Software Tools for Technology Transfer manuscript No.
(will be inserted by the editor)

Model Checking of State-rich Formalism *Circus* by Linking to *CSP* || *B*

Kangfeng Ye · **Jim Woodcock**

Received: March 3, 2015 / Revised version: September 18, 2015

Abstract Since state-rich formalism *Circus* is a combination of Z, CSP, refinement calculus and Dijkstra's guarded commands, its model checking is intrinsically more complicated and difficult than that of individual state-based languages or process algebras. Current solutions translate executable constructs of *Circus* programs to Java with JCSP, or translate them to CSP processes. Data aspects of *Circus* programs are expressed in the Java programming language or as CSP processes. Both of them have disadvantages. This work presents a new approach to model-checking *Circus* by linking it to *CSP* || *B*, then we utilise ProB to model-check and animate the *CSP* || *B* program. The most significant advantage of this approach is the direct mapping of the state part in *Circus* to Z and finally to B, which maintains the high-level abstraction of data specification. In addition, introduction of deadlock, invariant violation checking, LTL formula checking and animation is another key advantage. We present our approach, a link definition for a subset of *Circus* constructs, as well as a popular case study (reactive buffer) to show the practical usability of our work. We conclude with a discussion of related work, advantages and potential limitations of our approach, and future work.

Keywords *Circus* · *CSP* || *B* · CSP · Z · B · ProB · Model Checking · Buffer

K. Ye (✉) · J. Woodcock
Department of Computer Science, University of York, York, United Kingdom
E-mail: ky582@york.ac.uk

J. Woodcock
E-mail: jim.woodcock@york.ac.uk

1 Introduction

In the last two decades, the advent of model checking [14], a technique used for the verification of a finite-state system by automatically and exhaustively checking whether the model meets a given specification, has been getting ever increasing interest from both industry and academia. Verification of systems specified using formal methods by model checking is one among these. Comparing to theorem proving, another technique for formal verification, the advantages of model checking, including an automated checking procedure, counterexamples for debugging, and the capability of temporal logic properties checking, make it very important for a formalism to support both model checking and theorem proving in a complementary way.

Traditional research in formal methods often focuses on two schools: state-based, model-oriented specification languages such as Z [47], B [5] and VDM [26]; and behaviour-oriented process algebras such as CSP [25, 43], CCS [32] and ACP [8]. But in recent decades, there is an increasing research interest in specification languages that integrate both state and behavioural aspects. Early solutions aim to combine them together, such as CSP-OZ [18], ZCCS [23], Z and CSP [34, 44], and *CSP* || *B* [46]. Fischer gave a summary of combination solutions of Z and process algebras [19]. However, state-rich formalism *Circus* [49] is not a simple combination of Z and CSP. It is a combination of Z, CSP, refinement calculus [33] and Dijkstra's guarded commands [16]. Therefore, its model checking is intrinsically more complicated and difficult than that of individual Z and CSP. The complexity of model checking *Circus* is increased due to two main factors. The first one is state space explosion challenge. Basically, the state of a system specified by *Circus* is the state of its processes. However, for each process it may contain state and behaviour, and consequently its state is a combination of both variable state and action

state. In addition, the process's variable and action states are dynamically constructed and destroyed along with invocation and destroy of the process. It possibly has infinite number of distinct states as well. Because of this hierarchical structure of *Circus* and possible infinite states, how to represent and search its state space including infinite state space and infinite data type efficiently is really a challenge. Another factor is *Circus*'s very rich notations from Z, CSP and guarded commands. Along with its powerful expressiveness and high-level abstraction, all make the development of its model checker—to parse and typecheck *Circus* programs, check deadlock and livelock, check refinement in terms of *Circus* action refinement and data refinement, and CSP failure-divergence refinement—difficult. In addition, the relationship between state and behaviour in early solutions is always orthogonal during development. Therefore, they can use the existing tools for each school separately. However, for *Circus* its syntax is a free mixture of CSP and Z. As a result it cannot use current Z and CSP tools directly.

Our proposed approach to model-check *Circus* in this paper is to link *Circus* to $CSP \parallel B$ that integrates state and behaviour through synchronization of operations. On the one hand, the state part in *Circus* is transformed to the B machine. On the other hand, the behavioural part is converted to CSP. The resultant CSP and B specifications maintain high-level abstraction of the initial *Circus* specification because they are specification languages rather than programming languages. Accordingly, it is more direct and powerful to specify the state part in B than in CSP. Furthermore, the final $CSP \parallel B$ can be model-checked by ProB [3], a model checker and animator for multiple languages. Apart from these, with ProB we introduce LTL and CTL [14] property checking, automatic and manual animations, and refinement checking into *Circus* specification.

Our contribution is to define a formal link from *Circus* to $CSP \parallel B$. With this link, we take the model checking into *Circus* by model-checking $CSP \parallel B$ using ProB. Additionally, to establish the link between them, we studied the transformation from Z in ISO Standard [4] dialect to Z in ZRM [47] and finally to B, and the transformation of *Circus* expressions from ISO Standard Z to CSP as well. The soundness of the link from *Circus* to the intermediate CSP and Z in ZRM is proved based on their semantics in Hoare and He's Unifying Theories of Programming (UTP) [24]. We also studied the correctness of the link from Z in ZRM to B though it relies on the implementation of ProB. Furthermore, because of the expressiveness and high-level abstraction of *Circus*, it is impossible to define an inverse link from $CSP \parallel B$ to *Circus* and achieve exactly the same *Circus* constructs as the original constructs. Hence, our link is one direction only from *Circus* to $CSP \parallel B$ and can not form Galois connections [24].

The rest of the paper is structured as follows. We give a brief introduction of *Circus* and $CSP \parallel B$ in Sect. 2. Then in Sect. 3, the link function, its decomposition and overall strategies are described. Afterwards, Sect. 4 presents a subset of rules for each link function. A case study of a buffer is undertaken in Sect. 5 to illustrate how our approach works. Finally, in Sect. 6, we discuss related work, our work's pros and cons, and future work.

