A Sound Operational Semantics for Circus

S. L. M. Barrocas\textsuperscript{1} and M. V. M. Oliveira\textsuperscript{1,2}

\textsuperscript{1}Departamento de Informática e Matemática Aplicada, Universidade Federal do Rio Grande do Norte
\textsuperscript{2}Instituto Metrópole Digital, Universidade Federal do Rio Grande do Norte
Natal, RN, Brazil
Contact: marcel@dimap.ufrn.br

Abstract—The use of formal methods in software engineering considerably reduces the number of errors throughout system developments by enforcing a rigorous specification and verification before reaching a final implementation. We provide, on this paper, a full and sound Structural Operational Semantics for Circus, a formal notation that combines Z and CSP. Our work lifts the works of Freitas, Cavalcanti and Woodcock by creating rules that deal with any Circus construct. We also provide, on this paper, proof of soundness of the Structural Operational Semantics with the Unifying Theories of Programming (UTP).

Keywords: Operational Semantics, Circus, Soundness

1. Introduction

Formal methods are techniques that specify systems using formal notations underpinned by rigourously defined semantics. Due to the effort required to apply such techniques, their application usually become expensive. For this reason, their use have normally been justified only in the implementation of concurrent and safety-critical systems.

Circus\textsuperscript{17} is a specification language whose syntax combines the syntaxes of Z\textsuperscript{18} and CSP\textsuperscript{15}. This feature of Circus allows the representation of concurrent systems with large amount of data in a non-implicit fashion. Circus has a refinement calculus\textsuperscript{14} with transformation rules that can be used to refine Circus abstract specifications into Circus concrete implementations. Circus has been provided with both, a denotational semantics\textsuperscript{9}, and also an operational semantics\textsuperscript{9} that contains rules that can be applied to generate labelled predicate transition systems (LPTS) of a given specification. The Operational Semantics shown at\textsuperscript{9} is the first version of the Operational Semantics of Circus: it was presented having its rules defined using Z Schemas. Later, Woodcock and Cavalcanti have developed an updated Structural Operational Semantics using different definitions and a different description, envisaging to provide a better automation (using an automatic theorem prover). The rules were developed for a formalism that has similar constructs to Circus: CML. These rules are shown at\textsuperscript{5}. Similar rules are shown in\textsuperscript{4}. These works, however, show rules for Circus actions and basic processes, and Circus has other constructs for processes (compound processes, call processes and etc).

This paper aims at providing a full and sound Structural Operational Semantics for Circus. This will be achieved by lifting the Operational Semantics based on ideas described in\textsuperscript{4},\textsuperscript{5},\textsuperscript{9},\textsuperscript{14}, and then proving the soundness of the unproved laws with respect to the UTP\textsuperscript{10}.

2. Circus

Concurrent and Integrated Refinement CalculUS (\textit{Circus})\textsuperscript{17},\textsuperscript{8},\textsuperscript{9},\textsuperscript{14} is a formal language whose syntax is based on the syntaxes of two other formal languages, Z\textsuperscript{18} and CSP\textsuperscript{12}. Circus joins Z’s feature of representing complex data structures with CSP’s process algebra, that represents concurrency. Circus also has a refinement calculus\textsuperscript{14}, and its syntax is based on Dijkstra’s language of guarded commands\textsuperscript{6}.

A Circus program is formed by zero or more paragraphs, each of which can be a declaration of channels or a channel set, Z Paragraphs\textsuperscript{16} or Process paragraphs. When it is a process paragraph, it can be a compound (Parallel, Interleave, External Choice, Internal Choice or Sequence) process, an unary process (Hide, Rename, Parameterised), a Call Process (with parameters, which can be indexed or normal), an Iterated Process (that are a generalization of compound processes), or a basic process, that has a state (possibly with variables), inner actions (tasks that the process can perform) and a main action (that defines the behaviour of the process). Each action can be a command (Assignment, Variable Block, Specification Statement, If-Guarded or a Substitution command), a compound action (using the same operators of compound process), an unary action (Hide, Rename, Parameterised, Guarded or Prefixing), or a basic action (SKIP, STOP or CHAOS). The syntax of Circus is described on\textsuperscript{14}.

