Defining the Circus operational semantics in the K-framework

Alex Alberto, Marie-Claude Gaudel

▶ **To cite this version:**

Alex Alberto, Marie-Claude Gaudel. Defining the Circus operational semantics in the K-framework. [Research Report] LRI - CNRS, University Paris-Sud; ICMC, University of Sao Paulo. 2017, pp.59. hal-01438386

HAL Id: hal-01438386
https://hal.science/hal-01438386

Submitted on 17 Jan 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Defining the *Circus* operational semantics in the K-framework

Alex Alberto[1] and Marie-Claude Gaudel[2]

[1]Universidade de São Paulo, ICMC, São Carlos, Brazil
[2]LRI, Univ. Paris-Sud and CNRS, Université Paris-Saclay, 91405 Orsay, France

January 17, 2017

Abstract

This report documents how we have implemented a trace generator for the *Circus* specification language using K, a rewrite-based executable semantic framework in which programming languages, type systems and formal analysis tools can be defined using configurations, computations and rules. This implementation is based on the operational semantics of *Circus*, that we have revisited to make it exploitable with K. The motivation of this work is the development of a test generation environment for *Circus*. Moreover, it may provide some inspiration to the developers of tools for specification languages based on process algebras.

Contents

1 Introduction **3**

2 Background **4**
2.1 The K-Framework 4
2.2 *Circus* 5

3 Syntax **8**

4 Configuration **12**
4.1 Structures of loaded processes: the *procstrs* cell 14
4.2 Local configurations for symbolic evaluation of processes: the *procs* and *proc* cells 15
4.3 Possible initials after a configuration: the *inits* cell bag 16

5 Symbolic evaluation of process synchronizations **17**

6 Structural and embedding rules **20**
6.1 Specification processing and loading 20
6.2 Configuration maintenance 24

7 Operational semantic rules **25**
7.1 Internal progress 26
7.2 State and $\mathbb{Z}$ schema 29
7.3 Observable progress 31

8 Examples **45**

9 Conclusions and future work **50**

A Operational semantics: table of selected transition rules **53**

B Specification-oriented transition system: labels and rules **57**

1 Introduction

Circus is a state-rich process algebra combining Z [WD96], CSP [Ros98], and a refinement calculus [Mor94]. Its denotational and operational semantics are based on the Unifying Theories of Programming (UTP) [HJ98].

The K-Framework is a rewrite-based executable semantic framework for developing tools based on the operational semantics of programming languages [Ros07].

We report how we have defined a configuration structure and a set of rewrite rules within the K-Framework, which, given a *Circus* specification, produces an executable symbolic transition system that mimics the one defined by the formal operational semantics of *Circus*.

The result is a symbolic evaluator that takes the text of a *Circus* specification as input and produces, as output of interest, a constraint and two sets of symbolic traces: one corresponding to the *constrained symbolic traces* of the *Circus* operational semantics [CG11a]; the other to the *specification traces*, closer to the specification text, that were introduced in [CG14] to facilitate the analysis of test coverage criteria.

We have translated each *Circus* operational semantic rule into some equivalent K-Framework rewrite rule or set of rules. The efforts were focused on keeping a close similarity between the rules of the formal semantics and the input format of the K-framework.

Most of the rules were translated in a straightforward way, permitting a crosscheck between the operational semantics and the designed rewriting version. Nevertheless, some behaviors, especially those related to the synchronization of nested concurrent processes, required the adoption of more complex strategies.

Non-determinism is allowed in *Circus* specifications and it was handled exploiting the backtracking mechanism of the K-framework, guaranteeing the coverage of all possibilities.

We have organised this document as follows:

- The statement of a simplified, machine readable, syntax for the *Circus* language is given in Section 3;
- The definition of a configuration to keep the necessary information for handling the state and other properties for *Circus* processes is given in Section 4;
- The adopted strategy to handle communications and synchronizations between concurrent processes is described in Section 5;
- The structural rules for dealing with structural issues, such as loading and optimizing the configuration, are presented in Section 6;
- The *Circus* operational semantics rules designed as K-Framework rewrite rules are presented in Section 7;

- Some examples of specification texts, used for validating the behavior of the transition system, and the corresponding traces are presented in Section 8.

Moreover, the next section contains a brief introduction to some background concepts.

2 Background

2.1 The K-Framework

The K-framework is presented by its authors as a rewrite-based definitional framework in which programming languages, calculi, as well as type systems or formal analysis tools can be defined [RŞ10]. It provides many desirable aspects such as modularity, non-determinism and concurrency handling. The framework is presented formally [Ros07] and packed in a reference implementation, offering tools to generate executable interpreters that allows state-exploration and reasoning about programs [CLRR16].

Semantics for programming languages, calculi and other analysis tools are defined in the framework by using a set of rewrite rules and labelled, potentially nested, cell structures, refered to as the *configuration*. The content of the configuration cells keeps relevant information such as the state and general environment for the system/program.

There are two types of rewrite rules: *computational*, which count as computational steps, and *structural* rules, which are used to rearrange the terms so that the computational rules can apply. A *computation* is an element or a sequential list of elements that carries "computational meaning", that is, a sequence of computational tasks. For instance, the assignment of the value of an expression to a variable is a computation, composed by a sequence of computations which are required to, first, compute the value of the expression and, then, perform the adequate changes to the affected environment configuration cells.

In the K-Framework, computations are syntactical elements of the sort K and they are stored and handled inside the configuration cell labelled as k. This cell is referred to as the *k cell* or the *computation cell*. Computations have a list structure, capturing the intuition of computation sequentialization, with the symbol $\sim>$ as list the separator, to be read as "followed by" and the unit symbol "." (the empty computation).

In particular, computations extend the original language or calculus syntax. When necessary to avoid ambiguities, throughout this report we use the name *k-computation* when referring to a computation inside the k-cell. The k-computations can be handled like any other terms in the rewriting environment, that is, they can be matched, moved from one place to another in the original term, modified, or even deleted.

In such a framework, it is important to distinguish between computations under treatment and computations already completed, for instance, between

expressions and their values. To allow this distinction, the framework defines the *KResult* internal syntactic subcategory. Coming back to the variable assignment example, the treatment of the expression will be judged finished when the achieved result is classified under *KResult* category as, for example, an integer value.

The rewrite rules in the K-Framework are unconditional, although they may have ordinary side conditions, and they are context-insensitive, so the rules apply concurrently as soon as they match. They generalize conventional rewrite rules by making explicit which parts of the term is read, write, or ignored.

The framework has been developed with the mechanisation of structural operational semantics in mind. The correspondence can be summarised, in a nutshell, as:

- Computation terms and other auxiliary information such as environment and state, are contained in nested *configuration cells*. Computations are kept in special *k-cells*, and other information are organised and managed in cells and sub-cells, depending on the considered language.
- Rewrite rules correspond to transitions of the operational semantics. They describe the evolution of the top configuration cell. Computations progresses are reflected by evolutions of k-sub-cells. Changes of environment and state are reflected by changes of other cells.
- Structural rules describe computations rearrangements, so that rewrite rules can match and apply.

Such definitions can be processed by the rewrite engine that is the kernel of the K framework. As say the authors, "interpreters for free" can be obtained from formal language definitions, as well as various tools based on operational semantics. In this work, our immediate goal was symbolic traces generation for *Circus*, having in mind a test generation environment as described in [ACGS16]

2.2 *Circus*

This section is a slightly modified version of Section 2 of our paper [ACGS16]. As exemplified in Figure 1, *Circus* makes it possible to model systems and their components via a set of interacting processes. Each process has:

a state and some operations for observing and changing it in a Z style. In our example, the state is a pair *AState* of variables named *sec* and *min* with integer values between 0 and 59 (as defined by *RANGE*), and the data operations on this state are specified by the three schemas *AInit*, *IncSec*, *IncMin*.

actions that define communicating behaviours in a CSP style. In our example, the behaviour of the *Chrono* process is specified by the main action after the symbol •. It is a sequential composition of the schema *AInit* followed by the repeated execution of the *Run* action. *Run* starts with an external

channel $tick, time$
channel out

process $Chrono \mathrel{\widehat=}$ **begin**
  state $AState == [\, sec, min : RANGE \,]$
    $AInit == [\, AState' \mid sec' = min' \land min' = 0 \,]$
    $IncSec == [\, \Delta AState \mid sec' = (sec + 1) mod\ 60 \land min' = min \,]$
    $IncMin == [\, \Delta AState \mid min' = (min + 1) mod\ 60 \land sec' = sec \,]$
    $Run \mathrel{\widehat=} tick \rightarrow IncSec;\ ((sec = 0)\ \&\ IncMin)\ \Box\ ((sec \neq 0)\ \&\ \mathbf{Skip})$
        $\Box$
        $time \rightarrow out!(min, sec) \rightarrow \mathbf{Skip}$
    $\bullet\ (AInit;\ (\mu\ X \bullet (Run;\ X)))$
end

Figure 1: A *Circus* specification of a chronometer

choice between two events *tick* and *time*; *tick* is followed by the increment of the chronometer, and *time* by the display of the values of *min* and *sec*.

Circus comes with a denotational and an operational semantics, based on Hoare and He's Unifying Theories of Programming (UTP) [HJ98], and a notion of refinement. We can use *Circus* to write abstract as well as more concrete specifications, or even programs. A full account of *Circus* and its denotational semantics is given in [OCW09].

The operational semantics for *Circus* is briefly introduced below, and a significant part is reproduced in Appendix A. It is defined as a symbolic labelled transition system between configurations. These are triples ($\texttt{c} \mid \texttt{s} \models \texttt{A}$), with a constraint `c`, a state `s`, and a continuation `A`, which is a *Circus* action. Transitions associate two configurations and a label. The labels are either empty, represented by ϵ, or symbolic communications of the form `c?w` or `c!w`, where `c` is a channel name and `w` is a symbolic variable that represents an input (?) or an output (!) value.

The first component `c` of a configuration ($\texttt{c} \mid \texttt{s} \models \texttt{A}$) is a constraint over symbolic variables that are used to define labels and the state. The constraints are texts that denote *Circus* predicates over these symbolic variables. We use typewriter font for pieces of text. The second component `s` is a UTP predicate, which defines a total assignment $\texttt{x} := \texttt{w}$ of symbolic variables `w` to all variables `x` in scope, including the state components. State assignments, however, can also include declarations and undeclarations of variables using the constructs $\texttt{var x} := \texttt{e}$ and `end x`. The state assignments define a value for all variables in scope. These values are represented by symbolic variables similarly to what is

$$\frac{c \land T \neq \varnothing \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d?x : T \rightarrow A}) \xrightarrow{\mathtt{d?w_0}} (\mathtt{c} \land \mathtt{w_0} \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\,x := w_0} \models \mathtt{let\ x \bullet A})}$$

$$\frac{c \land (s;\ g)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{g} \,\&\, \mathtt{A}) \xrightarrow{\epsilon} (\mathtt{c} \land (\mathtt{s};\ \mathtt{g}) \mid \mathtt{s} \models \mathtt{A})}$$

Figure 2: Examples of two transition rules: for inputs, and for guards

classically done in symbolic execution of programs [Kin76].

Two examples of rules are given in Figure 2. The first rule defines the
150 transitions arising from an input prefixing $\mathtt{d?x : T \rightarrow A}$; it is rule (A.5) of A. The label of the transition is $\mathtt{d?w_0}$, where w_0 is a symbolic variable. The constraint that w_0 is of the right type ($\mathtt{w_0} \in \mathtt{T}$) is added to the constraint of the new configuration. The state of the new configuration is enriched, via the UTP sequence operator "; ", by a new component x, which is assigned value w_0. The
155 continuation of the new configuration is the action A in an environment enriched by x as defined by $\mathtt{let\ x \bullet A}$.

The second rule of Figure 2 defines the transitions for a guarded action g & A. The label of the transition is empty, since the evaluation of g is not an observable event; g is added to the constraint of the new configuration taking
160 into account the assignments in the current state s. The continuation is A.

Traces of a process are defined in the usual way, that is, as sequences of observable events. Due to the symbolic nature of configurations and labels, however, we can obtain from the operational semantics *constrained symbolic traces*, or *cstraces*, for short. These are pairs formed by a sequence of labels,
165 that is, a symbolic trace, and a constraint over the symbolic variables used in the labels. Roughly speaking, the constrained symbolic trace can be obtained by evaluating the operational semantics, collecting the labels together, and accumulating the constraints over the symbolic variables used in the labels. We give below some examples of cstraces of *Chrono*.

$$\begin{aligned}
cst1 :& (\langle tick \rangle, true) \\
cst2 :& (\langle time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 0 \land \alpha_1 = 0) \\
cst3 :& (\langle tick, time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 0 \land \alpha_1 = 1) \\
cst4 :& (\langle tick \rangle^{60}, true) \\
cst5 :& (\langle tick \rangle^{60} \frown \langle time, out!\alpha_0!\alpha_1 \rangle, \alpha_0 = 1 \land \alpha_1 = 0)
\end{aligned}$$

170 A trace is an instantiation of a cstrace, where the symbolic variables used in the labels are replaced by values satisfying the constraint. For instance, the two traces $\langle time, out!0!0 \rangle$ and $\langle tick, time, out!0!1 \rangle$ are instances of $cst2$ and $cst3$.

In [CG10, CG14, CG13a] *specification traces* were defined. Their distinctive feature is their labels. They record not only events, like in the operational semantics, but also guards and state changes. Moreover, they are expressed in terms of the variables of the specification text, rather than symbolic variables. Such traces are necessary to take into account coverage criteria of the specification text. The syntax of the labels and the main transition rules are given in Appendix B.

In this work, we have developed, using the K-Framework a trace generator for *Circus* that produces both the cstraces and the specification traces of *Circus* specifications.

Limitations of current implementation

The current version of our symbolic trace generator is limited in dealing with certain aspects of *Circus* specifications:

Type checking is not yet implemented and, as a result, it is necessary to handle each desired type with specific rewrite rules. In the current version, we handle integers and sets, but it is trivial to expand the implementation with rules for more types;

Multiple valued communication is not available, for the moment, we only deal with single valued inputs or outputs;

Schema expressions are not treated in the current version. Besides, the formulas and expressions usable when writing $\mathbb{Z}$ schemas are restricted to the use of native types of the K-framework;

Parallel assignments are not correctly tracked into the specification trace output, but it is done for cstraces;

Specification traces for external choice contains specifications labels of discarded branches. Nevertheless, this undesired behavior is consistent with the operational semantics. It does not affect symbolic traces generation.

In Section 9 we introduce a brief discussion of the limitations that should be alleviated in future works. The next section contains the details of how we designed the simplified machine readable syntax for treating *Circus* specification texts.

3 Syntax

To establish a simple, text-only, machine-readable syntax for *Circus* specifications, we started from the syntax defined by Feliachi et al. [FGW13] for processing the language in the proof assistant Isabelle/HOL [PW02]. We also drawn some inspiration from the machine readable language for CSP, namely CSP_m [SA11].