2 Background

2.1 *Circus*

The BNF syntax of *Circus* is shown in Figure 1. A *Circus* program consists of a sequence of paragraphs: Z paragraph, channel declaration, channel set definition or a process definition. A category, that is decorated with an additional star, such as **CircusPar***, denotes a possibly empty list of **CircusPar**, while a category decorated with an additional plus, such as $\mathbf{N}^+$, denotes a non-empty list of Z identifiers **N**. **Par**, **SchemaExp**, **Exp**, **Pred** and **Decl** represent Z paragraphs, schema expressions, expressions, predicates and declarations defined in the reference manual [47] respectively.

A *Circus* paragraph can be a Z paragraph, a channel declaration, a channel set declaration, or a process declaration. A channel declaration is very similar to that in CSP except a schema channel declaration which merely groups the channel declarations into the schema. And a channel set declaration relates a name to a set of channels. Additionally a process can be defined in the process declaration as a parametrised process ($Decl \bullet ProcDef$), an indexed process , an explicitly defined process (**begin** $\cdots$ **state** $\cdots \bullet A$ **end**), or a compound process which is defined in terms of CSP operators or the indexed operator ($\lfloor i \rfloor$). In particular, the indexed process ($IP \mathrel{\widehat=} i : T \odot P$) is new to *Circus* . For its instantiation $IP\lfloor e \rfloor$, it behaves like P, but for each channel c in P, it is changed to $c\_i.e$. In addition, a process renaming operator ($P[c_{old} := c_{new}]$) substitutes the channels in c_{new} for the channels in c_{old} in the process P.

An explicitly defined process is composed of a state schema, which declares a set of state components in the process, multiple schemas and action declarations (possibly none of them), a main action, which defines the behaviour of this process. An action can be a schema expression ($(\!|SchExp|\!)$), a command, an action defined in terms of CSP. And a command can be an assignment (:=), an alternation (**if** $\cdots$ **fi**), a guarded command ($(\!|g|\!) \,\&\, A$), a variable block (**var** $Decl \bullet A$), a parametrisation by value (**val**), by result (**res**) and by value-result (**vres**), a specification statement ($:[\,]$), an assumption ($\{\}$), or a coercion ($[\,]$). Particularly, for CSP actions, the parallel composition ($A_1 \,[\![\, ns_1 \mid cs \mid ns_2 \,]\!]\, A_2$ where ns_1 and ns_2 are the state partitions of the actions A_1 and A_2

$$
\begin{array}{ll}
\text{Program} & ::= \text{CircusPar}^* \\
\text{CircusPar} & ::= \text{Par} \mid \textbf{channel}\ \text{CDecl} \mid \textbf{channset}\ \text{N} == \text{CSExp} \mid \text{ProcDecl} \\
\text{CDecl} & ::= \text{SimpleCDecl} \mid \text{SimpleCDecl};\text{CDecl} \\
\text{SimpleCDecl} & ::= \text{N}^+ \mid \text{N}^+ : \text{Exp} \mid [\text{N}^+]\text{N}^+ : \text{Exp} \mid \text{SchemaExp} \\
\text{ProcDecl} & ::= \textbf{process}\ \text{N} \mathrel{\widehat=} \text{ProcDef} \mid \textbf{process}\ [\text{N}^+]\text{N}^+ \mathrel{\widehat=} \text{ProcDef} \\
\text{ProcDef} & ::= \text{Decl} \bullet \text{ProcDef} \mid \text{Decl} \odot \text{ProcDef} \mid \text{Proc} \\
\text{Proc} & ::= \textbf{begin}\ \text{PPar}^*\ \textbf{state}\ \text{N} \mathrel{\widehat=} \text{SchemaExp}\ \text{PPar}^* \bullet \text{Action}\ \textbf{end} \\
 & \quad \mid \text{Proc} \fatsemi \text{Proc} \mid \text{Proc} \Box \text{Proc} \mid \text{Proc} \sqcap \text{Proc} \mid \text{Proc} \llbracket \text{CSExp} \rrbracket \text{Proc} \\
 & \quad \mid \text{Proc} \mid\mid\mid \text{Proc} \mid \text{Proc} \setminus \text{CSExp} \mid (\text{Decl} \bullet \text{ProcDef})(\text{Exp}^+) \mid \text{N}(\text{Exp}^+) \mid \text{N} \\
 & \quad \mid (\text{Decl} \odot \text{ProcDef})\lfloor \text{Exp}^+ \rfloor \mid \text{N}\lfloor \text{Exp}^+ \rfloor \mid \text{Proc}[\text{N}^+ := \text{N}^+] \mid \text{N}[\text{Exp}^+] \\
 & \quad \mid \fatsemi \text{Decl} \bullet \text{Proc} \mid \Box\, \text{Decl} \bullet \text{Proc} \mid \sqcap \text{Decl} \bullet \text{Proc} \\
 & \quad \mid \llbracket \text{CSExp} \rrbracket \text{Decl} \bullet \text{Proc} \mid \Big|\Big|\Big| \text{Decl} \bullet \text{Proc} \\
\text{PPar} & ::= \text{Par} \mid \text{N} \mathrel{\widehat=} \text{ParAction} \mid \textbf{nameset}\ \text{N} == \text{NSExp} \\
\text{ParAction} & ::= \text{Action} \mid \text{Decl} \bullet \text{ParAction} \\
\text{Action} & ::= \big(\text{SchemaExp}\big) \mid \text{Command} \mid \text{N} \mid \text{CSPAction} \mid \text{Action}[\text{N}^+ := \text{Exp}^+] \\
\text{CSPAction} & ::= \mathit{Skip} \mid \mathit{Stop} \mid \mathit{Chaos} \mid \text{Comm} \rightarrow \text{Action} \mid \text{Pred}\ \ \& \ \text{Action} \\
 & \quad \mid \text{Action} \fatsemi \text{Action} \mid \text{Action} \Box \text{Action} \mid \text{Action} \sqcap \text{Action} \\
 & \quad \mid \text{Action} \llbracket \text{NSExp} \mid \text{CSExp} \mid \text{NSExp} \rrbracket \text{Action} \\
 & \quad \mid \text{Action} \lVert [\text{NSExp} \mid \text{NSExp}] \rVert \text{Action} \\
 & \quad \mid \text{Action} \setminus \text{CSExp} \mid \text{ParAction}(\text{Exp}^+) \mid \mu\ \text{N} \bullet \text{Action} \\
 & \quad \mid \fatsemi \text{Decl} \bullet \text{Action} \mid \Box\, \text{Decl} \bullet \text{Action} \mid \sqcap \text{Decl} \bullet \text{Action} \\
 & \quad \mid \llbracket \text{CSExp} \rrbracket \text{Decl} \bullet \llbracket \text{NSExp} \rrbracket \text{Action} \mid \Big|\Big|\Big| \text{Decl} \bullet \lVert [\text{NSExp}] \rVert \text{Action} \\
\text{Comm} & ::= \text{N}\ \text{CParameter}^* \mid \text{N}\ [\text{Exp}^+]\ \text{CParameter}^* \\
\text{CParameter} & ::= ?\text{N} \mid ?\text{N:Pred} \mid !\text{Exp} \mid .\text{Exp} \\
\text{Command} & ::= \text{N}^+ := \text{Exp}^+ \mid \textbf{if}\ \text{GActions}\ \textbf{fi} \mid \textbf{var}\ \text{Decl} \bullet \text{Action} \\
 & \quad \mid \text{N}^+ : [\text{Pred}, \text{Pred}] \mid \{\text{Pred}\} \mid [\text{Pred}] \\
 & \quad \mid \textbf{val}\ \text{Decl} \bullet \text{Action} \mid \textbf{res}\ \text{Decl} \bullet \text{Action} \mid \textbf{vres}\ \text{Decl} \bullet \text{Action} \\
\text{GActions} & ::= \text{Pred} \rightarrow \text{Action} \mid \text{Pred} \rightarrow \text{Action} \Box \text{GActions}
\end{array}
$$