We will give an example of Circus specification of a simple Cash Machine that allows cash withdrawal (if there is sufficient balance) and balance inquiry. The Cash Machine interacts with a user. It will be specified as the following
process:

channel cash, inquiry
channel entercnum,chosencash, amount : N
channel displaymoney, displaybalance : N
channel set CS \(\equiv\) \{cnum, inquire, entercnum, amount \}

process CashMachine \(\equiv\) begin
state St \(\equiv\) \{b : seq N \}
INQUIRY \(\equiv\) x : N \(\bullet\) displaybalance \((b(x)) \rightarrow\) SKIP
CASH \(\equiv\) x : N \(\bullet\) amount? y \(\rightarrow\)
if \((b(x) \geq y) \rightarrow b := b + \{x \mapsto b(x) - y\} ;\)
displaymoney \(\rightarrow\) SKIP
\{(b(x) < y) \rightarrow\) SKIP
fi
\bullet \(\mu\) X \(\bullet\) entercnum? x \(\rightarrow\)
((inquiry \rightarrow INQUIRY(x)) \(\bigcirc\) (cash \rightarrow CASH(x))); X
end

process Customer \(\equiv\) begin
state St \(\equiv\) \{mycash, cnum : N \}
INIT \(\equiv\) cnum := 2; mycash := 50 ;
entercnum! cnum \rightarrow\) SKIP
INQUIRYOP \(\equiv\) inquiry \rightarrow SKIP
CASHOP \(\equiv\) cash \rightarrow amount? mycash \rightarrow\) SKIP
\bullet INIT ;
\bullet \(\mu\) X \(\bullet\) (CASHOP \(\bigcirc\) INQUIRYOP) ; X
end

process System \(\equiv\) (CashMachine \(\parallel\) CS \(\parallel\) Customer) \(\setminus\) CS

The process System consists on the parallel composition between the process CashMachine and the process Customer. Thus, System is a Parallel Process with a hidden set of channels. The parallel composition between System and CashMachine will have its synchronisation on channels cash, inquiry, entercnum, amount. The channels that compose the parallel interface of System are hidden from the external environment (\(\setminus\) CS). Customer will initially store a card number (with the assignment command cnum := 2) and an amount to be cashed (with another assignment command mycash := 50) through action INIT. INIT then writes (!) the card number (cnum) on channel entercnum (entercnum! cnum). Then it will randomly (using the Internal Choice \(\bigcirc\) operator) choose between cashing (CASHOP) or requiring balance information (INQUIRYOP). If Customer chooses to require balance, it will perform event inquiry. If it chooses to cash, it will perform cash and then write (!) its desired value to be cashed (mycash) on channel amount (amount! mycash). After choosing between requiring balance or cashing, it recurs (invoking X). When it recurs, it goes to \(\mu\) X.

The process CashMachine initially inputs a card number (cnum) on channel entercnum, then it offers two choices (\(\bigcirc\)): one in which the user will see balance (in which the choice is resolved by performing inquiry), and the other in which the user will cash out money (in which the choice is resolved by performing the cash event). The amount to be cashed will be input (?) on channel amount and the value will be stored on variable y, and then if the chosen amount is bigger than the balance of the Customer \((b(x) \geq y)\), the CashMachine will subtract the balance \((b := b + \{x \mapsto b(x) - y\})\) of the user and display the money on displaybalance channel. If \((b(x) < y)\), then it terminates successfully (SKIP). After seeing balance or cashing money, CashMachine recurs.

The "\(\rightarrow\)" operator is used between a channel invocation (possibly followed by communication fields) and an action, and it establishes a Prefixing between the channel and the action (that is, the action will only be performed if an event on the channel occurs). The Sequence Operator (;) establishes a similar relation, but between two actions.

Circus has a theory on the Unifying Theories of Programming (UTP) [10], that is a framework that provides constructs for defining theories for formalisms. This theory of Circus is its Denotational Semantics, which will be explained as follows.

### 2.1 Circus Denotational Semantics

The Denotational Semantics of Circus [14] defines each construct of Circus as an UTP reactive design of the form \(R\ (Pre \vdash Post)\), in which the predicates Pre and Post are defined using UTP variables \((tr\ for\ trace,\ ref\ for\ refusals,\ ok\ for\ program\ ready\ to\ start\ and\ wait\ for\ program\ waiting)\) and their dashed versions \((tr',\ ref',\ ok'\ and\ wait')\). For any variable \(x\) on the UTP, \(x\) represents a variable on its current state, and \(x'\) represents a variable on a future state. The meaning of the design is:

\[
Pre \vdash Post \equiv Pre \land ok \Rightarrow Post \land ok'
\]