210 We have added some special syntactical constructs for internal use only: they are not expected in the *Circus* specification input by the user, but they are needed as place marks to control the K rewriting engine while performing complex step sequences. When it is the case, such fact will be pointed out in the syntax and semantics descriptions given in the sequel.

215 The syntax used in our implementation is organised in three modules: the main *Circus* syntax, the syntax for expressions, and a module of useful constructs shared between the other two. We refrain from presenting all modules in this section, focusing only on the elements of the first module.

We present our *Circus* syntax in a BNF style as it is accepted by the K 220 Framework. The root non-terminal element is *CircusSpec* (1), a complete *Circus* specification that is a sequence *CircusPars* (2) of *Circus* paragraphs.

```
syntax CircusSpec ::= CircusPars                                  (1)
syntax CircusPars ::= CircusPar | CircusPar CircusPars            (2)
```

A *Circus* paragraph *CircusPar* (3) can represent a process declaration, a $\mathbb{Z}$ paragraph, a channel declaration, a channel set declaration, a name set declaration or the special internal use keyword ":run" (6) that allows the user to select 225 which declared process initiates the execution.

```
syntax CircusPar ::=                                              (3)
    ProcDecl
  | ZParagraph
  | "chanset" Id "==" SetExp [strict(2)]                          (4)
  | "nameset" Id "==" SetExp [strict(2)]                          (5)
  | CompleteCDecl
  | ":run" Id                                                     (6)
```

The declaration of sets for channels (4) and names (5) associates an identifier with a set expression *SetExp*. This last element is tagged as *strict*, forcing the K-Framework to advance the computation of *SetExpr* before processing the declaration itself.

230 An element of the sort *SetExpr* is defined in the auxiliary syntax module for expressions, which is omitted here. In a brief description, a *SetExpr* allows the definition of sets of names using the *Circus* syntax, i.e., comma-separated and delimited by "{|" and "|}". It also allows expressions over these sets (such as union, intersection, exclusion, etc.) and identifiers to reference a previously 235 declared named set.

```
syntax CompleteCDecl ::= "channel" MultiDecl                      (7)
syntax Type ::= "Int"                                             (8)
syntax SimpleDecl ::= Id | Id ":" Type                            (9)
syntax MultiDecl ::= List{SimpleDecl, ","}                        (10)
```

Channel declarations are defined by *CompleteCDecl* (7). Prefixed by the

keyword "`channel`", such paragraphs can contain one or multiple (10), typed (8) or untyped simple declarations *SimpleDecl* (9). The channel name is an identifier.

240 A process declaration *ProcDecl* (11) states the association between an identifier, that will reference the process throughout the environment, and a process definition *ProcDef* (13) or an action paragraph *Action* (20). For specifications that use the second variant, we define a transformation macro rule (12) that will convert it into a full process definition and place the given action paragraph as 245 the so-called anonymous action of the process (see (13) below).

```
syntax ProcDecl ::= "process" Id "^=" ProcDef                    (11)
                   |"process" Id "^=" Action
rule  process P:Id ^= A:Action                                   (12)
   => process P    ^= begin @ A end [macro]
```

A process definition *ProcDef* (13) consists in an optional sequence of process paragraphs *PPars* (15) followed by the mandatory anonymous action paragraph *Action* (20), after the "`@`" symbol. The definition is surrounded by the keywords "`begin`" and "`end`".

```
syntax ProcDef ::=                                               (13)
    "begin" PPars  "@" Action "end"
  | "begin" "@" Action "end"
```

250 A process paragraph *PPar* (14) contains either a $\mathbb{Z}$ paragraph *ZParagraph* (16) or a named action declaration.

```
syntax PPar ::= ZParagraph | Id "^=" Action                      (14)
syntax PPars ::= PPar | PPar PPars                               (15)
```

The *ZParagraph* (16) is either a schema or a state declaration. The state declaration is prefixed by the keyword "`state`" followed by an association between the state name identifier and a single schema definition. The standard 255 non-state schema declaration also allows the association to a *SchemaExp* (17).

```
syntax ZParagraph ::= Id "==" SchemaExp                          (16)
                    | "state" Id "==" Schema
```

Although mentioned in this syntax, the current implementation offers no support to schema expressions. That explains the reason for the simple definition of *SchemaExp* (17). The extension to schema expressions is a subject for future work.

260 A *Schema* (18) has two parts: some variable declarations and a condition, which is a boolean expression. The second part is optional and, when omitted, a structural macro rule (19) introduces the "`true`" keyword at its place.

```
syntax SchemaExp ::= Schema                                  (17)
syntax Schema ::= "[" MultiDecl "]"                          (18)
                | "[" MultiDecl "|" BoolExp "]"
rule  [ M:MultiDecl ] => [ M | true ] [macro]                (19)
```

The most complex non-terminal is *Action* (20), which combines CSP syntax with *Circus* elements. It covers CSP constructs like the standard process names (22), prefixed communication (23), guarded actions (25), channel hiding (28) and several composition operators that are briefly presented by in-line comments. On the *Circus* side there are constructs like $\mathbb{Z}$ schema (21), local variable declarations (24), recursive actions (26) and other *Circus* specific commands (29) that are explained in the *Command* (31) element. The presence of comparison operators and the keyword "`left`", like in (27), gives the precedence and the associativity of nested syntactical elements.

```
syntax Action ::= Id                                         (20)
  | Schema                                                   (21)
  | "Skip" | "Stop" | "Chaos"                                (22)
  | "(" Action ")"
  > Comm "->" Action                                         (23)
  | "var" Id "@" Action                                      (24)
  >  BoolExp "&" Action                                      (25)
  >  BoolExp "g" Action [strict(1)]
  | "u" Id "@" Action                                        (26)
  > left:                                                    (27)
    Action "/H" SetExp                                       (28)
  | Action ";" Action   //sequential composition
  | Action "[]" Action  //external choice
  | Action "|-|" Action //internal choice
  | Action "[|" SetExp "|" SetExp "|" SetExp "|]" Action
  | Action "[|" SetExp "|]" Action //parallel composition
  | Action "|||" Action //interleave
  > Command                                                  (29)
```

The syntax of the communication part of a prefixed action is given by *Comm* (30). It can correspond to synchronization events, i.e. channel name identifiers, or single valued inputs and outputs. The use of the strictness attribute in the output is required to force the calculation of the expression value before processing the output construct as a whole. The support for multi-valued inputs and outputs is let to future improvement.

```
syntax Comm ::= Id                                           (30)
  | Id "?" Id
  | Id "!" Exp [strict(2)]
```

The *Command* non-terminal corresponds to *Circus* commands, such as variable assignment and variable scope finalization, and other special internal use
280 constructs.

```
syntax Command ::=                                                    (31)
  Id ":=" Exp [strict(2)]
  | "end" Id
  | InternalCommand
```

Such internal use constructs, showed in (32), are not meant for user input, but used for controlling intermediate steps of the rewriting engine. A complete description of these extra steps is given with the rules for the operational semantics in Section 7.

```
syntax InternalCommand ::=                                            (32)
  | "choice" Set
  | "par" List "," Set "," Set "," Set
  | "hide" Int Set
  | "proc" Int
  | "state" BoolExp [strict]
  | "schema" BoolExp [strict]
```

285 At last, we need to put all the syntax elements that should be handled as actual values under the framework built-in category *KResult* (33). A k-computation is considered completed as soon as the achieved result is of the sort of one of the elements listed under such category.

```
syntax KResult ::= Int | Set | Bool                                   (33)
```

After parsing a well formed *Circus* specification, the K-Framework will begin
290 the application of semantic rules. For instance, before evaluating the actual behavior of any processes, structural rules are applied to handle the loading of the input specification into the configuration cells. We present in the next section some relevant notions of the K-Framework configuration, along with the configuration structure that we have designed for holding the *Circus* specifications
295 elements.

4 Configuration

A configuration in the K-Framework is a nested bag of configuration items (or of configuration item terms) referred as configurations cells. During the processing of a *Circus* specification, the configuration provides an abstraction of the state
300 and of the infrastructure needed to process the continuation of the specification, i.e. to compute its subsequent behavior.

Depending on the considered semantics, configurations may contain cells that keep the current version of: the K-computation, the environment, the store, some analysis results, and various useful bookkeeping information [Ros07].
305 Given its wide range of use, the K framework does not limit nor impose any restrictions over the design of such a structure, leaving it entirely as a subject of the needs of the semantics being treated and of the kind of tool being developed: interpreter, type checker, symbolic evaluator, compiler, trace generator, etc.

There are two kinds of rules describing transitions between configurations:
310 structural rules, and computational rules. Structural rules process and organise the text of the specification in a way that fits the application of computational rules. Structural rules have no counterpart in the operational semantics; they can be seen as transformations, preserving the semantics of the text being treated, that aim at preparing and controlling the application of the computa-
315 tional rules. Computational rules correspond to transitions in the operational semantics and describe those changes of the configuration corresponding to computations, i.e. reduction steps.

More precisely, structural rules yield some initial content of the $< k >$-cells (and of some auxiliary cells) called in the sequel the *loaded specification* that
320 drives the computational rules to be applied.

This section starts with an overview of the *Circus* configuration. Then we present in sub-section 4.1 the cells that correspond to the loaded specification. These cells memorise the way the *Circus* specification is structured into various processes and actions. Finally, in sub-section 4.2 we explain the local configu-
325 ration that corresponds to the each of these processes or actions.

Figure 3 shows an overview of the configuration we have designed for representing and processing *Circus* specifications. It is presented using the same syntax as accepted by the framework: in a XML style where each tag represents a configuration cell. Such cells can be nested and replicated when explicitly
330 annotated with the attribute "`multiplicity`".

Due to the size of the whole structure, the content of some cells is omitted in this overview and presented later.

The *top* cell encapsulates the whole configuration and contains cells that handle global information, such as: the next fresh integer reference in *nextid*;
335 the map of channel names into channel types in *ch*; the map of set names into the actual sets in *nset*; the structures of loaded processes in *procstrs*; the set of discarded processes references in *discarded*; a temporary set used to convert *Circus* sets into native K-Framework sets in *tempset*; the particular configuration of every process in *procs*; non-resolved initials ready to be synchronised with
340 some other process in *inits*; non-consumed notifications of resolved initials in *oks*; and finally, the results of the treatment of the *Circus* specification, i.e. specification traces and symbolic traces in the *spectr* and *tr* cells.

In the sequel of this section, we give a first glimpse of the roles of these cells. This will be made more precise when presenting the transition rules between
345 configurations in Sections 6 and 7.

```
configuration
  <top>
    <nextid> 1 </nextid>
    <ch> .Map </ch>
    <nset> .Map </nset>
    <procstrs> ... </procstrs>
    <discarded> .Set </discarded>
    <tempset> .Set </tempset>
    <procs> ... </procs>
    <inits> ... </inits>
    <oks> ... </oks>
    <spectr> .K </spectr>
    <tr> .K </tr>
  </top>
```

Figure 3: Overview of the configuration

4.1 Structures of loaded processes: the *procstrs* cell

The content of the cell *procstrs* is a sub structure that represents the processes loaded from the specification text. Presented in Figure 4, the subcell *procstr*, which actually is a bag of subsubcells, can be replicated as many times as 350 processes are found in the specification text. It contains a name that identifies the process, stored in *pmodel* cell, a map of named action identifiers to action texts in *pact*, a map of schema names to schema texts in *psch*, and the complete process text, which is stored in *pdef*. Whenever a K-computation involves an identifier referring to any such elements, e.g., named action call, schema 355 or process inclusion, this structure is used to retrieve the required piece of specification text.

```
<top>...
  <procstrs>
    <procstr multiplicity="*">
      <pmodel> .K </pmodel>
      <pact> .Map </pact>
      <psch> .Map </psch>
      <pdef> .K </pdef>
      <pstate> .Map </pstate>
      <pinv> .K </pinv>
    </procstr>
  </procstrs>
...</top>
```

Figure 4: The process textual structures cell *procstrs*

4.2 Local configurations for symbolic evaluation of processes: the *procs* and *proc* cells

For processes, sub-processes or actions under symbolic evaluation, we manipulate individual *proc* structures that are subcells of the cell *procs*. This construction is shown in Figure 5. A *proc* cell can be referred as an individual environment for each element that performs K-computations.

This rather complex cell keeps all the elements needed to perform the K-computation that corresponds to the symbolic evaluation of a process or action and all the relationship to its context, i.e. other processes or actions.

The *proc* cells are spawned on demand whenever a new line of K-computation with its own environment is required, e.g., concurrent components in a parallel composition, alternatives of an external choice, inclusion of an independent state-isolated process, etc. These situations will be explained when the corresponding operational semantic rules are presented, in Section 7. The references between these cells reflect the hierarchical tree structures induced by the nested organisation of the processes and actions in the *Circus* specification.

```
<top>...
  <procs>
    <proc color="green" multiplicity="*">
      <model> .K </model>
      <k>$PGM:K</k>
      <id>0</id>
      <alt>0</alt>
      <parent>0</parent>
      <env> .Map </env>
      <postenv> .Map </postenv>
      <stack> .List </stack>
      <sync>false</sync>
      <const>true</const>
      <temp> .K </temp>
    </proc>
  </procs>
...</top>
```

Figure 5: The process cell *procs*

A *proc* cell contains: in the *model* subcell, an identifier that references the name of the process which originated it; in k, the K-computation cell itself; in *id*, an integer reference that is used to globally identify the current environment; in *alt*, a reference to an external choice alternative execution line, if any; in *parent*, a reference to the environment which spawned it; in *env*, a map that associates variable names to symbolic values; in *postenv*, the same sort of map as *env*, but specifically for manipulation of schema post conditions; in *inv*, the state invariant text as written in the process specification; in *stack*, a stack of

env environments, used for managing nested variable scopes; in *sync*, a boolean flag that is used to put the process evaluation on hold until an adequate synchronization of an initial (see subsection 4.3 and section 5); in *const*, the constraint on the symbolic values in use by the *env* variables; and an auxiliary cell *temp*, buffering the specification text before outputting it in the generated traces.

4.3 Possible initials after a configuration: the *inits* cell bag

The initials cell bag *inits*, shown in Figure 6 is a buffer of observable events or communications that are ready to be performed by an action. Its content is a subset of the set of initials as it is defined in the *Circus* testing theory [CG11b]. The symbolic evaluation of an action can be ready to perform several observable events: it may be the case for actions involving multiple nested parallel compositions or there may be pending synchronizations, or explicitly hidden events or communications. These complications led us to design the configuration structure described below: it makes it possible to buffer and manipulate all events or communications that are potentially observable after the configuration. One

```
<top>...
  <inits>
    <init multiplicity="*">
      <iresp> 0 </iresp>
      <iev> .K </iev>
      <from> .Set </from>
      <ialt> .Set </ialt>
    </init>
  </inits>
...</top>
```

Figure 6: The process initial structures cell *inits*

init cell holds a reference in *iresp* to the *proc* cell that is currently responsible for its management. In *iev* is the description of the observable synchronization event or communication that is managed. Moreover, the set *from* holds references to all *proc* cells that are standing for the outcome of such initial, i.e. the cells of the action that originated it and of all other actions that might synchronize when this initial is resolved. Besides, *ialt* is a set of references to alternative *proc* cells, i.e. the ones corresponding to actions that may be discarded as results of external choices when the initial is resolved. We use the term *resolved*, rather than performed, since there are no guarantees that an initial posted during the symbolic ecaluation of an action will be actually performed: it might be held infinitively in a synchronized parallel composition, or hidden by a channel hiding operation.