Fig. 1: Syntax of Circus

separately) and the interleaving ($A_1 \lVert [ns_1 \mid ns_2] \rVert A_2$) in *Circus* are slightly different. Both A_1 and A_2 have a copy of all variables in scope and may change the value of these variables. But only the changes made to the variables in ns_1 and ns_2 have an effect in the final state of the parallel composition and the interleaving. Furthermore, state components and local variables in an action can be renamed by a renaming operator ($A[v_{old} := v_{new}]$).

A simple buffer [12] specification in *Circus* is illustrated in Figure 2. The size of the buffer is bounded by *maxbuff*, a global constant declared in the axiomatic paragraph. Afterwards, two typed channenls *input* and *output* are declared to allow only natural number on their communications. Then an explicitly defined process named *Buffer* is defined. This process has two state variables *buff* and *size*. And the initial state of the process is an empty buffer where *buff* is an empty sequence and *size* is equal to 0. In addition to the state schema and the initial schema, there are two schemas *InputCmd* and *OutputCmd* defined as well. They are invoked by their corresponding schema expressions $\big(\mathit{InputCmd}\big)$ and $\big(\mathit{OutputCmd}\big)$ in the actions *Input* and *Output* respectively. The behaviour of *Buffer* is specified by its main action: it is initialized to the initial state; after that, it provides *input* (in case the buffer is not full) and *output* (in case the buffer is

$$
\begin{array}{l}
\textbf{section}\ \mathit{BufferSpec}\ \textbf{parents}\ \mathit{circus\_toolkit} \\
\mid \quad \mathit{maxbuff} : \mathbb{N}_1 \\
\textbf{channel}\ \mathit{input}, \mathit{output} : \mathbb{N} \\
\textbf{process}\ \mathit{Buffer} \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ \mathit{State} == [\mathit{buff} : \mathrm{seq}\,\mathbb{N}; \mathit{size} : 0 \mathinner{..} \mathit{maxbuff} \mid \\
\qquad\qquad \mathit{size} = \#\mathit{buff} \leq \mathit{maxbuff}\,] \\
\quad \mathit{Init} == [(\mathit{State})' \mid \mathit{buff}' = \langle\rangle \land \mathit{size}' = 0] \\
\quad \mathit{InputCmd} == [\Delta \mathit{State}; x? : \mathbb{N} \mid \mathit{size} < \mathit{maxbuff} \land \\
\qquad\qquad \mathit{buff}' = \mathit{buff} \frown \langle x? \rangle \land \mathit{size}' = \mathit{size} + 1] \\
\quad \mathit{OutputCmd} == [\Delta \mathit{State} \mid \mathit{size} > 0 \land \mathit{buff}' = \mathit{tail}\,\mathit{buff} \land \\
\qquad\qquad \mathit{size}' = \mathit{size} - 1] \\
\quad \mathit{Input} \mathrel{\widehat=} \big(\mathit{size} < \mathit{maxbuff}\big)\ \&\ \mathit{input}?x \rightarrow \big(\mathit{InputCmd}\big) \\
\quad \mathit{Output} \mathrel{\widehat=} \big(\mathit{size} > 0\big)\ \&\ \mathit{output}!(\mathit{head}\,\mathit{buff}) \rightarrow \big(\mathit{OutputCmd}\big) \\
\quad \bullet \big(\mathit{Init}\big) \fatsemi (\mu X \bullet (\mathit{Input} \Box \mathit{Output}) \fatsemi X) \\
\textbf{end}
\end{array}
$$

Fig. 2: The Specification of Buffer

not empty) events to its environment continuously. Accordingly, it buffers the input message in its end and increases its size by one, or outputs its head and decreases its size by one.