The expression above means: if the program has started having its pre-conditions holded, than when it finishes it will have its post-condition holded. The \(R\) on the expression is a function that guarantees soundness of the Denotational Expression with the UTP. This function is called Healthiness Condition. It is defined in terms of other 3 healthiness conditions that Circus has:

\[
R\ (P) \equiv R_1 \circ R_2 \circ R_3\ (P)
\]

\(R_1,\ R_2\) and \(R_3\) will have their definitions shown as follows:

\[
R_1\ (P) \equiv P \land tr \leq tr'
\]

\[
R_2\ (P\ (tr, tr')) \equiv P\ (\lt\!,\ tr' - tr)
\]

\[
R_3\ (P) \equiv H_{rea} \lt\!,\ wait \triangleright P
\]
• The healthiness condition \( R_1 \) says that the history of events cannot be undone. This is established by the expression \( tr \leq tr' \). On this expression, \( tr \) represents the trace in the current state and \( tr' \) represents the trace in a future state. The trace in the future state cannot be less than the trace in the current state;

• \( R_2 \) says that the behaviour of the process independes from what happened before. The expression of \( R_2 \) can be re-written as:

\[
R_2 (P (tr, tr')) \equiv P [<>, tr' - tr / tr, tr']
\]

The expression above replaces, on the predicate \( P, tr \) by the empty trace \((<>\)) and \( tr' \) by \( tr' - tr \). Thus, the history of events on \( tr \) is irrelevant for the behaviour of the process;

• \( R_3 \) says that processes that wait for other processes to finish must not start. This is represented by the conditional expression \( II_{\text{rea}} \triangleleft \text{wait} \triangleright P \). This expression says: if \( \text{wait} \) is true, then the conditional expression equals the reactive skip \((II_{\text{rea}})\). Otherwise it equals \( P \). The reactive skip is defined as follows:

\[
II_{\text{rea}} \equiv (\neg \text{ok} \land tr' \leq tr) \lor (ok' \land tr' = tr \land \text{wait'} = \text{wait} \land \text{ref'} = \text{ref} \land v' = v)
\]

The expression above stands for: The process either did not start (having its history of events not undone) or has finished \((ok')\) having its UTP variables unchanged. Thus the behaviour of \( R_3 \) is: if the process is waiting for other processes to finish, then it either did not start or has already finished. Otherwise it continues;

An important operator from the UTP is the Sequence Operator. It is described as the following expression:

\[
P; Q \equiv \exists v_0 . P (v, v_0) \land Q (v_0, v')
\]

The definition of the reactive designs for the constructs of Circus can be seen at [14].

The Denotational definition of each Circus construct is efficient for expressing their conditions, but it does not explicitly express the operational behaviour of each Circus construct. We will show some concepts of the Operational Semantics of Circus on the following sub-section.

### 2.2 Circus’ Operational Semantics

The Operational Semantics of Circus was firstly described on [9]. Cavalcanti and Gaudel [4], [11] described an updated Operational Semantics for Circus. It shows different kinds of definitions for both labelled and silent transitions, in which the nodes contain a constraint, that indicates if that node is enabled or not. On this paper, the node is represented by a triple \((c \mid s \models A)\), where \( c \) is the constraint, \( s \) is the sequence of assignments and \( A \) is the program text (in this case, an action) that remains to be executed. A similar kind of description is shown on Woodcock’s technical report [5], where the Operational Semantics of a language that has similar constructs to Circus (CML) is shown and explained. The denotational definition of each rule of this Semantics can be seen as follows:

**Definition 1**: Silent Transition

\[
(c_1 \mid s_1 \models A_1) \rightarrow (c_2 \mid s_2 \models A_2)
\]

= \( \forall w . c_1 \land c_2 \Rightarrow \text{Lift} (s_1) ; A_1 \sqsubseteq \text{Lift} (s_2) ; A_2 \)

**Definition 2**: Labelled Transition

\[
(c_1 \mid s_1 \models A_1) \rightarrow (c_2 \mid s_2 \models A_2)
\]

= \( \forall w . c_1 \land c_2 \Rightarrow \text{Lift} (s_1) ; A_1 \sqsubseteq (\text{Lift} (s_2); c.w1 \rightarrow A_2) \sqsubseteq (\text{Lift} (s_1) ; A_1) \)

where

\[
\text{Lift} (s) = R_1 \circ R_3 (\text{true} \vdash s \land tr' = tr \land \neg \text{wait'})
\]

Definition 1 gives the denotational meaning of a Silent Transition. It means: for all loose constants \( w \), if \( c_1 \) and \( c_2 \) are true, then the left side of the transition is refined by the right side of the transition. The meaning of the denotational definition on 2 is: if \( c_1 \) and \( c_2 \) are true, then the left side of the transition is refined by the external choice between the right side of the transition prefixed by the label and the left side of the transition.