The resolution of an initial creates an *ok* notification, described in Figure 7. Each *ok* notification contains the event or communication actually performed

410 and the set *to* of the references of all *proc* cells that were standing for the outcome of the resolved *init*.

```
<top>...
  <oks>
    <ok multiplicity="*">
      <oev> .K </oev>
      <to> .Set </to>
    </ok>
  </oks>
...</top>
```

Figure 7: The process initial notification structures cell *oks*

We give more details on the way synchronizations are managed during the symbolic evaluation in the next section.

5 Symbolic evaluation of process synchronizations

415

As mentioned above, the behavior modeling part of *Circus* is founded over similar bases as the CSP process algebra [Hoa85]. This long standing and well-known modeling algebra for processes provides a powerful set of constructs to precisely specify parallel behaviors and interactions.

420 Although allowing the interpretation of parallel computations (threads), as exemplified in [RS14] and [ŞAL+13], the K-Framework does not offer predefined resources to facilitate the control of synchronization between concurrent processes. It is left to the designer interested in entering an operational semantics with concurrent aspects to define a strategy and design the required 425 configuration structures that will guide a semantically correct parallel behavior.

In the most complete available formal presentation of the K framework [Ros07], two examples are briefly sketched to introduce the possibility of modeling process algebra semantics: for CCS and π-Calculus. But to the best of our knowledge, there is no published (or posted) operational semantics of CSP 430 in K.

In our context, it was essential to develop in K a correct semantic representation of the *Circus* concurrent features. That motivated the design of a synchronization strategy that could be embedded in the K-Framework configuration cells and manipulated respecting its single step rewrite logic rules.

435 Our strategy consists in organizing the references between the *proc* cells in a tree structure that splits hierarchically the lines of execution for critical constructions, namely: parallel composition, external choice, channel hiding and independent process inclusion. Given this tree hierarchy, it is possible to manipulate the observable events or communications that may be produced at

any depth in the tree and to manage potential synchronisations or deadlocks, and hiding.

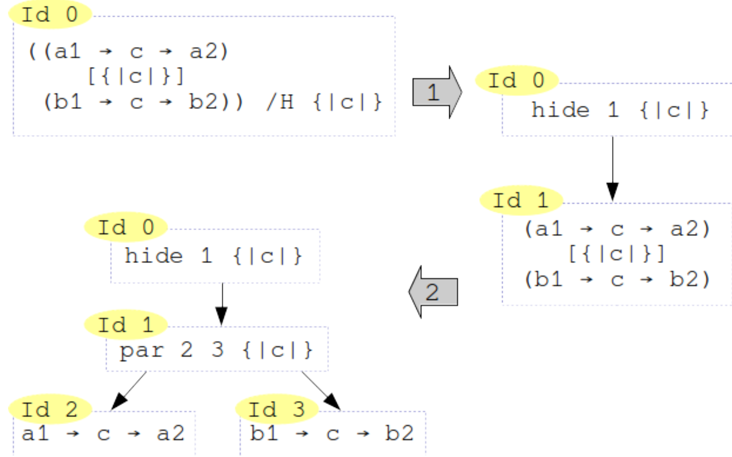

Figure 8: Sample graphical representation of a synchronization tree

We show in Figure 8 a graphical representation of the rewriting steps that build the tree structure for a simple example, which contains a parallel composition inside a channel hiding action.

Starting from the text of the main action specification, in the top level *proc* cell, (globally identified with the integer reference Id 0), the first step gives to the action subjected to the channel hiding operation a lower hierarchical level. Thus a new *proc* cell, identified with a fresh Id 1, is spawned and noted as a subtree of *proc* Id 0. For convenience, we will refer to an uniquely identified *proc* cell as *proc* # n, where n is its integer reference. The text of the action of the top-level *proc* # 0 is rewritten into an internal syntactical construction that indicates it as a subordinated hidden action, giving the reference to its *proc* cell and the set of hidden channels.

The second rewriting step splits both sides of the parallel composition present in the recently created *proc* # 1 into two fresh *proc* cells, *proc* # 2 and *proc* # 3, respectively for left and right side that are noted as subtrees of *proc* # 1. The text of the parallel action inside *proc* # 1 is rewritten into an internal syntactical construction that keeps as parameters the references to the new subtree *proc* cells and the set of channels that must be used for synchronizing the corresponding parallel actions.

When a K-computation being processed in a *proc* cell involves an observable event or communication, this item is never immediately added to the traces cells *tr* and *spectr*: it is posted as a potential initial in a new *init* cell as seen in subsection 4.3. In our example, it is the case for *proc* # 2 and *proc* # 3. Once an initial is posted, the current K-computation remains on hold by lifting the boolean flag inside its *sync* configuration cell. It will remain standing by until

a proper *resolution* of the initial is reached.

We say that an initial is *resolved* when it matches the criteria of a resolution rule: some semantic rule that will either suppress the initial (75) or write it into the traces (87). The resolution causes an *ok* notification inside the *oks* configuration cell. Each notification is accompanied by a *to* cell, that contains a set of references to all *proc* environments waiting for the initial resolution.

A posted initial will be manipulated and delegated to the upper tree levels by *forwarding rules*. Following the *Circus* operational semantics, there are different sorts of forwarding rules for each splitting construct.

The reason for having this life-cycle for initials, namely: posting, forwarding, resolving and/or discarding, is that the presence of an observable communication/event in a trace depends on the context of the action that triggers it. In two cases, such observable communications/events may never appear in the traces of the main process, i.e., the process under analysis.

First, it may appear in OPS rules where, above the rule, there is a non-ϵ labeled transition but, under the rule, the transition is labelled with ϵ (silent). It is the case for the OPS rules A.24, A.25 and A.27, which defines channel hiding and parallel synchronization.

The other case is when an action present in the original configuration of a transition disappears in the target configuration, as in the case of the OPS rules A.18 and A.19, which defines the behavior for external choice. As a result, observable events/communications triggered by the discarded action will never be written in the traces.

In the following, we introduce the management of initials for the *Circus* constructions mentioned in the two cases above, namely: channel hiding, parallel composition and external choice.

Channel hiding is modeled with rules that may selectively forward initials concerning non hidden channels (74) or instantly resolve an initial from the hidden ones (75), suppressing it and spawning the respective *ok* notification.

The parallel composition forwarding rules will pass by any initial that is not related to the channels listed in the synchronization set (80). Otherwise, the initial will be hold until another initial, matching the synchronization, is produced by the symbolic evaluation of the action at the other side of the composition (81).

An external choice operation will indiscriminately forward the initials from both sides, but an alternative reference is marked in the *alt* cell of each, pointing to the *proc* cell in the competing side: the first resolved initial will determine which side is chosen and, consequently, the discarding of all marked alternatives.

More details about these rules and the conditions for forwarding initials to upper levels are given in Section 7.

If an initial, after all the manipulations that it may receive in the lower levels, is finally delegated to the $proc\,\#\,0$ environment, then it is eligible for producing some observable event or communication. However, it is not assured that this will actually occur, since an initial might reach the top-level after the discarding of all those *proc* cells that were matching its outcome. In that case, the initial

will be just discarded. Otherwise, the observable event or communication is appended to the traces and the respective *ok* notification is posted.

A *proc* on hold by the *sync* flag is ready to continue its K-computation once the appropriate *ok* notification exists. The continuation is triggered by a rule (88), which tries to match the references of standing-by processes with references in the addressing subcell of *ok* notifications: once matched, a reference is removed from the *ok* notification, the process *sync* flag is released and the standing by K-computation takes place. A synchronized *ok* notification might be addressed to more than one *proc* at the same time and, for that reason, they are not discarded in this step: it would prevent the continuation of the other standing by *proc*. After the removal of all standing-by *proc* references, there is a specific structural rule dedicated to the cleaning of *ok* notifications with empty addressing subcell.

The two last sections have presented the defined structure for the configuration. In the two next sections, we proceed to the description of the structural rules that link the text of *Circus* specifications to this structure, and then of the rewrite rules that describe the transitions between configurations according to the *Circus* operational semantics [CG13b].

6 Structural and embedding rules

To handle the text of a *Circus* specification, it is necessary to write rules to process the input and organize it in a way that fits the designed configuration structure. We refer to them as structural rules, and they have neither semantic effect nor counterparts in the operational semantics.

6.1 Specification processing and loading

As seen in Section 3, the syntax for the source code of a *Circus* specification is a sequence of *Circus* paragraphs, i.e., non-terminals elements of the sort *CircusPar*. We start with a simple rule to enforce the sequential processing of all *CircusPar* that are present in the source code, displayed in (34).

```
rule P1:CircusPar P2:CircusPars => P1 ~>P2
```
(34)

The "=>" symbol is the separator between the term matching and how it should be rewritten. The "∼>" symbol is the sequential list separator for K-computations. After explicitly stating the processing order for the top most structure, we give the rules to deal with the sorts of *CircusPars* expected to be found in a specification. The following rules deal with the declaration of channels.

Remark (K-Framework rewrite rules)**.** *Rewriting into the K empty symbol, i.e. the "." dot, means that the matched content will be erased. The triple dots "...", that may be present either in the beginning or the end of a mentioned cell, means that there may be more content that are not considered in the matching and will*

```
rule <k> channel C:Id : Type:Id => . ...</k>                    (35)
        <ch> ... .Map => C |-> Type ...</ch>
rule <k> channel C:Id => . ...</k>                              (36)
        <ch> ... .Map => C |-> "sync" ...</ch>
```

550 *remain unchanged after the rewrite step. The "$|->$" symbol is the constructor of map pairs.*

According to rule (35), a paragraph with a channel declaration will be removed from the top of the computation cell after the addition of the name of this channel and its type (for typed channels) to the global configuration cell
555 $< ch >$ as a map pair, with the given channel name as key and the channel type as value. When there is no type declared, the channel name is mapped to *sync*, as seen in rule (36).

To deal with multiple channel declarations in a single line, we profit from the technique referred as syntax *desugaring* [RS14], that rewrites a complex
560 sequence of declarations into single-lined simple declarations. This is described in rule (37).

```
rule channel S1:SimpleDecl, S2:SimpleDecl, M:MultiDecl =>      (37)
    channel S1 channel S2, M
```

For paragraphs defining named sets, i.e., declaration of uniquely identified sets that holds channel and variable names, we use a strategy that embeds a *Circus* set construction into the representation of sets as used natively in the
565 K-framework. To achieve that, when there is a channel set in the top of the computation cell, we move all its elements, one by one, to the $< tempset >$ cell, as seen in rule (38). When the set being computed is emptied, it is replaced by the temporary one, which is a pure K-framework set construction, described in rule (39).

```
rule <k> {| Xi:Id, Xs:CommaIds |} => {| Xs |} ... </k>          (38)
     <tempset> ... . => SetItem(Xi) </tempset>
rule <k> {| .CommaIds |} => TempSet ... </k>                    (39)
     <tempset> TempSet => .Set </tempset>
```

570 Finally, in rules (40) for channels names and (41) for variable names, the named set declaration that now utilizes a K-framework set construction is loaded into the global cell $< nset >$, holding a map from declared set names to the actual K sets.

Rule (42), reverses the process: when an identifier that refers to a set name
575 is found in the top of the computation cell, that identifier is replaced by the referred set, loaded from the map in $< nset >$. The embedding of native K-Framework set type allows the convenient use of internal operations, such as union, intersection, difference and comparisons. Moreover, the framework provides an interface to SMT solvers, in our case, Z3 [DMB08], which is invokable
580 each time a formula involving native types needs to be solved.

```
rule <k>chanset CSName:Id == CSet:Set => . ... </k>                 (40)
     <nset>... .Map => CSName |-> CSet </nset>
rule <k>nameset NSName:Id == NSet:Set => . ... </k>                 (41)
     <nset>... .Map => NSName |-> NSet </nset>
rule <k>NSetId:Id => CSet ...</k>                                   (42)
     <nset>... NSetId |-> CSet:Set ...</nset>
```

Remark (Strictness)**.** *Here, we rely on the heating and cooling rules of the K-Framework [RŞ10], i.e., when present in a syntax construction, named sets are marked as strict arguments and pushed to the top of the computation cell (heating). As sets are declared under KResult category, the processed native set in the top of the K cell is plugged back to its original syntactical place (cooling).*

In order to support more than one process declaration per source text, each paragraph of such syntactical construct is loaded into a structure inside the configuration cell $< procstrs >$. These structures keep references between the declared process name and its textual definition body, named actions, or schemas. This is achieved by rule (43), that removes a process declaration from the top of the computation cell and places it into a new $< procstr >$ cell. The process name identifier P is kept into the $< pmodel >$ cell and its textual process definition PD is stored into the $< pdef >$ cell.

```
rule                                                                (43)
  <k> process P:Id ^= PD:ProcDef  => . ...</k>
    <procstrs>...
      (.Bag => <procstr>...
        <pmodel> P </pmodel>
        <pdef> PD </pdef>
    ...</procstr>)
 ...</procstrs>
```

To select which process will be processed in the K simulation, we added the command keyword *:run* in the syntax of *CircusPar* (cf. rule (6)). As stated in rule (44), when the *:run* command is presented with an identifier naming a process model that already exists in some $< procstr >$ structure, the body PD of such process is extracted from the $< pdef >$ to the top of the computation cell, triggering its treatment. The rule also sets the $< model >$ configuration cell of the current *proc* environment to the name of the loaded model.

```
rule <k> :run ProcId:Id => PD ... </k>                              (44)
     <model> _ => ProcId </model>
     <procstr>...
        <pmodel> ProcId </pmodel>
        <pdef> PD </pdef>
     ...</procstr>
```

Once loaded in the configuration cells, the next step is to deal with the presence or absence of process paragraphs in the header of the process (cf. rules (13) and (14)). Such paragraphs may declare a state, some named actions and $\mathbb{Z}$ schemas before the main action. Rule (45) deals with a process that contains only the main anonymous action, moving this action straight to the top of the computation cell and discarding the rest of the construct.

```
rule <k> begin @ Pa:ParAction end =>  Pa ... </k>                      (45)
```

Rules (46) and (47) deal with processes with process paragraphs. Such rules simply establish the sequential processing of the *PPars* before the main action (46) and enforces the natural processing order of each process paragraph (47).

```
rule <k> begin P:PPars  @ A:ParAction end => P ~> A ... </k>   (46)
rule P1:PPar P2:PPars => P1 ~> P2                              (47)
```

Rule (48) deals with the declaration of a named action in a process paragraph. It removes the declaration from the computation cell and inserts the action body into the < *pact* > map of the correct < *procstr* > structure, using the model of the process and the name of the action as references.