Additionally, since *Circus* is a combination of several different languages that have different semantics, there arises an issue about how to unify its semantics into one. For

example, CSP-OZ introduces the failures-divergences model [43] into Object-Z [11] classes and then integrates CSP processes with Object-Z classes based on the same failures-divergences semantics. But $CSP \parallel B$ treats a B machine as a CSP process, and gives CSP traces, stable failures and failures-divergences semantics to B machines [46]. *Circus* needs to combine not only Z and CSP, but also the guarded command language and the refinement calculus. *Circus* formalises its model in UTP because UTP is a common framework for the unification of programs from different paradigms. Denotational semantics [37, 38, 49] and operational semantics [22, 51, 52] of *Circus* have been given.

2.2 Unifying Theories of Programming

UTP is a unified framework to form theoretical basis for describing and specifying computer languages across different paradigms such as imperative, functional, declarative, nondeterministic, concurrent, reactive and high-order. A theory in UTP is described from three parts: *alphabet*, a set of variable names for the theory to be studied; *signature*, rules of primitive statements of the theory and how to combine them together to get more complex program; and *healthiness conditions*, a set of mathematically provable laws or equations to characterise the theory.

The alphabetised relational calculus [13] is the most basic theory in UTP. A relation is defined as a predicate with undecorated variables (v) and decorated variables (v') as its alphabet. v denotes the observation made initially and v' denotes the observation made at the intermediate or final state. The behaviour of a design is described from initial observation and final observation by relating precondition P to postcondition Q as $(P \vdash Q)$ [24, 50]. It is defined as $(P \vdash Q \mathrel{\widehat{=}} okay \wedge P \Rightarrow okay' \wedge Q)$ where $okay$ records the program has started and $okay'$ that it has terminated.

But a reactive process cannot be characterised from the final observation alone because it interacts with its environments (other programs and users). It must take intermediate states into account. Therefore, three extra variables and their dashed counterparts are introduced: tr and tr', the sequences of the events occurred; ref and ref', the sets of events that may be refused; $wait$ and $wait'$, the boolean variables that denote whether the process has terminated ($true$) or is in an intermediate state ($false$). Cavalcanti and Woodcock [13] lift the theory of reactive processes to CSP processes. "CSP processes are reactive; moreover they are $\boldsymbol{R}$-image of designs" [13, Figure 1, p.257]. The reactive processes are expressed as the reactive design $(\mathbf{R}(P \vdash Q))$.

2.3 Combination of CSP and B

$CSP \parallel B$ [9] is a combination of CSP and B aiming to introduce behavioural specification into state-based B machines. The B method characterises abstract state, operations with respect to their enabling conditions, and their effect on the abstract state, while CSP specifies overall system behaviour. But different from *Circus*, the CSP specification and B machine in $CSP \parallel B$ are always orthogonal. They are individually complete specifications and can be checked separately.

Semantically the combination of CSP and B works in terms of CSP. The B machine, which has operations Ops $(\{Op_1, \ldots, Op_n\})$ and variables $Vars$ $(\{v_1, \ldots, v_m\})$, is regarded as a process B_{proc} (1). The operation Op_i is enabled if its precondition holds or $enabled(Op_i)$ is true. CSP may have all or part of events from the events Ops $(\{Op_1, \ldots, Op_n\})$ that have the same name as operations Ops in B. CSP and B processes are composed together by a generalised parallel composition in CSP, as shown in (2). B and CSP synchronise on events Ops in CSP and operations Ops in B. If an event Op_i in Ops is allowed by the CSP specification and at the same time the operation Op_i in B is enabled as well, then they can make progress. After that, the state $Vars$ of the B machine, which are specified in the predicate of operation Op_i, are updated. Otherwise, if Op_i is not allowed at the same time as Op_i is enabled, neither of them can make progress. In addition, for events only in CSP and not having corresponding operations in B, they engage independently. Conversely, for operations only specified in B and not in CSP, they are prevented from executing. If none of operations required by CSP specification are enabled or the CSP process is blocked, then the $CSP \parallel B$ program is deadlocked. Consequently no state can be changed and no event can be engaged.

$$B_{proc} \mathrel{\widehat{=}} \mu X \bullet \begin{pmatrix} (Op_1 \rightarrow X) \trianglelefteq enabled(Op_1) \trianglerighteq STOP \\ \Box \ \ldots \\ \Box \ (Op_n \rightarrow X) \trianglelefteq enabled(Op_n) \trianglerighteq STOP \end{pmatrix} \tag{1}$$

$$CSP \parallel B \mathrel{\widehat{=}} (CSP \underset{Ops}{\parallel} B_{proc}) \setminus \{|Ops|\} \tag{2}$$

2.4 ProB

ProB is a model checker and animator developed by the University of Düsseldorf originally for the B language. It has been extended to support a variety of formal specification languages, such as Z, CSP, Event-B [6], TLA+ [27], Promela and $CSP \parallel B$. Particularly, the dialect of Z supported is ZRM [47] and the syntax of CSP is written in CSP_M [45]. The main functions of ProB include temporal logic and refinement model checking, deadlock and invariant violation checking with counterexamples available, au-

tomatic and manual animations, visualisation of state spaces and test-case generation. Its kernel is written in SICStus Prolog [10]. Most importantly, its source code is open and licensed under EPL v1.0 [17].

3 Link Definitions

3.1 Overall Link Function

A function Υ (pronounced "upsilon") is defined to map a *Circus* program to a *CSP* ‖ *B* program.

$$Circus \xrightarrow{\Upsilon} CSP \parallel B$$

The overall strategies of Υ are defined.

- Fundamentally, the state part of *Circus* is linked to a B machine and the behavioural part to a CSP specification. Some constructs such as command actions, which specify both state and behaviour, are mapped to constructs in both B and CSP.
- The definitions, such as type definitions, abbreviation and axiomatic definitions, are mapped to the counterparts in B and possibly in CSP if they are referred to in the behavioural part of *Circus*.
- State components in *Circus* are mapped to variables in B. However, since state components are encapsulated in explicitly defined processes, they are merged to form variables in B when mapped.
- Operational schemas within an explicitly defined process, which includes the state schema in its declaration and is not included by other schemas, are mapped to operations in B. However, these operations are restricted to manipulate variables that are mapped from state components of the same processes in *Circus* , and never change variables that are mapped from state components of different processes.
- Channel declarations are mapped to channel declarations in CSP.
- The main action of an explicitly defined process is mapped to a same name process in CSP.
- Compound processes are linked to the same name processes in CSP.