The Lift function assures that the assignment is healthy with respect to the theory of Circus on the UTP. It applies healthiness conditions \( R_1 \) and \( R_3 \) to the assignment put as parameter with no restriction \((\text{true} \) on the pre-condition of the design), and assures that the assignment does not change the history of events \((tr' = tr)\) and that the program will not be in a waiting state \((\neg \text{wait'})\).
An example of Operational description can be seen for action INQUIRYOP of process Customer as follows:

\[(true \mid \{} \models in\text{quiry} \rightarrow \text{SKIP}) \xrightarrow{in\text{quiry}} (true \mid \{} \models \text{SKIP})\]

The above description consists on a transition between two nodes. The first node has in\text{quiry} \rightarrow \text{SKIP} as the program text that remains to be executed, and true as constraint (that is, there is no restriction on this node). The labelled arc in\text{quiry} then establishes that, when event in\text{quiry} occurs, the program goes to the node (true \mid \{} \models \text{SKIP}). As there are no arcs going out from the node containing SKIP, then the program terminates.

### 2.2.1 Extending and Lifting the Operational Semantics to Circus processes

Among the constructs that lie on Circus’ syntax, all of them refer to a subset of Circus actions. Woodcock defined rules for Prefixing, Assignment, External Choice, Internal Choice, Parallelism, Interleaving, Guards, Hiding and Recursion, all of them for actions. For this paper, we created new rules for the remaining constructs of Circus actions (If-guarded-command, Specification Statements, Alphabetised Parallel Action, Parameter Action Call, Iterated Actions, and etc), and lifted the semantics to Circus processes by creating a new kind of transition, the syntactic transition, whose denotational definition is seen as follows:

**Definition 3: Syntactic Transition**

\[(c_1 \models P_1) \xrightarrow{\sigma} (c_2 \models P_2) = \]

\[
\forall w. c_1 \land c_2 \Rightarrow ((\text{Lift} (gA (P_1))) \; ; \; P_1 \sqsupset \; \text{Lift} (gA (P_2));
\text{Lift} (gA (P_1))) = \text{Lift} (gA (P_2))\]

getAssignments (abbreviated as gA) is an auxiliary function that calculates the sequence of assignments of a node whose program text is a process. A node whose program text is a process is defined by a constraint and a process, represented as (c \models P). The transition \((c_1 \models P_1) \xrightarrow{\sigma} (c_2 \models P_2)\) means that, if \(c_1\) and \(c_2\) are true, then \(P_1\) can be syntactically transformed to \(P_2\) without semantically changing the program. The Syntactic Transition, thus, does not specify a path of computation on the program. The idea is to syntactically transform the process in order to reach a Basic Process. During this transformation, inner assignments on the program cannot change (\(\text{Lift} (\text{getAssignments} (P_1)) = \text{Lift} (\text{getAssignments} (P_2))\)) and the assignments have to preserve healthiness.

One of the constructs that Woodcock did not encompass was If-Guarded-Command. We created a set of rules that describe the behaviour of If-Guarded-Command, among other constructs. We will show its rules as an example:

#### Rule 1: If-Guarded Command:

Be

\[\text{IGC} \triangleq \text{if} (\text{pred}_1) \rightarrow A_1 \parallel (\text{pred}_2) \rightarrow A_2 \parallel ... \parallel (\text{pred}_n) \rightarrow A_n \text{ fi}, \text{ then}\]

\[(c \mid s \models \text{IGC}) \xrightarrow{\tau} (c \land \text{pred}_1 \mid s \models A_1)
(c \mid s \models \text{IGC}) \xrightarrow{\tau} (c \land \text{pred}_2 \mid s \models A_2)
...\]

\[(c \mid s \models \text{IGC}) \xrightarrow{\tau} (c \land \text{pred}_n \mid s \models A_n)
(c \mid s \models \text{IGC}) \xrightarrow{\tau} (c \land \lnot \text{pred}_1 \land ... \land \lnot \text{pred}_n \mid s \models \text{CHAOS})\]

The set of rules 1 are for If-Guarded Commands. Each rule shows a possibility of computation for the program text, each one depending on a predicate from the If-Guarded Command. There is a possibility in which the command goes to the node whose program text is \(A_1\) and whose constraint is \(c \land \text{pred}_1\) (indicating that \(A_1\) is reachable only if \(\text{pred}_1\) is true), and so on. When all guards are false, the If-Guarded Command diverges (with CHAOS), and when more than one guard is true, the If-Guarded Command behaves as an internal choice between the true-guarded branches.