Similarly, rule (49) describes how the declared $\mathbb{Z}$ schemas are processed and loaded.

```
rule                                                           (48)
  <k>N:Id ^= A:Action => . ... </k>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pact> LMap => LMap[N <- A] </pact>
  ...</procstr>
```

```
rule                                                           (49)
  <k>  N:Id == S:SchemaExp  => . ... </k>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <psch> LMap => LMap[N <- S] </psch>
  ...</procstr>
```

Consequently, a rule is needed that takes the named action definition from its storage and places its body into the top of the computation cell: rule (50) is triggered by the occurrence of a named action, i.e., the next computation is an identifier that refers to a valid named action. Valid means that it exists in the named action map < *pact* > under the < *procstr* > configuration of the model *PModel*, from the process under treatment.

```
rule                                                        (50)
  <k> A:Id => (B)  ...</k>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pact>... A |-> B ...</pact>
  ...</procstr>
```

Rule (51) that fetches a schema by its name is similar to the one for actions, but it also requires the preparation of a new environment for treating the decorated $\mathbb{Z}$ identifiers. The conditions in an operation schema, i.e., schemas that
625 may change state or local variables, can involve variable assignments from both before and after the schema application. For that reason, the current environment *env* is replicated into the *postenv* cell, and this replica can be manipulated and compared with the original *env*. If the schema condition is satisfied, the replica replaces the original copy.

```
rule                                                        (51)
  <k> A:Id => B ...</k>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <psch>... A |-> B ...</psch>
  ...</procstr>
  <env> Env </env>
  <postenv> _ => Env </postenv>
```

630 6.2 Configuration maintenance

The rules presented in this subsection are designed to maintain the structures of configuration cells, performing optimizations, and cleaning those pieces that are not required anymore, such as discarded processes and their initials.

Rule (52) is designed to delete an *ok* notification structure when there are
635 no more process references in the addressing set *to*. This is accomplished by matching the empty set (*.Set*) and rewriting the whole bag of cells to the empty cell bag (*.Bag*).

```
rule                                                        (52)
  <oks>
    (<ok>... <to>.Set</to> ...</ok> => .Bag)
  ...</oks>
```

Another rule, (53), deletes an *init* structure that became useless due to the discarding of all the processes referred in the *from* set. To check such

640 property, we require the subset inclusion $From \subset Disc$, where $Disc$ is the set of all discarded processes, registered in the *discarded* global configuration cell.

```
rule                                                              (53)
  <discarded>Disc</discarded>
  <inits>...
    (<init>... <from> From </from> ...</init> => .Bag)
  ...</inits>
  requires (From <=Set Disc)
```

The rule that removes the *proc* environments of discarded processes is shown in (54). It requires the matching of the process reference PId with an element of the set inside the *discarded* cell.

```
rule                                                              (54)
  <discarded> ... SetItem(PId) ... </discarded>
  <procs>...
    <proc>...
      <id>PId</id>
    ...</proc> => .Bag
  ...</procs>
```

645 It is possible that a discarded process leave children environments. For such cases, rule (55) performs the removal of these orphans *proc* environment and add the reference of the removed orphan to the set of discarded processes.

```
rule                                                              (55)
  <discarded>
    Disc SetItem(PId:Int) => SetItem(PId) Disc SetItem(Orphan)
  </discarded>
  <procs>...
    <proc>...
      <id>Orphan:Int</id>
      <parent>PId</parent>
    ...</proc> => .Bag
  ...</procs>
```

7 Operational semantic rules

Once the configuration is loaded and an action is present in the top of the
650 computation cell, the rewriting engine is expected to proceed accordingly to the semantics of *Circus*: it is mandatory to keep the rewrite rules faithful to the formal operational semantics. Our implementation is based on such rules as given in [CG13b] and recalled in Appendix A.

As usual, the operational semantics of *Circus*, abbreviated to OPS in the sequel, relies upon configurations and transition rules between configurations. As seen in subsection 2.2 configuration consists of three parts: a logical constraint between some symbolic variables, a state/environment definition mapping *Circus* variables into symbolic variables, and a continuation, i.e. the remaining *Circus* action. The application of an OPS rule will modify at least one of these parts. An OPS rule is defined by: a precondition, i.e. a logical expression on symbolic variables; an observable or an empty label, noted ϵ; and two configuration patterns, for the configurations before and after the rule application.

Transposed into the K-Framework, i. e. in the configuration structure presented in Section 4, the constraint is a logical expression in the *const* cell, the state/environment is a map in the *env* cell and the remaining action is processed at the k computation cell. The rewrite rules take into consideration the current configuration cells and, if the logical precondition applies, the content of the cells is rewritten according to the patterns.

The rules presented in this section are paired with some reference to the original OPS rules that motivated them. Ideally, we would have exactly one rewrite rule in K for each OPS rule. However, due to particularities of the K-framework and the complexity of the semantics of *Circus*, this was not always feasible. In such cases, we report the reasons that motivated our decisions.

7.1 Internal progress

We present here the implementation of those OPS rules that does not produce observable labels, i.e., rules that guides the progress of processes while producing an ϵ label and, consequently, does not require any appending to the symbolic trace *tr* cell. Nevertheless, they may induce the appending of specification labels to the specification trace contained in the *spectr* cell.

The assignment of variables is described by the *Circus* OPS rule A.2. There is no precondition. A new fresh symbolic value $(?W)$ is assigned to the given variable identifier V and a condition $(?W == E)$ is appended to the constraint. This condition imposes an equality relation between the new symbolic value and the expression (E) given in the right hand side of the assignment construct. As the syntax of variable assignment is tagged for strictness in its right side element, the value of the expression is evaluated before the treatment of the assignment itself. After this treatment, the action in the new configuration gets a `Skip` in place of the treated assignment.

We have two rewrite rules to accomplish this semantic behavior, due to the necessity of treating differently local and state variables. The first case is covered by rule (56), where the explicit requirement in (57) guarantees that this rule will only be applied for a variable V that is not a name in the set of keys from the state map St.

The assignment to state variables is treated by rule (58). This separation is motivated by the need of keeping the validity of the state invariant after changing the value of a state variable. After reassigning a symbolic value to a state variable, it is necessary to rewrite the state invariant into the constraint. This is

```
rule                                                        (56)
  <k> V:Id := E:Int => Skip ... </k>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pstate> St </pstate>
  ...</procstr>
  <env> Env => Env[V <- ?W:Int] </env>
  <const> Const => Const andBool (?W ==Int E) </const>
  <spectr>... . => V := E </spectr>
  requires (notBool (V in (keys(St))))                      (57)
```

simply achieved by adding these conditions and, relying on SMT optimizations, to remove the old clauses that are no more required in the configuration. To
700 trigger this extra step, we insert the construction `state Inv` in the resulting action before the `Skip` originally required by the OPS rule A.2. The computation triggered by this auxiliary construction is further described within the rules concerning state evaluations (rules (68) and (67)).

```
rule                                                        (58)
  <k> V := E:Int => state Inv ~> Skip ... </k>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pstate> ... V |-> _ ... </pstate>
    <pinv> Inv:BoolExp </pinv>
  ...</procstr>
  <env>... V |-> (_ => ?W:Int) ...</env>
  <const> Const => Const andBool (?W ==Int E) </const>
  <spectr>... . => V := E </spectr>
```

The *declaration and scope handling for local variables* in *Circus* are described
705 by OPS rule A.6, rule A.7 and rule A.8. In *Circus*, the precondition for a variable declaration concerns its type and name. As the current implementation does not handle type checking, the declared name is considered only. This precondition imposes $V \notin \alpha s$, i.e., the given name V for a variable is not already used in the alphabet αs. Since the parser already rejects identifiers that conflicts with
710 syntactical tokens, the condition of the rule we introduce in (59) only checks if V is not a name of a state variable. The *Circus* OPS also requires every declared variable to be assigned to a fresh symbolic value, reflected in K by the use of the fresh identifier generator `?W`.

OPS rule A.6 introduces the keyword `let`, to handle the scope of the de-
715 clared variable in the remaining action. In our K-Framework implementation, we have considered that it is not necessary to use such keyword since the declared variables names are available in the environment *env* cell. As defined

```
rule                                                    (59)
  <k> var V:Id @ A:Action => A ~> end V ...</k>
  <env> Env => Env[V <- ?W:Int] </env>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pstate> St </pstate>
  ...</procstr>
  <stack> . => ListItem(Env) ... </stack>
  requires (notBool (V in (keys(St))))
```

by the original OPS rule, in rule (59) we state that the declared variable scope is restricted to the action `A` specified after the `@` symbol. This is ensured by appending "`end V`" to be evaluated after the action `A`.

To guarantee the possibility of reusing variable names in nested scopes, a stack of environments is used and the current *env* configuration is pushed onto the *stack* cell at each new variable declaration. Once a scope ends, the previous environment is popped back into the *env* cell. The assignments of variables declared in the ending scope are lost. All the other assignments are kept by the `updateMap` operation, as shown in rule (60).

```
rule                                                    (60)
  <env>
    RetEnv => updateMap(Env, removeAll(RetEnv, SetItem(V)))
  </env>
  <stack> ListItem(Env) => .List ... </stack>
  <spectr>... . => end V </spectr>
```

For the *sequential composition of processes*, the *Circus* OPS includes the rule A.9 and rule A.10. The first defines the continuation of an action `A1` in the left hand side of the operator while there are computations to be performed. The OPS rule A.10 defines the continuation with the right hand side action `A2` once the computations in `A1` finishes with a `Skip`. In our implementation, the described behavior is achieved by ordering the computations of two sequentially composed actions, as presented in (61). To guarantee the correct continuation after reaching `Skip` in the first executed action, we include rule (62).

```
rule                                                    (61)
  <k> A1:Action ; A2:Action => A1 ~> A2 ... </k>
rule                                                    (62)
  <k> Skip ~> A:Action => A ... </k>
```

The behavior of *the internal choice* construction is given in *Circus* OPS rule A.11, consisting in two parts: the process unconditionally behaves as the

left hand side action or the right hand side action of the operator. This choice is made non-deterministically and, for that reason, the rules in (63) and (64) are tagged with the `[transition]` annotation. When a transition is specified, the rewriting engine marks the current step for backtracking when searching for alternative executions [ŞAL+13].

```
rule                                                          (63)
  <k> A1:Action |-| A2:Action => A1 ... </k> [transition]
rule                                                          (64)
  <k> A1:Action |-| A2:Action => A2 ... </k> [transition]
```

The *guarded actions* are defined in *Circus* OPS rule A.12. It states that the computation of an action `A` guarded by a logical condition `G` results in continuing the process with the action `A` while adding the condition specified in G to the constraint. The precondition for applying the rule requires that the constraint remains valid after adding `G`. Rule (65) strictly reproduces such behavior.

Remark (Guarded actions in specification traces)**.** *To write the specification text of a guard in a specification trace, we use an intermediary step that is not given here. It is available in the source code listing of the implementation.*

```
rule                                                          (65)
  <k> G:Bool & A:Action => A ... </k>
  <const> Const => Const andBool G </const>
  requires Const andBool G
```

7.2 State and $\mathbb{Z}$ schema

Due to its richness and its complexity, the semantics of $\mathbb{Z}$ [Spi88] would deserve to be treated in a full companion project. In absence of such a project, we decided to give semantics to a subset of $\mathbb{Z}$ schema constructs in the K-Framework, allowing the declaration and handling of states for *Circus* processes. In a nutshell, we deal with $\mathbb{Z}$ constructs where variables and formulas are written using the K predefined types and operators.

We divide the schemas into two types: the state schemas and operation schemas. A process declaration can include a single state schema only and may contain multiple operation schemas. A state schema contains the declaration of state variables and an invariant, i.e. a logical expression involving these variables that should remain always valid throughout the evolution of the process configuration. Operation schemas define state changes via pre- and post-conditions. These are logical expressions on state variables and input/output schema variables (that may reference any variable name within the current environment). Variables are possibly decorated in the post-condition: they describe the effect of the operation on these variables.

The variables declared in the state schema are treated by rule (66), processing each name V in the list of multiple declarations M. The variable names and types are inserted both into the *state* and *env* maps. Since all *Circus* variables
770 are arbitrarily initialised, a fresh symbolic value ?W is assigned to each declared variable.

```
rule                                                          (66)
  <k> state [ (V:Id , M:MultiDecl => M) | B:BoolExp ] ...</k>
  <env>... .Map =>  V |-> ?W:Int ...</env>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pstate>... .Map => V |-> "int" ...</pstate>
    <pinv> Inv:BoolExp </pinv>
  ...</procstr>
```

```
rule                                                          (67)
  <k> state [ .MultiDecl | B:BoolExp ] => state B ...</k>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pinv> _ => B </pinv>   ...</procstr>
```

Once there are no more variable declarations to be processed in the declaration part of the schema, rule (67) is applied, storing the state invariant expression B into the *inv* cell and scheduling the evaluation of the invariant for
775 the next computation step.

Rule (68) triggers the evaluation of the state invariant using the current *env* variable assignment. If the invariant is valid within the context of the constraint in *const*, the rule is applied, the invariant expression is added to the constraint and the process continues. If the invariant is invalidated, no rule applies.

```
rule                                                          (68)
  <k> state B:Bool => . ...</k>
  <const> Const => Const andBool B </const>
  requires Const andBool B
```

780 Note that rule (58) forces the computation of the invariant after the treatment of assignments to state variables, rule (67) here for operations schema.

The behavior induced by a reference to an operation schema inside a *Circus* action is given by the OPS rule A.1. In our implementation, once ready for computation, the schema is loaded according to the structural rule (51). Then,
785 rewrite rule (69) begins a sequence of steps by processing the declaration part: each variable gets a fresh symbolic value into its *postenv* cell map entry. This

is necessary for handling decorated identifiers in logical expressions involved in post-conditions.

```
rule                                                                  (69)
  <k> [ (V:Id , M:MultiDecl => M) | B:BoolExp ] ...</k>
  <postenv> Postenv => Postenv[V <- ?W:Int] </postenv>
```

Similarly as ensured by the rewrite rule (67), when the variable declaration part of an operation schema is emptied, rule (70) takes place and puts the schema logical expression `B` to be processed in the ¡k¿ cell. The computation is triggered by the internal keyword `schema` followed by the logical expression.

```
rule                                                                  (70)
  <k> [ .MultiDecl | B:BoolExp ] => schema B...</k>
```

```
rule                                                                  (71)
  <k> schema B => state Inv ~> Skip ...</k>
  <const> Const => Const andBool B </const>
  <env> _ => PostEnv </env>
  <postenv> PostEnv => .Map </postenv>
  <model> PModel:Id </model>
  <procstr>...
    <pmodel> PModel </pmodel>
    <pinv> Inv </pinv>   ...</procstr>
  requires Const andBool B
```

Rule (71) defines the evaluation of the schema logical expression `B` in the context of the current environment *env* and the post-condition environment in *postenv*. If the conjunction of the condition with the current constraint *const* is valid, the rule is applied and all the involved configuration elements are updated. The OPS rule A.3 of *Circus* operational semantics defines `Skip` as the continuation of a successful operation schema application. In rule (71), we force the reevaluation of the state invariant before `Skip`.