3.2 Υ Function Decomposition

Because a *Circus* program is linked to a *CSP* ‖ *B* program with a complete B machine and a CSP specification, we decompose the Υ function into two functions: Ω (pronounced "omega") function and Φ (pronounced "phi") function. The Ω function is responsible for the translation of the state part in *Circus* to B, while the Φ function is for that of the behavioural part to CSP.

However, *Circus* itself is not a simple combination of the CSP and Z languages but a free mixture of CSP and Z with additional guarded commands. An exact example is the assignment command that may specify both state and behaviour. For instance, this action (3) inputs a value x over c channel, then the assignment command updates the state variable s to x plus the local variable l. In this action, state and behaviour are mixed together. As a result, Ω and Φ functions cannot apply to the original *Circus* program directly.

$$c?x \rightarrow s := x + l \tag{3}$$

Thus, another function, named R_{wrt}, is defined. It aims to rewrite a *Circus* program to separate the state and behavioural parts into Z and CSP. The action (3) is rewritten to an action and a schema (4) according to the R_{wrt} Rule 7 which is defined latter. Finally, Ω and Φ can be easily applied to this rewritten *Circus* program.

$$c?x \rightarrow \left(AssOp\right) \tag{4}$$

where $AssOp == [\Delta P\_StPar; l? : T_{l?}; x? : T_c \mid s' = x? + l?]$

The relation of Υ function decomposition is displayed in Figure 3. In a rewritten *Circus* program, state and behaviour are separate. No construct will specify both state and behaviour at the same time. The interaction between them highly depends on schema expressions. The original schemas and schema expressions in Z and behaviour respectively are still kept in the rewritten program. In addition, it is worth noting that additional operational schemas are added in Z, and any direct state components accessed and updated in *Circus* actions will rely on schema expressions. Furthermore, for other constructs such as commands, they are rewritten to additional schemas and their schema expressions as well. Finally we state that the rewritten *Circus* program has the same structure as the original program, which means state components of each process are still encapsulated in its own process.

3.2.1 Ω Function Decomposition

For the Ω function, our strategy is to reuse the currently available solution [41] in ProB to translate Z in ZRM to B. Considering this strategy, we map the state part of the *Circus* program to ISO Standard Z first because *Circus* itself is written in ISO Standard Z, then to Z in ZRM, and finally from ZRM to B by ProB. Accordingly, the Ω function is decomposed as well: the Ω_1 function translates the state part in Z in a rewritten *Circus* specification to a complete specification in ISO Standard Z by merging all state components and schemas from all processes; the Ω_2 function syntactically transforms Z in ISO Standard Z to that in ZRM; the Ω_3 function, translation function from ZRM to B, is implemented in ProB and stated in Daniel Plagge et al.'s work [41].

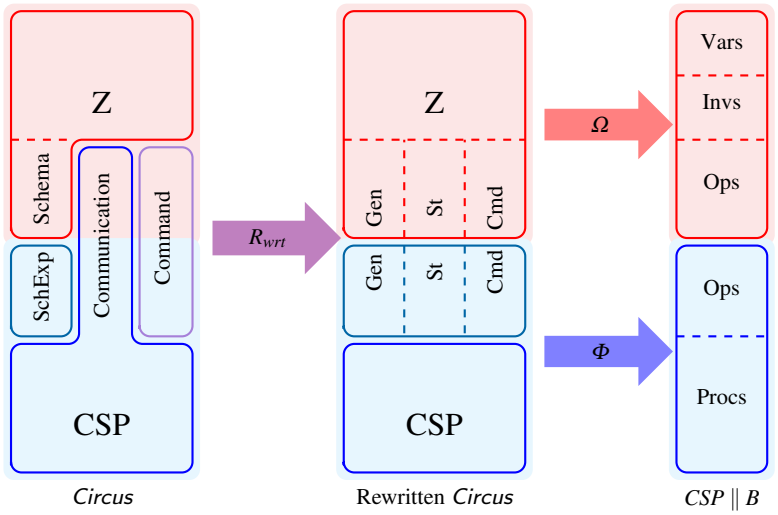

Fig. 3: Translation Function (Υ) Decomposition

3.3 Link Strategies

In addition to the overall strategies and link functions, some other strategies are defined.

- Every rule defined for Υ is sound unless stated otherwise. The soundness of the map is based on UTP semantics. If the corresponding linked constructs in *CSP* $\parallel$ *B* have the same semantics as the original constructs in *Circus*, then the link is sound.
  - From the design perspective, a design $P_1 \vdash Q_1$ is equal to another design $P_2 \vdash Q_2$ if, and only if, $(P_1 = P_2) \wedge (P_1 \Rightarrow (Q_1 = Q_2))$. If both designs have the same alphabets (ok, v, and their dashed counterparts), the same preconditions that imply the equal postcondition, we say they are semantically equal.
  - From the reactive process ($\mathbf{R}(P \vdash Q)$) perspective, if both reactive processes have the same alphabets (ok, $wait$, tr, ref, v, and their dashed counterparts), the same preconditions that imply the equal postcondition and the same other observation variables, we say two reactive processes are semantically equal.
  - For state-based specification languages such as Z and B, their semantics are specified in the designs of UTP. But for CSP and the behavioural part of *Circus*, their semantics are specified in the reactive theory of UTP.
- State components of *Circus* are maintained in a Z specification and finally a B machine. Thus we require they are updated only in the B machine but can be accessed in both B and CSP programs. The CSP specification will not maintain states. If a process in the CSP specification needs to get the value of variables in B, it shall retrieve them through a communication between CSP and B.

4 Link Rules

4.1 Identifiers

In ISO Z Standard, an identifier is a `DECORWORD` that is composed of `WORD` and `STROKE` [4, 8.4]. Stokes (`'`, `!`, and `?`) are very important part in Z specification. They may denote dashed variables, input variables and output variables within a schema. In addition, they may form the schema decoration and binding construction expressions as well. A word can be a keyword, operator or name. In addition to letter, digital and underscore (_), a name may have other special symbols such as subscript and superscript.

