma-One. *ProofPower Tutorial*, 2003.
13. Jeremy M. R. Martin and Yvonne Huddart. Parallel Algorithms for Deadlock and Livelock Analysis of Concurrent Systems. *Commun. Proc. Archs.* IOS Press 2000.
14. Irwin Meisels and Mark Saaltink. *Z/Eves 1.5 Reference Manual. Technical Report TR-97-5493-03d.* ORA Canada, September 1997.
15. Gift Nuka and Jim Woodcock. Mechanising the alphabetised relational calculus. *WMF2003. Electronic Notes in Theoretical Computer Science* **95** 2004.
16. Marcel Oliveira, Ana Cavalcanti, and Jim Woodcock. Unifying theories in ProofPowerZ. *Draft.* University of York. January 2005.
17. G. D. Plotkin. A Structural approach to Operational Semantics. *Journal of Logic and Algebraic Programming* **60–61**:19–140 2004.
18. *ProBE User's Manual version 1.28*. Formal Systems (Europe) Ltd. May 2000.
19. A. W. Roscoe. Model Checking CSP. In [20] chapter 21 pp.353–378 1994.
20. A.W. Roscoe. *A Classsical Mind: Essays for C.A.R. Hoare.* Prentice Hall 1994.
21. A. W. Roscoe. *Theory and Practice of Concurrency.* Prentice Hall 1997.
22. Mark Saaltink. Z/Eves 2.0 User's Guide. *Technical Report TR-99-5493-06a.* ORA Canada 1999.
23. A.C.A. Sampaio, J.C.P. Woodcock, and A.L.C. Cavalcanti. Refinement in *Circus*. *FME 2002 Lecture Notes in Computer Science* **2391**:451–470 2002.
24. B. Scattergood. *The Semantics and Implementation of Machine Readable CSP*. PhD thesis. Oxford University 1998.
25. S. Schneider. *Concurrent and Real-Time Systems: The CSP Approach.* Wiley 2000.
26. S. Schneider and H. Treharne. Communicating B Machines. *ZB2002. Lecture Notes in Computer Science* **2272**:415–435. 2002.
27. Adnan Sherif and He Jifeng. Toward a Time Model for *Circus*. *ICFEM 2002. Lecture Notes in Computer Science* **2495** pp.613–624. Springer-Verlag.
28. Xinbei Tang and Jim Woodcock. Towards mobile processes in unifying theories. *SEFM 2004*. IEEE Computer Society 2004.
29. Xinbei Tang and Jim Woodcock. Travelling processes. *Mathematics of Program Construction. Lecture Notes in Computer Science* **3125**:381–399 2004.
30. J. C. P. Woodcock. Unifying Theories of Parallel Programming. In *Logic and Algebra for Engineering Software*. IOS Press, 2002.
31. Jim Woodcock and Ana Cavalcanti. A Concurrent Language for Refinement. *5th Irish Workshop on Formal Methods*, 2001.
32. Jim Woodcock and Ana Cavalcanti. The Semantics of *Circus*. *ZB 2002. Lecture Notes in Computer Science*:184–203 Springer-Verlag 2002.
33. Jim Woodcock and Jim Davies. *Using Z: Specification, Refinement, and Proof.* Prentice Hall 1996.
34. `www-users.cs.york.ac.uk/~leo`.