7.3 Observable progress

In this subsection we present the rules that model the synchronization strategy described in Section 5. Such rules are designed to post, forward and resolve the *init* structures that represents the potential initials after an action.

We refrain from presenting all the rules when subtle structural differences are necessary to handle initials for events, inputs and outputs. In such cases, only the rule for events is presented and further details can be obtained consulting the complete listing of rules.

Initial posting

The posting of an initial is necessary when the current computation involves a communication or an event. Rule (72) deals with the case of an event, following the *Circus* OPS rule A.4 for outputs, just omitting the value to be outputted. (actually, there is no specific rule for the case of event synchronisation in the version of the *Circus* operational semantics we use).

When an identifier `C` referring to a channel name is in the top of the computation cell, i.e., an event on channel `C` is required, the *sync* flag is lifted and an initial is created inside the bag *inits*. This new initial refers in cell *iresp* to the *proc* environment currently responsible for it, namely `PId`. It keeps in *iev* the name of the potential event, namely `C`. The set of subscriber processes is in *from*, initialized by the reference to the current *proc* environment, and the set of references to alternatives, in *ialt*, is initialized with the empty set.

```
rule                                                          (72)
  <k> C:Id ... </k>
  <sync> false => true </sync>
  <id>PId:Int</id>
  <ch> ... C |-> _ ... </ch>
  <inits>...
  .Bag => <init>
            <iresp>PId</iresp>
            <iev>C</iev>
            <from>SetItem(PId)</from>
            <ialt> .Set </ialt>
          </init>
  </inits>
  [transition]
```

The `transition` annotation is used because the order of unsynchronized communications is non-deterministic.

The rewriting engine should be MC: what do you mean? aware of potential different continuations that may arise from the non-determinism after the application of rule (72).

After a rule for posting the initial, we present the forwarding rules sketched in Section 5, first for channel hiding, then for parallel composition, and finally for external choice.

Channel hiding

The behavior induced by hiding observable communications of some channels is defined in *Circus* OPS rules A.26, A.27 and A.28. To correctly reproduce this behavior, we use the tree structure presented in Section 5: when a channel hiding operator is applied to an action `A`, a child *proc* environment is spawned to continue action `A`.

835 As shown in rule (73), the new environment gets a copy of all the semantically relevant configuration cells: *env*, *model* and *const*; a fresh integer reference is generated by incrementing the integer inside the global *nextid* cell; the reference to the parent *proc* environment is kept in the cell *parent*. The parent computation cell is rewritten into the internal construction `hide` with two pa-
840 rameters: the child *proc* environment reference (`L +Int 1`) and the set `CS` of channels to be hidden.

```
rule                                                    (73)
  <k>A:Action /H CS:Set
       => hide (L +Int 1) CS ... </k>
  <nextid> L => L +Int 1 </nextid>
  <id> ParId </id>
  <model> PModel:Id </model>
  <env> Env </env>
  <const> Const </const>
  (.Bag =>
    <proc>...
      <k> A </k>
      <model> PModel </model>
      <parent> ParId </parent>
      <id> (L +Int 1) </id>
      <alt> (L +Int 1) </alt>
      <env> Env </env>
      <const> Const </const>
    ...</proc>)
```

The forwarding of all the initials posted by the action in the child *proc* environment will necessarily pass through the responsibility of the parent environment for reaching upper levels and, if succeeding to reach the top level,
845 being appended to the traces. The hierarchical structure allows the design of rewrite rules to filter the initials on their way to the top level according to the OPS rules A.26 and A.27.

The case covered by OPS rule A.26 is the occurrence of an event or communication that does not involve a hidden channel, i.e., a channel that is not
850 an element of the hiding channel set `CS`. As shown in rewrite rule (74), the forwarding is performed by replacing the *iresp* reference of the matched *init* from the child `CId` to the parent `PId`.

OPS rule A.27 covers the case where the involved channel is a member of the hiding set. No observable label is produced, but the evaluation of the action
855 proceeds. Thus, in rule (75) the initial is resolved before reaching the top level environment, therefore canceling its potential writing into the traces. This removes the matched *init* cell and spawns the corresponding *ok* notification, allowing the action to continue as if the initial was actually written into the traces. The rewrite rule contains some additional conditions to avoid posting
860 *ok* notifications to discarded processes.

```
rule                                                              (74)
  <proc>...
      <k>  hide CId:Int CS:Set  ... </k>
      <id> PId:Int </id>
  ...</proc>
  <inits>...
    <init>...
      <iresp>CId => PId</iresp>
      <iev>C:Id</iev>
    ...</init>
  ... </inits>
  requires notBool (C in CS)
```

```
rule                                                              (75)
  <proc>...
    <k>  hide CId:Int CS:Set  ... </k>
    <id> PId:Int </id>
  ...</proc>
  <inits>
    (<init>...
      <iresp>CId</iresp>
      <ialt>Alt</ialt>
      <iev>C:Id</iev>
      <from>FSet</from>
     </init> => .Bag)
  ... </inits>
  <oks>...
    (.Bag =>
      <ok> <oev>C</oev> <to>FSet -Set Disc</to> </ok>)
  ... </oks>
  <discarded>Disc => Disc Alt</discarded>
  requires notBool (size(FSet -Set Disc) ==Int 0)
           andBool (C in CS)
```

The finalization of a hidden action is defined in OPS rule A.28: when the hidden action reaches a `Skip`, this closes the scope of the hidding. Thus the hiding operator and channel set are removed from the action text, which is reduced to a single `Skip`. This is ensured by rule (76), where the semantically relevant
865 configuration cells are copied from the child environment under finalization back to the parent for continuing the remaining evaluations.

Parallel composition

The parallel composition operator is defined by the *Circus* OPS rules A.20 to A.25. The evaluation of such a structure begins with the spawning of two
870 parallel actions `A1` and `A2` inside new children *proc* environments: each one will

```
rule                                                          (76)
  <k>hide CId:Int CSet:Set => Skip ... </k>
  <id> ParId </id>
  <env> _ => Env </env>
  <const> _ => Const </const>
  (<proc>...
      <k> Skip </k>
      <id>CId</id>
      <parent> ParId </parent>
      <env> Env </env>
      <const> Const </const>
    ...</proc> => .Bag)
```

```
rule                                                          (77)
  <k>
    A1:Action [|X1:Set|CSet:Set|X2:Set|] A2:Action =>
    par ListItem(L +Int 1) ListItem(L +Int 2), CSet,X1,X2
  ... </k>
  <nextid> L => L +Int 2 </nextid>
  <id> ParId </id>
  <model> PModel:Id </model>
  <env> Env </env>
  (.Bag =>
    <proc>...
      <k> A1 </k>
      <model> PModel </model>
      <parent> ParId </parent>
      <id> (L +Int 1) </id>
      <alt> (L +Int 2) </alt>
      <env> Env </env>
    ...</proc>
    <proc>...
      <k> A2 </k>
      <model> PModel </model>
      <parent> ParId </parent>
      <id> (L +Int 2) </id>
      <alt> (L +Int 1) </alt>
      <env> Env </env>
    ...</proc>)
```

correspond to the left and the right parts of the composition.

We designed the rewrite rule (77) to match OPS rule A.20. The context of each child *proc* environment replicates the semantically relevant configuration cells from the parent *proc*. The computation cell of the parent is rewritten, starting with the internal keyword `par`, followed by: the fresh references to the

left `(L +Int 1)` and right `(L +Int 2)` child *proc* environments, the set `CSet` of synchronization channels, two disjoint sets `X1` and `X2` of variable names to handle the parallel assignment of variables.

We also provide two complementary structural rules: rule (78) accepts the syntax for parallel composition with no parallel state writing; and rule (79) transforms the syntax of an interleaving in a parallel composition with no synchronizations.

```
rule                                                          (78)
  <k>
     A1:Action [|CS:Set|] A2:Action
     => A1:Action [|.Set|CS|.Set|] A2:Action
  ... </k> [structural]
```

```
rule                                                          (79)
  <k>
     A1:Action ||| A2:Action
     => A1:Action [|.Set|] A2:Action
  ... </k> [structural]
```

Once spawned, both actions are evualated independently until the posting of an initial occurs. Forwarding of initials inside a parallel composition takes into consideration the synchronization of the concurrent actions when there are events or communications over channels in the `CSet` synchronization set.

Initials involving a channel name `C` that is not in `CSet` are forwarded according to the OPS rules A.22 and A.23, i.e., they are forwarded immediately. Since we do not discriminate which side of the parallel composition is treated, only one rewrite rule is necessary for defining the behavior of both OPS rules. Rule (80) matches a *proc* environment `CId` that is a child of a parallel composition environment `PId` and has an initial that respects the condition `notBool (C in CSet)`. Once matched, the rule forwards it, rewriting the responsible *iresp* from the child to the parent (`CId => PId`).

```
rule                                                           (80)
  <proc>...
    <id>CId:Int</id>
    <parent>PId:Int</parent>
  ...</proc>
  <proc>...
    <id>PId</id>
    <k> par _:List,CSet:Set,_:Set,_:Set ... </k>
  ...</proc>
  <inits>...
    <init>...
      <iresp>CId => PId</iresp>
      <iev>C:Id ...</iev>
    ...</init>
  ... </inits>
  requires notBool (C in CSet)
```

895 For initials over channel names in the synchronization set, the forwarding is postponed until there is another initial, posted by the environment at the opposite side of the parallel composition, configuring one of the synchronization behaviors as defined in OPS rulesA.24 or A.25.

Rule (81) handles the forwarding of two synchronized initials: it matches 900 two children environment, the left side `LId` and the right side `RId`; a parent parallel composition *proc* environment `PId`; and two initials, one for each child *proc*. The two initials involved in a synchronization are merged into a single one that will carry in its *from* and *alt* sets all the references to the standing by environments from both. The initial carrying all the references is forwarded 905 and the other is removed.

```
rule                                                              (81)
  <proc>...
    <id>LId:Int</id>
    <parent>PId:Int</parent>
  ...</proc>
  <proc>...
    <id>RId:Int</id>
    <parent>PId:Int</parent>
  ...</proc>
  <proc>...
    <id>PId</id>
    <k> par _:List,CSet:Set,_:Set,_:Set ... </k>
  ...</proc>
  <inits>...
    <init>...
      <iresp>LId => PId</iresp>
      <iev>C:Id ...</iev>
      <ialt> LAlt => LAlt RAlt </ialt>
      <from> LFrom => LFrom RFrom </from>
    ...</init>
    (<init>...
      <iresp>RId</iresp>
      <iev>C ...</iev>
      <ialt>RAlt</ialt>
      <from>RFrom</from>
    ... </init> => .Bag)
  ... </inits>
  requires (C in CSet)
```

An input/output synchronization, as defined in OPS rule A.24, establishes the communication of a value between the two sides of a parallel composition. Such cases are treated by rule (82). We omit the *proc* environment matching part, since it is identical to rule (81). The difference is the explicit requirement 910 of matching an input value with an output value over the same synchronized channel. The initials are merged within the output communication, since the output label is the correct observable behavior accordingly to the *Circus* OPS.

```
rule                                                                    (82)
  (...)
  <inits>...
    <init>...
      <iresp>LId => PId</iresp>
      <iev>C:Id ! E:Exp</iev>
      <ialt> LAlt => LAlt RAlt </ialt>
      <from> LFrom => LFrom RFrom </from>
    ...</init>
    (<init>...
      <iresp>RId</iresp>
      <iev>C:Id ? _</iev>
      <ialt>RAlt</ialt>
      <from>RFrom</from>
    ... </init> => .Bag)
  ... </inits>
  requires (C in CSet)
```

The finalization of a parallel composition is given in the OPS rule A.21. When both parallel actions reach their final `Skip`, the whole parallel composition 915 continues as a single `Skip` and the variable assignments performed in the parallel branches are merged accordingly to the name sets `X1` and `X2`. Only assignments made in the left (resp. right) side to variable names listed in `X1` (resp. `X2`) are kept.

Rule (83) performs this step, removing the two finished children environ- 920 ments and merging both assignment maps in the *env* cells into the parent *proc* environment, taking into consideration the name sets `X1` and `X2`.

```
rule                                                        (83)
  (<proc>...
    <id>LId:Int</id>
    <k> Skip </k>
    <parent>PId:Int</parent>
    <const> LConst </const>
    <env> LEnv </env>
  ...</proc> => .Bag )
  (<proc>...
    <id>RId:Int</id>
    <k> Skip </k>
    <parent>PId:Int</parent>
    <const> RConst </const>
    <env> REnv </env>
  ...</proc> => .Bag )
  <proc>...
    <id>PId</id>
    <const> Const =>
        Const andBool LConst andBool RConst
    </const>
    <env> Env =>
      updateMap(Env,
        removeAll(LEnv, keys(LEnv) -Set X1)
        removeAll(REnv, keys(REnv) -Set X2)
      )
    </env>
    <k> par ListItem(LId) ListItem(RId),
        CSet:Set, X1:Set, X2:Set => Skip ...</k>
  ...</proc>
```

External choice

Starting an external choice, as defined in the OPS rule A.13, is similar to starting a parallel composition: the *Circus* process is split into two actions and, as given in
925 OPS rules A.16 and A.17, they can proceed until an initial is posted. Rule (84) implements the behavior of OPS rule A.13, spawning two *proc* environments to evaluate each action, and rewriting the computation cell into the internal keyword `choice` followed by a set of two elements containing the fresh references to the spawned environments. Actually, the *Circus* OPS rules A.16 and A.17 do
930 not require an explicit rewrite rule, since they deal with internal progress, and internal progress does not involves initial handling as seen in subsection 7.1

```
rule                                                                    (84)
  <k> A1:Action [] A2:Action =>
      choice SetItem(L +Int 1) SetItem(L +Int 2) ... </k>
  <nextid> L => L +Int 2 </nextid>
  <id>PId:Int</id>
  <model> PModel:Id </model>
  <env> Env </env>
  <const> Const </const>
  (.Bag =>
    <proc>...
      <k> A1 </k>
      <model> PModel </model>
      <parent>PId</parent>
      <env> Env </env>
      <const> Const </const>
      <id> (L +Int 1) </id>
      <alt> (L +Int 2) </alt>
    ...</proc>
    <proc>...
      <k> A2 </k>
      <model> PModel </model>
      <env> Env </env>
      <const> Const </const>
      <id> (L +Int 2) </id>
      <alt> (L +Int 1) </alt>
      <parent>PId</parent>
    ...</proc>)
```

As stated by OPS rule A.18 and rule A.19, once an observable event or communication is performable at one of sides of the external choice, the other side is no more considered. The *Circus* process continues with the observable event or
935 communication and its continuation. To achieve such a behavior, the forwarding of initials inside an external choice is handled by rule (85). Initials are forwarded as soon as they are posted by one of the sides. For each forwarded initial, we mark in the set *alt* of alternatives the reference to the *proc* environment in the other side of the choice. When an initial with a non-empty set of alternatives
940 is resolved, all *proc* environments referred in the set are discarded and only the chosen side of each external choice gets an *ok* notification to proceed.

```
rule                                                         (85)
  <proc>...
    <id>PId</id>
    <k> choice Set ... </k>
  ...</proc>
  <proc>...
    <id>CId</id>
    <parent>PId</parent>
    <alt>AId</alt>
  ...</proc>
  <inits>...
    <init>...
      <iresp>CId => PId</iresp>
      <ialt>... .Set => SetItem(AId) ...</ialt>
    ... </init>
  ... </inits>
```

The finalization of an external choice is given by the OPS rules A.14 and A.15. When one of the choice sides reaches its final `Skip`, the external choice ends, and the *Circus* process continues with the assignments and constraint from 945 the successfully finished side. By using the `transition` annotation, we capture the behavior of both OPS rules with rule (86).

```
rule                                                         (86)
  <procs>...
    <proc>...
      <id>PId:Int</id>
      <k> choice _ => Skip ... </k>
      <env> _ => Env </env>
      <const> _ => Const </const>
    ...</proc>
    (<proc> ...
      <id>P1:Int</id>
      <parent>PId</parent>
      <alt>AId:Int</alt>
      <k>Skip</k>
      <env> Env </env>
      <const> Const </const>
    ...</proc> => .Bag)
  ...</procs>
  <discarded>...
    .Set => SetItem(AId)
  ...</discarded>
  [transition]
```

Initial resolution

As explained in Section 5, the resolution of an initial is the end of its life-cycle. It is removed from the set of initials and two consequences may follow: the
950 appending of its content to the traces and the generation of an *ok* notification to allow the continuation of *proc* environments affected by this initial.