However, the pattern of an identifier or name in CSP_M and B, `[a-zA-Z][a-z A-Z0-9_]` [15,20], is limited. It begins with an alphabetic character (`[a-zA-Z]`) and are followed by any number of alphanumeric characters or underscores. Particularly, for CSP_M, it can be optionally followed by prime characters (`'`).

Therefore, we restrict the pattern used in *Circus* for a name the same as that in CSP_M and B. But for strokes, they are necessary and specially treated when translating to CSP_M and B.

4.2 *Circus* Rewriting Function - R_{wrt}

The R_{wrt} function is defined to rewrite *Circus* constructs to facilitate the application of the Φ function and the Ω function in the later stage.

R_{wrt} Rule 1 (Parametrised Process) *For the parametrised process, it is expanded to a number of explicitly defined processes, provided that T in (5) is finite and has n elements:* x_1,

$\ldots$, x_n. *The number of explicitly defined processes is equal to the cardinality of* T.

$$R_{wrt}\,(\textbf{process}\ PP \mathrel{\widehat=} x : T \bullet P) = \begin{pmatrix} R_{wrt}\,(\textbf{process}\ PP\_x_1 \mathrel{\widehat=} P[x_1/x]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ PP\_x_n \mathrel{\widehat=} P[x_n/x]) \end{pmatrix} \tag{5}$$

where the substitution notation $P[x_1/x]$ *denotes the expression* x_1 *consistently substituted for free occurrences of the variable* x *in* P.

R_{wrt} **Rule 2 (Indexed Process)** *An indexed process (6) is rewritten to a parametrised process with all its channels renamed at first, then it is expanded to a number of explicitly defined processes by the parametrised process rule (5).*

$$\begin{aligned} & R_{wrt}\,(\textbf{process}\ IP \mathrel{\widehat=} i : T \odot P) \\ & = R_{wrt}\,(\textbf{process}\ IP \mathrel{\widehat=} i : T \bullet P[c := c\_i.i]) \\ & = \begin{pmatrix} R_{wrt}\,(\textbf{process}\ IP\_i_1 \mathrel{\widehat=} (P[c := c\_i.i])[i_1/i]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ IP\_i_n \mathrel{\widehat=} (P[c := c\_i.i])[i_n/i]) \end{pmatrix} \\ & = \begin{pmatrix} R_{wrt}\,(\textbf{process}\ IP\_i_1 \mathrel{\widehat=} P[c := c\_i.i_1]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ IP\_i_n \mathrel{\widehat=} P[c := c\_i.i_n]) \end{pmatrix} \end{aligned} \tag{6}$$

where $P[c := c\_i.i]$ *denotes the renaming of each channel* c *in* P *to* $c\_i.i$.

R_{wrt} **Rule 3 (Renaming Operator)** *The renaming operator* $P[c_{old} := c_{new}]$ *renames the channel* c_{old} *in* P *to the channel* c_{new}.

$$R_{wrt}\,(P[c_{old} := c_{new}]) = (F_{Ren}(P, \{(c_{old}, c_{new})\})) \tag{7}$$

where $F_{Ren}(P, \{x, y\})$ *is a renaming function that replaces occurrences of the term* x *in* P *to the term* y.

R_{wrt} **Rule 4 (Indexed Process with Renaming)** *In Circus, the indexed process notation is commonly used with the renaming operator together to define more expressive processes. Therefore,*

$$\begin{aligned} & R_{wrt}\,((\textbf{process}\ IP \mathrel{\widehat=} i : T \odot P)\,[c\_i := d]) \\ & = R_{wrt}\,((\textbf{process}\ IP \mathrel{\widehat=} i : T \bullet P[c := c\_i.i])\,[c\_i := d]) \\ & \qquad [R_{wrt}\ \text{Rule 2}] \\ & = \begin{pmatrix} R_{wrt}\left(\begin{array}{l}\textbf{process}\ IP\_i_1 \mathrel{\widehat=} \\ \quad ((P[c := c\_i.i])\,[i_1/i])\,[c\_i := d]\end{array}\right) \\ \ldots \\ R_{wrt}\left(\begin{array}{l}\textbf{process}\ IP\_i_n \mathrel{\widehat=} \\ \quad ((P[c := c\_i.i])\,[i_n/i])\,[c\_i := d]\end{array}\right) \end{pmatrix} \\ & \qquad [R_{wrt}\ \text{Rule 1}] \\ & = \begin{pmatrix} R_{wrt}\left(\begin{array}{l}\textbf{process}\ IP\_i_1 \mathrel{\widehat=} \\ \quad (P[c := c\_i.i_1])\,[c\_i := d]\end{array}\right) \\ \ldots \\ R_{wrt}\left(\begin{array}{l}\textbf{process}\ IP\_i_n \mathrel{\widehat=} \\ \quad (P[c := c\_i.i_n])\,[c\_i := d]\end{array}\right) \end{pmatrix} \\ & \qquad [\text{Substitution}] \\ & = \begin{pmatrix} R_{wrt}\,(\textbf{process}\ IP\_i_1 \mathrel{\widehat=} P[c := d.i_1]) \\ \ldots \\ R_{wrt}\,(\textbf{process}\ IP\_i_n \mathrel{\widehat=} P[c := d.i_n]) \end{pmatrix} \quad [R_{wrt}\ \text{Rule 3}] \end{aligned}$$

For explicitly defined processes, the R_{wrt} function is to separate the state part and the behavioural part as well as renaming of state components, schema paragraphs and action paragraphs. Consequently, all interactions between state and behaviour are through schema expressions only.