We will also give an example of rule, created for this paper, that can be applied to any compound (Parallel, Interleave, External Choice, Internal Choice and Sequence) process:

#### Rule 2: Compound Process Left:

\[(c_1 \models P_1 \parallel P_2) \xrightarrow{\tau} (c_2 \models P_3)\]

\[(c_1 \models P_1 \parallel P_2) \xrightarrow{\tau} (c_2 \models P_3 \parallel P_2)\]

Where \(\text{OP} \in \{\cap, \text{||}, \text{;}, \text{||}, \text{||}, \text{CS} \} \}

Rule 2 describes a syntactic transformation by syntactically advancing the left branch of the compound operator. It says:

- if \((c_1 \models P_1)\) can be syntactically transformed to \((c_2 \models P_3)\),
- so \((c_1 \models P_1 \parallel P_2)\) can be syntactically transformed to \((c_2 \models P_3 \parallel P_2)\).

### 3. Soundness with respect to the UTP

We provided, for this paper, the proof of soundness with respect to the UTP for all rules of the Operational Semantics of Circus. Each rule was proved using theorems from Woodcock’s technical report [5] and lemmas created for this
paper. All proofs and referenced lemmas are shown on [2]. Each proof is divided in sub-goals (intermediate expressions that compose the proof), and tactics between sub-goals (tactics can be composed by lemmas and assumptions that are used to justify the advancement from a sub-goal to another). Each tactic has a source sub-goal and a destination sub-goal. Sometimes tactics will be omitted. On these cases, transformations between the sub-goals will consist only on predicate calculus. The first sub-goal is the expression to be proved. The last sub-goal (which is the goal of the proof) will mandatorily be true. In order to modularize the proof, lemmas were created and proved as well. We will also provide explanation for each step of the proof. Each tactic will be shown throughout the proof as an explanation with parenthesis. Proofs will be shown on the following format:

Sub-goal 1
(explanation 1)
...
= Sub-goal n
(explanation n)

We will show and explain, step by step, the proof of soundness of the first rule of 1, which is

\( (c \mid s \models \text{IGC}) \rightarrow (c \land \text{pred}_1 \mid s \models A_1) \)

The first step for proving this law is creating a lemma to prove that IGC is refined by \( A_1 \) (IGC \( \sqsubseteq A_1 \)), provided that \( \text{pred}_1 \) is true. This lemma will be used throughout the proof:

Lemma 1: IGC \( \sqsubseteq A_1 \), provided that \( \text{pred}_1 \)

Proof:

Be \( \text{EXPR} = (\text{if} (\text{pred}_2) \rightarrow A_2)[…](\text{pred}_n) \rightarrow A_n, \text{fi} \), then

\( \text{IGC} \sqsubseteq A_1 \)

(the first step of the proof is to infer that, as \( \text{pred}_1 \) is true, IGC either equals \( A_1 \) (if \( \text{pred}_1 \) is true and all other \( \text{pred}_i \) are false) or an internal choice between \( A_1 \) and another expression - if \( \text{pred}_1 \) is true and at least one predicate \( \text{pred}_i \) is true; for IGC = \( A_1 \), the proof is trivial: The expression becomes a refinement between equal sides - \( A_1 \sqsubseteq A_1 \), which is true. For IGC = \( A_1 \cap \text{EXPR} \), the proof evolves as we will show)

\( = A_1 \cap \text{EXPR} \sqsubseteq A_1 \)

(in the UTP, an internal choice between two actions is an OR composition between the denotational definitions of both actions)

\( = A_1 \lor \text{EXPR} \sqsubseteq A_1 \)

(a refinement expression \( A \sqsubseteq B \) can be written as \( B \Rightarrow A \), where the brackets means universal quantification on the variables of the expression. As, on a proof, the expression under proof is proved for all possible values of variables, then brackets can be omitted)

\( = A_1 \Rightarrow A_1 \lor \text{EXPR} \)

(from here below, only predicate calculus until reaching true as final sub-goal)

\( = \neg A_1 \lor (A_1 \lor \text{EXPR}) \)

\( = \neg A_1 \lor A_1 \lor \text{EXPR} \)

\( = \text{true} \lor \text{EXPR} \)

\( = \text{true} \)

Now we will prove the following expression:

\( (c \mid s \models \text{IGC}) \rightarrow (c \land \text{pred}_1 \mid s \models A_1) \)

Proof:

\( (c \mid s \models \text{IGC}) \rightarrow (c \land \text{pred}_1 \mid s \models A_1) \)

(we firstly had to apply a tactic with the definition 1 of Silent Transition to transform the expression that we want to prove into a First Order Logic expression)

\( = \forall w . c \land c \land \text{pred}_1 \)

\( \Rightarrow (\text{Lift} (s) \land \text{IGC} \sqsubseteq \text{Lift} (s) \land A_1) \)

(then we applied a tactic that combined lemma 1, the assumption \( \text{pred}_1 \) and Monotonicity of Refinement in order to prove that the refinement expression was true; if \( \text{Lift} (s) \sqsubseteq \text{Lift} (s) \) and IGC \( \sqsubseteq A_1 \), by 1, and (assms) \( \text{pred}_1 \), \( \text{pred}_1 \) is true because it appears on the left side of the implication).

\( = \forall w . c \land c \land \text{pred}_1 \Rightarrow \text{true} \)

(then two Predicate Calculus tactics were applied and finally the final sub-goal (true) was reached)

\( = \forall w . \text{true} \)

We will now show the proof of the last rule of 1. The expression is given by

\( (c \mid s \models \text{IGC}) \rightarrow (c \land (\neg \text{pred}_1) \land \ldots \land (\neg \text{pred}_n) \mid s \models \text{CHAOS}) \)

Proof:

At first, we will consider:

\( \neg \text{PRED}_{\text{S}} = (\neg \text{pred}_1) \land \ldots \land (\neg \text{pred}_n) \)
(c \ s \models IGC) \rightarrow (c \land \neg \text{NEGPREDS} \land s \models CHAOS)

(we firstly apply a tactic to expand the first sub-goal into a First Order Logic expression (1), that becomes the second sub-goal)

\[ \forall w . (c \land c \land \neg \text{NEGPREDS}) \Rightarrow (\text{Lift} (s) ; IGC \sqsubseteq \text{Lift} (s) ; \text{CHAOS}) \]

(we make here an assumption (assms) that $c$ is true in order to reach the third sub-goal. If $c$ was false, the antecedent of the implication would also be false, and an implication with a false antecedent is always true)

\[ \forall w . (\neg \text{NEGPREDS}) \Rightarrow (\text{Lift} (s) ; IGC \sqsubseteq \text{Lift} (s) ; \text{CHAOS}) \]

(if all predicates are false, then IGC equals CHAOS)

\[ \forall w . (\neg \text{NEGPREDS}) \Rightarrow (\text{Lift} (s) ; \text{CHAOS} \sqsubseteq \text{Lift} (s) ; \text{CHAOS}) \]

(we then reach a refinement between similar expressions. Be $E = \text{Lift} (s) ; \text{CHAOS}$)

\[ \forall w . (\neg \text{NEGPREDS}) \Rightarrow E \sqsubseteq E \]

(from now on, all tactics will apply only predicate calculus)

\[ \forall w . (\neg \text{NEGPREDS}) \Rightarrow [E \Rightarrow E] \]

\[ \forall w . (\neg \text{NEGPREDS}) \Rightarrow [\neg E \lor E] \]

\[ \forall w . (\neg \text{NEGPREDS}) \Rightarrow [true] \]

\[ \forall w . (\neg \text{NEGPREDS}) \Rightarrow true \]

\[ = true \]

4. Conclusions and future work

On this paper, we provided a full and sound Operational Semantics for Circus, having Woodcock’s [5] and Freitas’ [9] Operational Semantics as a basis. We also used ideas from Oliveira [14] in order to build the syntactic transformation rules for Circus processes. We proved soundness of the laws of the Operational Semantics with respect to the theory of Circus on the Unifying Theories of Programming (UTP), which is the Denotational Semantics of Circus. Our long-term goal is to allow the best possible theory for mechanising it on an automatic theorem prover.

As a future work, we can point out the mechanisation using Isabelle-UTP [1], [7] of the Structural Operational Semantics we developed on this paper. The automatization of the Semantics on the theorem prover will strengthen the reliability of the proofs for the laws we created on this paper. What is more, the Structural Operational Semantics we created for this paper can serve as a basis for all formalisms created from Circus. OhCircus [3] (a version of Circus that is Object Oriented) and SCJ-Circus [13] (Circus for Safety Critical Java) are examples of formalisms whose Operational Semantics can be based on Circus.

References