The resolution rewrite rules are triggered in two cases. The first one is when an initial is embedded in the scope of a hiding operation. This case is treated in rule (75). The other one is when an initial has reached the top level, i.e.
955 its *iresp* sub-cell references the *proc* Id 0 environment. It is treated in rule (87), which ensures the removal of the *init*, the appending of the event *iev* to the trace *tr* and specification trace *spectr* and, finally, the spawning of the *ok* notification, addressed to all the *proc* environments listed in the set *from* of the resolving *init*. Extra precautions are taken to avoid the resolution of an initial
960 that concerns only discarded processes, such cases are treated by the structural rule in (53).

```
rule                                                          (87)
  <id>0</id>
  (<init>...
    <iresp>0</iresp>
    <ialt>Alt</ialt>
    <iev>C:Id</iev>
    <from>FSet</from>
  </init> => .Bag)
  <oks>...
    (.Bag => <ok>
      <oev>C</oev>
      <to>FSet -Set Disc</to>
    </ok>)
  ... </oks>
  <tr>... . => C </tr>
  <spectr>... . => C </spectr>
  <discarded>Disc => Disc Alt</discarded>
  requires notBool (size(FSet -Set Disc) ==Int 0)
```

When an *ok* notification is available and addressed to a standing-by *proc* environment, i.e., holding true in its *sync* flag sub-cell, rule (88) is triggered. It matches such standing-by *proc* `PId` referenced in an *ok* structure and its event
965 *oev* with the *proc* current computation. As result, the current computation is concluded and the *sync* flag is set to false, allowing the continuation of the remaining computations in *proc*. The reference `PId` is also removed from the *to* set of the *ok* notification. The notification itself remains available to match possible other *proc* environments in the remaining references. When the *to* set
970 becomes empty, the structural rule in (52) removes the *ok* structure from the configuration.

```
rule                                                                    (88)
  <k> C:Id => . ... </k>
  <sync> true => false </sync>
  <id>PId:Int</id>
  <oks>...
    <ok>
      <oev>C:Id</oev>
      <to>... SetItem(PId) => .Set ...</to>
    </ok>
  ... </oks>
```

8 Examples

This section presents some examples of *Circus* specifications that we used for checking the resulting traces from our K-Framework implementation against the ones expected following the formal operational semantics.

The first example, shown in Figure 9, specifies a *Circus* process containing in the main anonymous action a parallel composition with a defined synchronization set and two concurrent external choices. The initials of events from channels that are not elements of the synchronization set are forwarded as soon as they are posted. The non-deterministic search finds all the possible solutions, including the partial ones, i.e., traces which do not reach the final `Skip`.

```
channel a1,a2,a3,b1,b2,b3,c1,c2
channelset SyncSet == {|a1,b1|}
process proc3 ^=
begin
  @
  ((b2 -> b3 -> Skip) [] (a1 -> a2 -> a3 -> Skip))
    [|SyncSet|]
  ((c1 -> b1 -> Skip) [] (c2 -> a1 -> Skip))
end
```

Figure 9: Synchronization test specification

The output is shown in Figure 10, listing all the solutions found and combining all the possible event orders caused by the non-determinism. The `Solution 6` shows the only trace that reaches the final `Skip`.

```
Search results:
Const:Bool --> true

Solution 1: b2 ~> b3 ~> c1
Solution 2: b2 ~> b3 ~> c2
Solution 3: b2 ~> c1 ~> b3
Solution 4: b2 ~> c2 ~> b3
Solution 5: c1 ~> b2 ~> b3
Solution 6: c2 ~> a1 ~> a2 ~> a3 ~> Skip
Solution 7: c2 ~> b2 ~> b3
```

Figure 10: Synchronization test output result

The specification shown in Figure 11 exercises the variable scope handling. The scope of the input variable `x` is checked before and after the redeclaration

that occurs inside the named action `Act`.

```
channel c
process Scope ^=
begin
  Act ^= c?x -> c!x -> Skip
  @
  c?x -> (c!x -> Act; c!x -> Skip)
end
```

Figure 11: Variable scope test specification

As defined in the operational semantics, the symbolic value assigned to a previously declared variable name is restored with the ending of the scope that 990 contains the redeclaration. The output is shown in Figure 12, including the constraint and tracking of scope ending in the specification trace.

```
Search results:
Solution 1:
Const:Bool -->
V0 ==K V1 andBool V2 ==K V3
SpecTrace:K -->
( c ?? x ) ~> ( c !! x ) ~> ( c ?? x ) ~> ( c !! x )
~> ( end x ) ~> ( c !! x ) ~> end x
Trace:K -->
( c ?? V1 ) ~> ( c ! V0 ) ~> ( c ?? V3 ) ~> ( c ! V2 )
~> ( c ! V0 )
```

Figure 12: Scope test output result

In Figure 13 we show a specification that exercises the concurrent assignment to state variables. The main action begins by assigning 0 to all the three variable names. Then it splits into two parallel actions: the left part is allowed to write 995 to `x` and the right part is allowed to write to `z`. The left and right parts assigns, respectively, 1 and 2 to all the three variable names. The parallel composition finishes with the assignments and the main action continues communicating the values of the three variables through the channel `out`.

```
channel out
process ParState ^=
begin
  state myState == [x,y,z]
  @
  x:=0;y:=0;z:=0;
  ((x:=1;y:=1;z:=1) [|{|x|}|{||}|{|y|}|] (x:=2;y:=2;z:=2));
  out!x -> out!y -> out!z -> Skip
end
```

Figure 13: Parallel state assignment specification text

The generated output can be seem in Figure 14, showing a result accordingly to the expected: the sequence of the three `out` communications containing the values of `x`, `y` and `z`, respectively, 1, 2 and 0. It means that `x` and `y` were changed by the left and right parts of the parallel composition while `z` remained with its original assignment. The output is given by symbolic values that are referenced in the constraint `Const`.

```
Search results:
Solution 1:
Const:Bool -->
V1 ==K 0 andBool V0 ==K 1 andBool V2 ==K 2
SpecTrace:K -->
( x := 0 ) ~> ( y := 0 ) ~> ( z := 0 ) ~> ( x := 1 )
~> ( x := 2 ) ~> ( y := 2 ) ~> ( z := 2 ) ~> ( y := 1 )
~> ( z := 1 ) ~> ( out !! x ) ~> ( out !! y ) ~> ( out !! z )
Trace:K -->
( out ! V0 ) ~> ( out ! V2 ) ~> ( out ! V1 ) ~> Skip
```

Figure 14: Parallel state assignment output result

The example *Circus* process in Figure 15 specifies the behavior of a Fibonacci sequence generator as introduced in [CMW13]. It uses operation schema to initialize and update the state variables. Since `OutFib` is a recursive action, the traces are unbounded and it is necessary to limit the number of rewrite steps we are going to apply in the evaluation.

```
channel out

process Fibonacci ^=
begin
  state FibState == [x,y]
  InitFibState == [x',y' | x' == y' && x' == 1 ]
  OutFibState == [y',x' | y' == (x + y) && x' == y ]
  InitFib ^= out!1 -> out!1 -> InitFibState
  OutFib ^= out!(x+y) -> OutFibState; OutFib
  @
  InitFib; OutFib
end
```

Figure 15: *Fibonacci* specification text

1010 In Figure 16 we show the output arbitrarily limited to 500 rewrite steps. The resulting trace is consistent with the operational semantics, as it was discussed in [CMW13].

```
Search results:
Solution 1:
Const:Bool -->
true
SpecTrace:K -->
( out ! 1 ) ~> ( out ! 1 ) ~> InitFibState
~> (out !! ( x + y )) ~> OutFibState ~> (out !! ( x + y ))
~> OutFibState ~> (out !! ( x + y )) ~> OutFibState
~> (out !! ( x + y )) ~> OutFibState ~> (out !! ( x + y ))
~> OutFibState  ~> (out !! ( x + y )) ~> OutFibState
~> (out !! ( x + y )) ~> OutFibState ~> (out !! ( x + y ))
~> OutFibState ~> (out !! ( x + y ))
Trace:K -->
( out ! 1 ) ~> ( out ! 1 ) ~> ( out ! 2 ) ~> ( out ! 3 )
~> ( out ! 5 ) ~> ( out ! 8 ) ~> ( out ! 13 ) ~> ( out ! 21 )
~> ( out ! 34 ) ~> ( out ! 55 ) ~> ( out ! 89 )
```

Figure 16: *Fibonacci* output result

The chronometer specification, shown in Figure 17, was introduced in [ACGS16]. It contains three *Circus* processes: the *Chrono*, that keeps in its state the count 1015 of elapsed minutes and seconds; the *Ticker*, that synchronizes with *Chrono* in the *tick* channel, causing the increment of seconds, and in *time* channel, causing the output of the current number of elapsed minutes and seconds; and *Clock*,

that parallel composes *Chrono* and *Ticker*, setting the synchronization channels and hiding the output of the channel *tick*.

```
channel out,tick,time

process Chrono ^=
begin
  state AState == [sec,min]
  AInit == [min,sec | sec' == min' && min' == 0]
  IncSec == [min,sec | sec' == ((sec+1) mod 60) && min' == min]
  IncMin == [min,sec | min' == ((min+1) mod 60) && sec' == sec]
  Run ^= (tick -> IncSec;
            (((sec==0) & IncMin) [] ((sec != 0) & Skip))
         )
         []
         (time -> out!min -> out!sec -> Skip)
  @
  AInit; u X @ (Run; X)
end

process Ticker ^= u X @ (tick -> time; X)
process Clock ^= (Chrono [|{|tick,time|}|] Ticker) /H {|tick|}

:run Clock
```

Figure 17: *Chrono* specification text

1020 The line `:run Clock` defines *Clock* as the process to be evaluated. We shown the resulting output in Figure 18, that is correct in consideration to the operational semantics analysis presented in [ACGS16].

```
Search results:
Solution 1:
Const:Bool -->
true
SpecTrace:K -->
AInit ~> IncSec ~> ( sec != 0 ) ~> time ~> ( out !! min )
~> ( out !! sec ) ~> IncSec ~> ( sec != 0 ) ~> time
~> ( out !! min ) ~> ( out !! sec ) ~> IncSec ~>
(...)
~> time ~> ( out !! min ) ~> ( out !! sec ) ~> IncSec
~> ( sec == 0 ) ~> IncMin ~> time ~> ( out !! min )
~> ( out !! sec ) ~> IncSec ~> ( sec != 0 ) ~> time
~> ( out !! min ) ~> ( out !! sec ) ~> (...)
Trace:K -->
time ~> ( out ! 0 ) ~> ( out ! 1 ) ~> time
~> ( out ! 0 ) ~> ( out ! 2 ) ~> time ~> ( out ! 0 )
~> ( out ! 3 ) ~> time ~> ( out ! 0 ) ~>
(...)
~> time ~> ( out ! 0 ) ~> ( out ! 59 ) ~> time
~> ( out ! 1 ) ~> ( out ! 0 ) ~> time ~> ( out ! 1 )
~> ( out ! 1 ) ~> (...)
```

Figure 18: *Chrono* output result

9 Conclusions and future work

In this work, we have established a set of rewrite rules within the K-Framework, which automates the behavior of the transition system defined by the formal operational semantics of *Circus*. We also embedded a subset of the operational semantics of the *Circus* specification traces.

The current result is an executable interpreter, which accepts an input specification text and enumerate its constrained symbolic traces and specification traces. Presently, the tool is the first symbolic specification traces generator for the *Circus* language.

Beyond providing interpreters based on formal semantics, the K-Framework has proven to be usable for other static analysis and state-exploration purposes: once introduced the rewrite rules of the operational semantics, it is possible to extend the configuration and the structural rules in order to develop a wide class of tools. Pursuing the original motivation of our work, as described in [ACGS16], we plan to exploit the symbolic trace generators for the development of a test platform for *Circus* specification. With this goal in mind, we have plans to work on guided trace generation, aiming at ensuring the pertinence of a trace set with respect to a testing criterion (for instance, the coverage of mutations in a

specification text).

We also plan to alleviate some current limitations of the tool: covering the whole specification traces semantics, handling multivalued communications, and accepting $\mathbb{Z}$ schema expressions. Moreover, the current version of the implementation does not includes type checking, which is easily supported by the K-Framework. This is another considered extension.

Most of the rewrite rules are very close to their counterparts in the operational semantics. Nevertheless, some adaptations were needed and they complicate the establishment of the correctness of the implementation. Clearly, some proofs need to be conducted.

When dealing with *Circus*, parts of CSP and $\mathbb{Z}$ semantics were introduced in the rule set. It would be an interesting exercise to implement the whole semantics for both specification languages in the K-Framework. And more generally, we believe that this piece of work brings some insights to the automation of process algebra operational semantics in the K-Framework.

Acknowledgments

We are grateful to the Brazilian Funding Agency CNPq (Grant 400834/2014-6) for supporting this project, and to LRI (Laboratoire de Recherche en Informatique, Univ. Paris-Sud and CNRS) for hosting Alex Alberto during this work.

References

[ACGS16] Alex Alberto, Ana Cavalcanti, Marie-Claude Gaudel, and Adenilso Simão. Formal mutation testing for circus. *Information and Software Technology*, 2016.