R_{wrt} **Rule 5 (Additional State Components Retrieve Schemas)** *The rule for state components retrieve schemas is shown in Figure 4, where B function denotes the body of the action. For each state component in an explicitly defined process, one schema is added to retrieve the value of this state component. The name of the output variable in this schema is composed of the state component name and* !. *And its type is the same as the type of the state component.*

R_{wrt} **Rule 6 (Renaming of State Components, Schemas, Actions and their Refereneces)** *The rule for renaming is shown in Figure 5. State components, schema paragraphs, action paragraphs, and each reference to them within an explicitly defined process are renamed by prefixing the process's name. The only exception is that the reference to state components in action is not changed.*

R_{wrt} **Rule 7 (Action Rewriting)** *The rule for action rewriting is illustrated in Figure 6, where P_assOp (8) is a schema added in the process of this assignment. The definitions of* R_{pre} *and* R_{post} *functions are given in Definition 1. To rewrite the external choice, a* R_{mrg} *function is provided to merge the rewriting prefixes of both actions and it is defined in Definition 2. Note that it is not syntactically correct in Circus because schema expression actions cannot be a channel event in communication. But when schema expression actions are translated to events in CSP, it is valid in the final CSP* ‖ *B.*

$$\begin{aligned} P\_assOp == [\,& \Delta P\_StPar; l? : T_{l?}; l! : T_{l!} \mid \\ & P\_s' = (e_s[l?/l]) \land l! = (e_l[l?/l])\,] \end{aligned} \tag{8}$$

Definition 1 (R_{pre} **and** R_{post}) Rewriting an action to get the value of state components in its first construct, $R_{wrt}(A)$, is composed of $R_{pre}(A)$ and $R_{post}(A)$ which denotes the prefix (state components retrieve schema expressions) and the remaining respectively: $R_{wrt}(A) = R_{pre}(A) \rightarrow R_{post}(A)$. For

$$R_{wrt}\left(\begin{array}{l}\textbf{process}\ P \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state}\ StPar == [s_1 : T_1 ; \cdots s_n : T_n \mid p]\\ \quad Pars == [\cdots]\\ \quad APars \mathrel{\widehat=} B(APars)\\ \quad \bullet A\\ \textbf{end}\end{array}\right) = \left(\begin{array}{l}\textbf{process}\ P \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state}\ StPar == [s_1 : T_1 ; \cdots s_n : T_n \mid p]\\ \quad Pars == [\cdots]\\ \quad Op\_s_1 == [\Xi StPar ; s_1! : T_1 \mid s_1! = s_1]\\ \quad \cdots\\ \quad Op\_s_n == [\Xi StPar ; s_n! : T_n \mid s_n! = s_n]\\ \quad APars \mathrel{\widehat=} B(APars)\\ \quad \bullet A\\ \textbf{end}\end{array}\right)$$

Fig. 4: Additional Schemas for State Components Retrieve

example,

$$\begin{array}{l} R_{pre}(\textbf{Skip}) = \qquad R_{post}(\textbf{Skip}) = \textbf{Skip}\\ R_{pre}(\textbf{Stop}) = \qquad R_{post}(\textbf{Stop}) = \textbf{Stop}\\ R_{pre}(c!s_i!\ldots!s_j \to A) = \big(OP\_s_i\big) \to \cdots \to \big(OP\_s_j\big)\\ R_{post}(c!s_i!\ldots!s_j \to A) = c!s_i!\ldots!s_j \to R_{wrt}(A)\\ R_{pre}(g) = \big(OP\_s_i\big) \to \cdots \to \big(OP\_s_j\big)\\ R_{post}(g) = g \end{array}$$

provided the condition g evaluates state components $s_i,\ldots,s_j$, and OP_s_i is the schema name for state component s_i.

Definition 2 (R_{mrg}) A $R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2))$ function is defined to merge the rewriting prefixes of A_1 and A_2 into one final prefix. Basically, it is equal to $R_{pre}(A_1) \to R_{pre}(A_2)$ if each state component retrieve schema expression in $R_{pre}(A_2)$, $\big(OP\_s_i\big)$, is different from any in $R_{pre}(A_1)$. However, for any state component retrieve schema expression in $R_{pre}(A_2)$, if it is the same as that in $R_{pre}(A_1)$, it is removed from $R_{pre}(A_2)$ before combination. For example,

$$\begin{array}{l} R_{mrg}\Big(\big(OP\_x\big), \big(OP\_y\big)\Big) = \big(OP\_x\big) \to \big(OP\_y\big)\\ R_{mrg}\Big(\big(OP\_x\big), \big(OP\_y\big) \to \big(OP\_x\big)\Big)\\ \qquad = \big(OP\_x\big) \to \big(OP\_y\big) \end{array}$$

4.3 *Circus* state part to B - Ω

4.3.1 *Circus* state part to ISO Standard Z - Ω_1

The function Ω_1 translates the state part in a rewritten *Circus* program to a Z specification in ISO Standard Z. Because the state part of *Circus* is also written in ISO Standard Z, for most constructs they are just a direct map without changes. However, a rewritten *Circus* program still has the same structure as the original program—all state components and schemas are encapsulated within the processes—but the state and

$$R_{wrt}\left(\begin{array}{l}\textbf{process}\ P \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state}\ StPar == [s_1 : T_1 ; \cdots s_n : T_n \mid p]\\ \quad Pars == [\cdots]\\ \quad Op\_s_1 == [\Xi StPar ; s_1! : T_1 \mid s_1! = s_1]\\ \quad \cdots\\ \quad Op\_s_n == [\Xi StPar ; s_n! : T_n \mid s_n! = s_n]\\ \quad APars \mathrel{\widehat=} B(APars)\\ \quad \bullet A\\ \textbf{end}\end{array}\right) = \left(\begin{array}{l}\textbf{process}\ P \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state}\ P\_StPar == [P\_s_1 : T_1 ; \cdots P\_s_n : T_n \mid p]\\ \quad P\_Pars == [\cdots]\\ \quad P\_Op\_s_1 == [\Xi P\_StPar ; s_1! : T_1 \mid s_1! = P\_s_1]\\ \quad \cdots\\ \quad P\_Op\_s_n == [\Xi P\_StPar ; s_n! : T_n \mid s_n! = P\_s_n]\\ \quad P\_APars \mathrel{\widehat=} B(P\_APars)\\ \quad \bullet R_{wrt}(A)\\ \textbf{end}\end{array}\right)$$