[CG10] A. L. C. Cavalcanti and M.-C. Gaudel. Specification Coverage for Testing in *Circus*. In S. Qin, editor, *Unifying Theories of Programming*, volume 6445 of *Lecture Notes in Computer Science*, pages 1–45. Springer-Verlag, 2010.

[CG11a] A. L. C. Cavalcanti and M.-C. Gaudel. Testing for Refinement in *Circus*. *Acta Informatica*, 48(2):97–147, 2011.

[CG11b] Ana Cavalcanti and Marie-Claude Gaudel. Testing for refinement in circus. *Acta informatica*, 48(2):97–147, 2011.

[CG13a] A. L. C. Cavalcanti and M.-C. Gaudel. Data-flow coverage for testing in Circus. Technical Report 1567, LRI, Universit'e Paris-Sud XI, December 2013.

[CG13b] A. L. C. Cavalcanti and M.-C. Gaudel. Data Flow Coverage of *Circus* Specifications-Extended Version. Technical Report 1565, LRI, Université Paris-Sud XI, September 2013.

[CG14] A. L. C. Cavalcanti and M.-C. Gaudel. Data Flow coverage for *Circus*-based testing. In *Fundamental Approaches to Software Engineering*, volume 8441 of *Lecture Notes in Computer Science*, pages 415–429, 2014.

[CLRR16] Stefan Ciobaca, Dorel Lucanu, Vlad Rusu, and Grigore Rosu. A language-independent proof system for full program equivalence. *Formal Aspects of Computing*, 28(3):469–497, May 2016.

[CMW13] A. L. C. Cavalcanti, A. Mota, and J. C. P. Woodcock. Simulink timed models for program verification. In Z. Liu, J. C. P. Woodcock, and H. Zhu, editors, *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, volume 8051 of *Lecture Notes in Computer Science*, pages 82–99. Springer, 2013.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *International conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340. Springer, 2008.

[FGW13] A Feliachi, Marie-Claude Gaudel, and B Wolf. Exhaustive testing in hol-testgen/cirta - a case study. Technical Report 1562, Universit Paris Sud - LRI, 2013.

[HJ98] C. A. R. Hoare and He Jifeng. *Unifying Theories of Programming*. Prentice-Hall, 1998.

[Hoa85] C. A. R. Hoare. *Communicating sequential processes*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1985.

[Kin76] James C. King. Symbolic execution and program testing. *Commun. ACM*, 19(7):385–394, July 1976.

[Mor94] C. C. Morgan. *Programming from Specifications*. Prentice-Hall, 2nd edition, 1994.

[OCW09] M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1-2):3–32, 2009.

[PW02] Lawrence C Paulson and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer, 2002.

[Ros98] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall Series in Computer Science. Prentice-Hall, 1998.

[Ros07] Grigore Rosu. K: A rewriting-based framework for computations–preliminary version–. Technical report, University of Illinois at Urbana-Champaign, 2007.

1115 [RŞ10] Grigore Roşu and Traian Florin Şerbănuţă. An overview of the K semantic framework. *Journal of Logic and Algebraic Programming*, 79(6):397–434, 2010.

[RS14] Grigore Rosu and Traian Florin Serbanuta. K overview and simple case study. In *Proceedings of International K Workshop (K'11)*, 1120 volume 304 of *ENTCS*, pages 3–56. Elsevier, June 2014.

[SA11] Bryan Scattergood and Philip Armstrong. *CSPM: A Reference Manual*, January 2011.

[ŞAL+13] Traian Florin Şerbănuţa, Andrei Arusoaie, David Lazar, Chucky Ellison, Dorel Lucanu, and Grigore Roşu. The K primer (version 3.2). 1125 Technical report, K-Framework team, 2013.

[Spi88] J Michael Spivey. *Understanding Z: A formal specification language and its formal semantics*, volume 3. Cambridge University Press, 1988.

[WD96] J. C. P. Woodcock and J. Davies. *Using Z—Specification, Refine-* 1130 *ment, and Proof.* Prentice-Hall, 1996.

A Operational semantics: table of selected transition rules

$$\frac{\mathtt{c} \land (\mathtt{s};\ \mathtt{p}) \land (\exists\, \mathtt{v}' \bullet \mathtt{s};\ \mathtt{Q})}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{p} \vdash \mathtt{Q}) \xrightarrow{\epsilon} (\mathtt{c} \land (\mathtt{s};\ \mathtt{Q}\,[\mathtt{w}_0/\mathtt{v}']) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w}_0 \models \mathtt{Skip})} \quad v' = out\alpha s \tag{A.1}$$

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{v} := \mathtt{e}) \xrightarrow{\epsilon} (\mathtt{c} \land (\mathtt{s};\ \mathtt{w}_0 = \mathtt{e}) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w}_0 \models \mathtt{Skip})} \tag{A.2}$$

1135

$$\frac{c \land (s;\ \mathbf{pre}\ Op)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{Op}) \xrightarrow{\epsilon} (\mathtt{c} \land (\mathtt{s};\ \mathtt{Op}\,[\mathtt{w}_0/\mathtt{v}']) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w}_0 \models \mathtt{Skip})} \quad v' = out\alpha s \tag{A.3}$$

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d!e} \to \mathtt{A}) \xrightarrow{\mathtt{d!w_0}} (\mathtt{c} \land (\mathtt{s};\ \mathtt{w}_0 = \mathtt{e}) \mid \mathtt{s} \models \mathtt{A})} \tag{A.4}$$

$$\frac{c \land T \neq \varnothing \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d?x} : \mathtt{T} \to \mathtt{A}) \xrightarrow{\mathtt{d?w_0}} (\mathtt{c} \land \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var}\,\mathtt{x} := \mathtt{w}_0 \models \mathtt{let}\ \mathtt{x} \bullet \mathtt{A})} \tag{A.5}$$

$$\frac{c \land T \neq \varnothing \qquad x \notin \alpha s}{(\mathsf{c} \mid \mathsf{s} \models \mathtt{var}\, \mathtt{x} : \mathtt{T} \bullet \mathtt{A}) \xrightarrow{\epsilon} (\mathsf{c} \land \mathtt{w}_0 \in \mathtt{T} \mid \mathsf{s};\ \mathtt{var}\, \mathtt{x} := \mathtt{w}_0 \models \mathtt{let}\ \mathtt{x} \bullet \mathtt{A})} \tag{A.6}$$

$$\frac{(\mathsf{c}_1 \mid \mathsf{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathsf{c}_2 \mid \mathsf{s}_2 \models \mathtt{A}_2)}{(\mathsf{c}_1 \mid \mathsf{s}_1 \models \mathtt{let}\ \mathtt{x} \bullet \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathsf{c}_2 \mid \mathsf{s}_2 \models \mathtt{let}\ \mathtt{x} \bullet \mathtt{A}_2)} \tag{A.7}$$

$$\frac{c}{(\mathsf{c} \mid \mathsf{s} \models \mathtt{let}\ \mathtt{x} \bullet \mathtt{Skip}) \xrightarrow{\epsilon} (\mathsf{c} \mid \mathsf{s};\ \mathtt{end}\, \mathtt{x} \models \mathtt{Skip})} \tag{A.8}$$

1140

$$\frac{(\mathsf{c}_1 \mid \mathsf{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathsf{c}_2 \mid \mathsf{s}_2 \models \mathtt{A}_2)}{(\mathsf{c}_1 \mid \mathsf{s}_1 \models \mathtt{A}_1;\ \mathtt{B}) \xrightarrow{\mathtt{l}} (\mathsf{c}_2 \mid \mathsf{s}_2 \models \mathtt{A}_2;\ \mathtt{B})} \tag{A.9}$$

$$\frac{c}{(\mathsf{c} \mid \mathsf{s} \models \mathtt{Skip};\ \mathtt{A}) \xrightarrow{\epsilon} (\mathsf{c} \mid \mathsf{s} \models \mathtt{A})} \tag{A.10}$$

$$\frac{c}{(\mathsf{c} \mid \mathsf{s} \models \mathtt{A}_1 \sqcap \mathtt{A}_2) \xrightarrow{\epsilon} (\mathsf{c} \mid \mathsf{s} \models \mathtt{A}_1)} \qquad \frac{c}{(\mathsf{c} \mid \mathsf{s} \models \mathtt{A}_1 \sqcap \mathtt{A}_2) \xrightarrow{\epsilon} (\mathsf{c} \mid \mathsf{s} \models \mathtt{A}_2)} \tag{A.11}$$

$$\frac{c \land (s;\ g)}{(\mathsf{c} \mid \mathsf{s} \models \mathsf{g} \,\&\, \mathtt{A}) \xrightarrow{\epsilon} (\mathsf{c} \land (\mathsf{s};\ \mathsf{g}) \mid \mathsf{s} \models \mathtt{A})} \tag{A.12}$$

$$\frac{c}{(\mathsf{c} \mid \mathsf{s} \models \mathtt{A}_1 \,\square\, \mathtt{A}_2) \xrightarrow{\epsilon} (\mathsf{c} \mid \mathsf{s} \models (\mathsf{loc}\ \mathsf{c} \mid \mathsf{s} \bullet \mathtt{A}_1) \boxplus (\mathsf{loc}\ \mathsf{c} \mid \mathsf{s} \bullet \mathtt{A}_2))} \tag{A.13}$$

1145

$$\frac{c_1}{(\mathsf{c} \mid \mathsf{s} \models (\mathsf{loc}\ \mathsf{c}_1 \mid \mathsf{s}_1 \bullet \mathbf{Skip}) \boxplus (\mathsf{loc}\ \mathsf{c}_2 \mid \mathsf{s}_2 \bullet \mathtt{A})) \xrightarrow{\epsilon} (\mathsf{c}_1 \mid \mathsf{s}_1 \models \mathbf{Skip})} \tag{A.14}$$

$$\frac{c_2}{(\mathsf{c} \mid \mathsf{s} \models (\mathsf{loc}\ \mathsf{c}_1 \mid \mathsf{s}_1 \bullet \mathtt{A}) \boxplus (\mathsf{loc}\ \mathsf{c}_2 \mid \mathsf{s}_2 \bullet \mathbf{Skip})) \xrightarrow{\epsilon} (\mathsf{c}_2 \mid \mathsf{s}_2 \models \mathbf{Skip})} \tag{A.15}$$

$$
\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\epsilon} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)}
{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \\ \boxplus \\ (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ \mathtt{c}_3 \mid \mathtt{s}_3 \bullet \mathtt{A}_3) \\ \boxplus \\ (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right)} \tag{A.16}
$$

$$
\frac{(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\epsilon} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)}
{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \\ \boxplus \\ (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} \left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{l} (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \\ \boxplus \\ (\mathsf{loc}\ \mathtt{c}_3 \mid \mathtt{s}_3 \bullet \mathtt{A}_3) \end{array}\right) \end{array}\right)} \tag{A.17}
$$

$$
\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad \mathtt{l} \neq \epsilon}
{(\mathtt{c} \mid \mathtt{s} \models (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \boxplus (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2)) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)} \tag{A.18}
$$

$$
\frac{(\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad \mathtt{l} \neq \epsilon}
{(\mathtt{c} \mid \mathtt{s} \models (\mathsf{loc}\ \mathtt{c}_1 \mid \mathtt{s}_1 \bullet \mathtt{A}_1) \boxplus (\mathsf{loc}\ \mathtt{c}_2 \mid \mathtt{s}_2 \bullet \mathtt{A}_2)) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3)} \tag{A.19}
$$

$$
\frac{c}
{(\mathtt{c} \mid \mathtt{s} \models \mathtt{A}_1 \llbracket\, \mathtt{x}_1 \mid \mathtt{cs} \mid \mathtt{x}_2 \,\rrbracket\, \mathtt{A}_2) \xrightarrow{\epsilon} \left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \ (\mathtt{par}\ \mathtt{s} \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \llbracket\, \mathtt{cs} \,\rrbracket (\mathtt{par}\ \mathtt{s} \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right)} \tag{A.20}
$$

$$
\frac{c}
{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{Skip}) \\ \llbracket \mathtt{cs} \rrbracket \\ (\mathtt{par}\ \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{Skip}) \end{array}\right) \end{array}\right) \xrightarrow{\epsilon} (\mathtt{c} \mid (\exists\, \mathtt{x}_2' \bullet \mathtt{s}_1) \wedge (\exists\, \mathtt{x}_1' \bullet \mathtt{s}_2) \models \mathtt{Skip})} \tag{A.21}
$$

$$
\frac{(\mathtt{c} \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad \mathtt{l} = \epsilon \vee \mathrm{chan}\, \mathtt{l} \notin cs}
{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \\ \llbracket \mathtt{cs} \rrbracket \\ (\mathtt{par}\ \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\mathtt{l}} \left(\begin{array}{l} \mathtt{c}_3 \mid \mathtt{s} \\ \models \\ \quad \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_3 \mid \mathtt{x}_1 \bullet \mathtt{A}_3) \\ \llbracket \mathtt{cs} \rrbracket \\ (\mathtt{par}\ \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right)} \tag{A.22}
$$

$$\frac{(\mathtt{c} \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\mathtt{l}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad \mathtt{l} = \epsilon \lor \mathrm{chan}\,\mathtt{l} \notin cs}{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\mathtt{l}} \left(\begin{array}{l} \mathtt{c}_3 \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{s}_3 \mid \mathtt{x}_2 \bullet \mathtt{A}_3) \end{array}\right) \end{array}\right)} \tag{A.23}$$

1155

$$\frac{\left(\begin{array}{c} (\mathtt{c} \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{d?w_1}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \ \land \ (\mathtt{c} \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\mathtt{d!w_2}} (\mathtt{c}_4 \mid \mathtt{s}_4 \models \mathtt{A}_4) \\ \lor \\ (\mathtt{c} \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{d!w_1}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \ \land \ (\mathtt{c} \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\mathtt{d?w_2}} (\mathtt{c}_4 \mid \mathtt{s}_4 \models \mathtt{A}_4) \\ \lor \\ (\mathtt{c} \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{d!w_1}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \ \land \ (\mathtt{c} \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\mathtt{d!w_2}} (\mathtt{c}_4 \mid \mathtt{s}_4 \models \mathtt{A}_4) \end{array}\right) \\ \mathtt{d} \in cs \quad c_3 \land c_4 \land w_1 = w_2}{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\mathtt{d!w_2}} \left(\begin{array}{l} \mathtt{c}_3 \land \mathtt{c}_4 \land \mathtt{w}_1 = \mathtt{w}_2 \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_3 \mid \mathtt{x}_1 \bullet \mathtt{A}_3) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{s}_4 \mid \mathtt{x}_2 \bullet \mathtt{A}_4) \end{array}\right) \end{array}\right)} \tag{A.24}$$

$$\frac{(\mathtt{c} \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{d?w_1}} (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \quad (\mathtt{c} \mid \mathtt{s}_2 \models \mathtt{A}_2) \xrightarrow{\mathtt{d?w_2}} (\mathtt{c}_4 \mid \mathtt{s}_4 \models \mathtt{A}_4) \\ \mathtt{d} \in cs \quad c_3 \land c_4 \land w_1 = w_2}{\left(\begin{array}{l} \mathtt{c} \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_1 \mid \mathtt{x}_1 \bullet \mathtt{A}_1) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{s}_2 \mid \mathtt{x}_2 \bullet \mathtt{A}_2) \end{array}\right) \end{array}\right) \xrightarrow{\mathtt{d?w_2}} \left(\begin{array}{l} \mathtt{c}_3 \land \mathtt{c}_4 \land \mathtt{w}_1 = \mathtt{w}_2 \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c} (\mathtt{par}\ \mathtt{s}_3 \mid \mathtt{x}_1 \bullet \mathtt{A}_3) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{par}\ \mathtt{s}_4 \mid \mathtt{x}_2 \bullet \mathtt{A}_4) \end{array}\right) \end{array}\right)} \tag{A.25}$$

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \quad \mathtt{l} \neq \epsilon \quad \mathrm{chan}\,\mathtt{l} \notin cs}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \setminus \mathtt{cs}) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2 \setminus \mathtt{cs})} \tag{A.26}$$

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \xrightarrow{\mathtt{l}} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \quad \mathtt{l} = \epsilon \lor \mathrm{chan}\,\mathtt{l} \in cs}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \setminus \mathtt{cs}) \xrightarrow{\epsilon} (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2 \setminus \mathtt{cs})} \tag{A.27}$$

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{Skip} \setminus \mathtt{cs}) \xrightarrow{\epsilon} (\mathtt{c} \mid \mathtt{s} \models \mathtt{Skip})} \tag{A.28}$$

1160

B Specification-oriented transition system: labels and rules

The syntax of the labels of specification traces is given below, followed by the transition rules.

$$
\begin{array}{lll}
\mathit{Label} & ::= & \mathit{Pred} \mid \mathit{Comm} \mid \mathit{LAct} \\
\mathit{Comm} & ::= & \epsilon \mid \mathit{CName} \mid \mathit{CName}!\mathit{Exp} \mid \mathit{CName}?\mathit{VName} \\
 & & \mid \mathit{CName}?\mathit{VName} : \mathit{Pred} \\
\mathit{LAct} & ::= & \mathit{VName}^* : [\mathit{Pred}, \mathit{Pred}] \mid \mathit{Schema} \mid \mathit{VName} := \mathit{Exp} \\
 & & \mid \mathtt{var}\ \mathit{VName} : \mathit{Exp} \mid \mathtt{var}\ \mathit{VName} := \mathit{Exp} \mid \mathtt{end}\ \mathit{VName}
\end{array}
$$

$$\frac{(\mathit{state}(\mathtt{P}_1) \models \mathit{maction}(\mathtt{P}_1)) \stackrel{\mathtt{l}}{\Longrightarrow} (\mathit{state}(\mathtt{P}_2) \models \mathit{maction}(\mathtt{P}_2))}{\mathtt{P}_1 \stackrel{\mathtt{l}}{\Longrightarrow} \mathtt{P}_2} \quad \text{(B.1)}$$

1165

$$\frac{c \wedge (s;\ p) \wedge (\exists\, v' \bullet s;\ Q)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{p} \vdash \mathtt{Q}) \stackrel{\mathtt{p} \vdash \mathtt{Q}}{\Longrightarrow}_P (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{Q}\,[\mathtt{w}_0/\mathtt{v}']) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w}_0 \models \mathtt{Skip})} \quad v' = out\alpha s \quad \text{(B.2)}$$

$$\frac{c \wedge (s;\ \mathbf{pre}\ Op)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{Op}) \stackrel{\mathtt{Op}}{\Longrightarrow} (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{Op}\,[\mathtt{w}_0/\mathtt{v}']) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w}_0 \models \mathtt{Skip})} \quad v' = out\alpha s \quad \text{(B.3)}$$

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{v} := \mathtt{e}) \stackrel{\mathtt{v:=e}}{\Longrightarrow}_P (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{w}_0 = \mathtt{e}) \mid \mathtt{s};\ \mathtt{v} := \mathtt{w}_0 \models \mathtt{Skip})} \quad \text{(B.4)}$$

$$\frac{c \wedge (s;\ g)}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{g} \mathbin{\&} \mathtt{A}) \stackrel{\mathtt{g}}{\Longrightarrow}_P (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{g}) \mid \mathtt{s} \models \mathtt{A})} \quad \text{(B.5)}$$

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d!e} \rightarrow \mathtt{A}) \stackrel{\mathtt{d!e}}{\Longrightarrow}_P (\mathtt{c} \wedge (\mathtt{s};\ \mathtt{w}_0 = \mathtt{e}) \mid \mathtt{s} \models \mathtt{A})} \quad \text{(B.6)}$$

1170

$$\frac{c \wedge T \neq \varnothing \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{d?x} : \mathtt{T} \rightarrow \mathtt{A}) \stackrel{\mathtt{d?x}}{\Longrightarrow} (\mathtt{c} \wedge \mathtt{w}_0 \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\,x} := \mathtt{w}_0 \models \mathtt{let\ x} \bullet \mathtt{A})} \quad \text{(B.7)}$$

$$\frac{c \wedge T \neq \varnothing \qquad x \notin \alpha s}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{var\,x : T \bullet A}) \overset{(\mathtt{var\,x:T})}{\Longrightarrow}_P (\mathtt{c} \wedge \mathtt{w_0} \in \mathtt{T} \mid \mathtt{s};\ \mathtt{var\,x := w_0} \models \mathtt{let\ x \bullet A})} \tag{B.8}$$

$$\frac{c}{(\mathtt{c} \mid \mathtt{s} \models \mathtt{let\ x \bullet Skip}) \overset{(\mathtt{end\,x})}{\Longrightarrow}_P (\mathtt{c} \mid \mathtt{s};\ \mathtt{end\,x} \models \mathtt{Skip})} \tag{B.9}$$

$$\frac{(\mathtt{c_1} \mid \mathtt{s_1} \models \mathtt{A_1}) \overset{\mathtt{l}}{\Longrightarrow}_P (\mathtt{c_2} \mid \mathtt{s_2} \models \mathtt{A_2})}{(\mathtt{c_1} \mid \mathtt{s_1} \models \mathtt{let\ x \bullet A_1}) \overset{\mathtt{l}}{\Longrightarrow}_P (\mathtt{c_2} \mid \mathtt{s_2} \models \mathtt{let\ x \bullet A_2})} \tag{B.10}$$

$$\frac{(\mathtt{c_1} \mid \mathtt{s_1} \models \mathtt{A_1}) \overset{\mathtt{l}}{\Longrightarrow}_P (\mathtt{c_2} \mid \mathtt{s_2} \models \mathtt{A_2})}{(\mathtt{c_1} \mid \mathtt{s_1} \models \mathtt{A_1};\ \mathtt{B}) \overset{\mathtt{l}}{\Longrightarrow}_P (\mathtt{c_2} \mid \mathtt{s_2} \models \mathtt{A_2};\ \mathtt{B})} \tag{B.11}$$

1175

$$\frac{c}{\begin{array}{l} (\mathtt{c} \mid \mathtt{s} \models \mathtt{A_1} [\![\mathtt{x1} \mid \mathtt{cs} \mid \mathtt{x2}]\!] \mathtt{A_2}) \\ \overset{\mathtt{var\,v_l,v_r:=v,v}}{\Longrightarrow}_P \\ \left(\begin{array}{l} \mathtt{c} \mid \mathtt{s};\ \mathtt{var\,v_l, v_r := v, v} \\ \models \\ \left(\begin{array}{c} (\mathtt{spar\ v} \mid \mathtt{v_l} \mid \mathtt{v_r} \mid \mathtt{x1 := x1_l \bullet A_1[v_l/v]}) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{spar\ v} \mid \mathtt{v_r} \mid \mathtt{v_l} \mid \mathtt{x2 := x2_r \bullet A_2[v_r/v]}) \end{array} \right) \end{array} \right) \end{array}} \quad \begin{array}{l} v' = out\alpha s \\ v = x1, x2 \\ fresh\ v_l, v_r \end{array} \tag{B.12}$$

$$\frac{(\mathtt{c} \mid \mathtt{s};\ \mathtt{end\,v, y} \models \mathtt{A_1}) \overset{\mathtt{l}}{\Longrightarrow}_P (\mathtt{c_3} \mid \mathtt{s_3} \models \mathtt{A_3}) \quad \mathrm{chan}\,\mathtt{l} = \epsilon \vee \mathrm{chan}\,\mathtt{l} \notin cs}{\begin{array}{l} (\mathtt{c} \mid \mathtt{s} \models (\mathtt{spar\ v} \mid \mathtt{x} \mid \mathtt{y} \mid \mathtt{x_1 := z_1 \bullet A_1}) [\![\mathtt{cs}]\!] (\mathtt{spar\ v} \mid \mathtt{y} \mid \mathtt{x} \mid \mathtt{x_2 := z_2 \bullet A_2})) \\ \overset{\mathtt{l}}{\Longrightarrow}_P \\ \left(\begin{array}{l} \mathtt{c_3} \mid \mathtt{s_3} \wedge \mathtt{s};\ \mathtt{end\,x} \\ \models \\ \left(\begin{array}{c} (\mathtt{spar\ v} \mid \mathtt{x} \upharpoonright (end\,\mathtt{l}), (var\,\mathtt{l}) \mid \mathtt{y} \mid \mathtt{x_1 := z_1 \bullet A_3}) \\ [\![\mathtt{cs}]\!] \\ (\mathtt{spar\ v} \mid \mathtt{y} \mid \mathtt{x} \upharpoonright (end\,\mathtt{l}), (var\,\mathtt{l}) \mid \mathtt{x_2 := z_2 \bullet A_2}) \end{array} \right) \end{array} \right) \end{array}} \tag{B.13}$$

$$\frac{\begin{array}{c}(\mathtt{c} \mid \mathtt{s};\ \mathbf{end}\,\mathtt{v},\mathtt{y} \models \mathtt{A}_1) \overset{\mathtt{d?a}}{\Longrightarrow}_P (\mathtt{c}_3 \mid \mathtt{s}_3 \models \mathtt{A}_3) \\ (\mathtt{c} \mid \mathtt{s};\ \mathbf{end}\,\mathtt{v},\mathtt{x} \models \mathtt{A}_2) \overset{\mathtt{d!e}}{\Longrightarrow}_P (\mathtt{c}_4 \mid \mathtt{s}_4 \models \mathtt{A}_4) \\ \mathtt{d} \in cs \quad c_3 \wedge c_4 \wedge \exists\, w_0 \bullet (s_3;\ (w_0 = a)) \Leftrightarrow (s_4;\ (w_0 = e))\end{array}}{\begin{array}{l}(\mathtt{c} \mid \mathtt{s} \models (\mathbf{spar}\ \mathtt{v} \mid \mathtt{x} \mid \mathtt{y} \mid \mathtt{x}_1 := \mathtt{z}_1 \bullet \mathtt{A}_1) \llbracket\, \mathtt{cs} \,\rrbracket (\mathbf{spar}\ \mathtt{v} \mid \mathtt{y} \mid \mathtt{x} \mid \mathtt{x}_2 := \mathtt{z}_2 \bullet \mathtt{A}_2)) \\ \overset{\mathtt{d!e}}{\Longrightarrow}_P \\ \left(\begin{array}{l}\mathtt{c}_3 \wedge \mathtt{c}_4 \wedge \exists\, \mathtt{w}_0 \bullet (\mathtt{s}_3;\ (\mathtt{w}_0 = \mathtt{a})) \Leftrightarrow (\mathtt{s}_4;\ (\mathtt{w}_0 = \mathtt{e})) \mid \mathtt{s} \\ \models \\ \left(\begin{array}{c}(\mathbf{spar}\ \mathtt{v} \mid \mathtt{x} \upharpoonright \mathtt{a} \mid \mathtt{y} \mid \mathtt{x}_1 := \mathtt{z}_1 \bullet \mathtt{A}_3) \\ \llbracket \mathtt{cs} \mid \mathbf{var}\,\mathtt{a} := \mathtt{e} \mid \mathtt{s}_3 \wedge \mathtt{s}_4 \wedge \mathtt{s};\ \mathbf{end}\,\mathtt{x},\mathtt{y} \rrbracket \\ (\mathbf{spar}\ \mathtt{v} \mid \mathtt{y} \mid \mathtt{x} \upharpoonright \mathtt{a} \mid \mathtt{x}_2 := \mathtt{z}_1 \bullet \mathtt{A}_4)\end{array}\right)\end{array}\right)\end{array}} \tag{B.14}$$

The above rule uses a new parallel construct that keeps track of the new input variable declared and the new state obtained as a consequence. It is used to ensure that, as required here, all transitions have a single label, and the label contains a guard, a communication, or an action. The next rule ensures that in the next step of the evaluation of the parallelism, the variable declaration and state change are recorded. This concern was not present in [CG10].

$$\frac{c}{\begin{array}{l}\left(\mathtt{c} \mid \mathtt{s} \models \left(\begin{array}{c}(\mathbf{spar}\ \mathtt{v} \mid \mathtt{x} \mid \mathtt{y} \mid \mathtt{x}_1 := \mathtt{z}_1 \bullet \mathtt{A}_1) \\ \llbracket \mathtt{cs} \mid \mathbf{var}\,\mathtt{a} := \mathtt{e} \mid \mathtt{s}_1 \rrbracket \\ (\mathbf{spar}\ \mathtt{v} \mid \mathtt{y} \mid \mathtt{x} \mid \mathtt{x}_2 := \mathtt{z}_2 \bullet \mathtt{A}_2)\end{array}\right)\right) \\ \overset{\mathbf{var}\,\mathtt{a}:=\mathtt{e}}{\Longrightarrow}_P \\ (\mathtt{c} \mid \mathtt{s}_1 \models (\mathbf{spar}\ \mathtt{v} \mid \mathtt{x} \mid \mathtt{y} \mid \mathtt{x}_1 := \mathtt{z}_1 \bullet \mathtt{A}_1) \llbracket\, \mathtt{cs} \,\rrbracket (\mathbf{spar}\ \mathtt{v} \mid \mathtt{y} \mid \mathtt{x} \mid \mathtt{x}_2 := \mathtt{z}_2 \bullet \mathtt{A}_2))\end{array}} \tag{B.15}$$

Rules similar to those above for parallelism are needed for external choice.

$$\frac{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1) \overset{\mathtt{l}}{\Longrightarrow}_P (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2) \quad \text{chan}\ \mathtt{l} \notin cs}{(\mathtt{c}_1 \mid \mathtt{s}_1 \models \mathtt{A}_1 \setminus \mathtt{cs}) \overset{\mathtt{l}}{\Longrightarrow}_P (\mathtt{c}_2 \mid \mathtt{s}_2 \models \mathtt{A}_2 \setminus \mathtt{cs})} \tag{B.16}$$

Above, we assume that, if $\mathtt{l}$ is not a communication, then chan $\mathtt{l}$ is some special channel ϵ that does not belong to any synchronisation set $\mathtt{cs}$.