Fig. 5: Renaming

$$\begin{array}{l} R_{wrt}\,(\textbf{Skip}) = \textbf{Skip}\\ R_{wrt}\,(\textbf{Stop}) = \textbf{Stop}\\ R_{wrt}\Big(\big(SExp\big)\Big) = \big(SExp\big)\\ R_{wrt}\,(c!s_i!\ldots!s_j \to A) = \big(Op\_s_i\big) \to \cdots \to \big(Op\_s_j\big) \to\\ \qquad c!s_i!\ldots!s_j \to R_{wrt}\,(A)\\ R_{wrt}\Big(\big(g\big)\,\&A\Big) = R_{pre}(g) \to R_{pre}(A) \to \Big(\big(g\big)\,\&R_{post}\,(A)\Big)\\ R_{wrt}\,(A_1 \,;A_2) = R_{wrt}\,(A_1)\,;R_{wrt}\,(A_2)\\ R_{wrt}\,(A_1 \,\square\, A_2) = R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \to\\ \qquad (R_{post}\,(A_1) \,\square\, R_{post}\,(A_2))\\ R_{wrt}\,(A_1 \sqcap A_2) = R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \to\\ \qquad (R_{wrt}\,(A_1) \sqcap R_{wrt}\,(A_2))\\ R_{wrt}\,(A_1 \,[\![ns_1 \mid cs \mid ns_2]\!]\, A_2) = R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \to\\ \qquad (R_{post}\,(A_1) \,[\![ns_1 \mid cs \mid ns_2]\!]\, R_{post}\,(A_2))\\ R_{wrt}\,(A_1 \,|\!|\!|\, ns_1 \mid ns_2 \,|\!|\!|\, A_2) = R_{mrg}\,(R_{pre}(A_1), R_{pre}(A_2)) \to\\ \qquad (R_{post}\,(A_1) \,|\!|\!|\, ns_1 \mid ns_2 \,|\!|\!|\, R_{post}\,(A_2))\\ R_{wrt}\Big(A \setminus cs\Big) = R_{pre}\,(A) \to \Big(R_{post}\,(A) \setminus cs\Big)\\ R_{wrt}\,\big(\mu X \bullet A\,(X)\big) = \mu X \bullet R_{wrt}\,(A(X)) \text{ and } (R_{wrt}\,(X) = X)\\ R_{wrt}\,(A) = R_{wrt}\,(B(A))\\ R_{wrt}\,(s, l := e_s, e_l) = \big(P\_assOp\big)\\ R_{wrt}\,(\textbf{var}\,x : T \bullet A) = R_{pre}\,(A) \to (\textbf{var}\,x : T \bullet R_{post}\,(A)) \end{array}$$

Fig. 6: Action Rewriting

schemas in a ISO Standard Z specification are flat. Therefore, we need to merge all state components and schemas into one global and flat specification in ISO Standard Z.

Ω_1 Rule 1 (States and Schemas Merge) *If there are more than one explicitly defined process, their states and operations are merged in the resultant Z specification. Assume there are n explicitly defined processes, named $P_1, P_2, \ldots, P_n$, in a Circus specification. Their states and schemas are merged as shown in Figure 7. The* state *schema is a conjunction of state schemas from all processes, as well as the* Init *schema. All other schemas from each process will be translated to corresponding schemas with their own* declaration *and* predicate. *Additionally they shall keep state components*

Ω_1(Rewritten *Circus* Program)

$$
= \Omega_1 \left(\begin{array}{l}
\left(\begin{array}{l}
\textbf{process}\ P_1 \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ P_1\_StPar == [P_1\_s_1 : T_{11}; \cdots \\
\qquad P_1\_s_{m_1} : T_{1m_1} \mid ps_1] \\
\quad P_1\_Init == [(P_1\_StPar)' \mid pi_1] \\
\quad P_1\_Pars == [decl_1 \mid p_1] \\
\quad \bullet\ A \\
\textbf{end}
\end{array} \right) \\
\cdots \\
\left(\begin{array}{l}
\textbf{process}\ P_n \mathrel{\widehat=} \textbf{begin} \\
\quad \textbf{state}\ P_n\_StPar == [P_n\_s_1 : T_{n1}; \cdots \\
\qquad P_n\_s_{m_n} : T_{nm_n} \mid ps_n] \\
\quad P_n\_Init == [(P_n\_StPar)' \mid pi_n] \\
\quad P_n\_Pars == [decl_n \mid p_n] \\
\quad \bullet\ A \\
\textbf{end}
\end{array} \right)
\end{array} \right)
$$

$$
= \left(\begin{array}{l}
P_1\_StPar == [P_1\_s_1 : T_{11}; \cdots P_1\_s_{m_1} : T_{1m_1} \mid ps_1] \\
\cdots \\
P_n\_StPar == [P_n\_s_1 : T_{n1}; \cdots P_n\_s_{m_n} : T_{nm_n} \mid ps_n] \\
State == P_1\_StPar \wedge \cdots \wedge P_n\_StPar \\
Init == [(State)' \mid pi_1 \wedge \cdots \wedge pi_n] \\
P1\_Pars == [P_1\_Pars.decl_1; \Xi P_2\_StPar; \ldots; \\
\qquad \Xi P_n\_StPar \mid P_1\_Pars.p_1] \\
\cdots \\
Pn\_Pars == [P_n\_Pars.decl_n; \Xi P_1\_StPar; \ldots; \\
\qquad \Xi P_{n-1}\_StPar \mid P_n\_Pars.p_n]
\end{array} \right)
$$

Fig. 7: Ω_1 Function

from other processes unchanged by including Ξ *of all other state paragraphs into their declaration.*

4.3.2 ISO Standard Z to ZRM - Ω_2

The function Ω_2 takes the constructs in ISO Standard Z as input and outputs the corresponding constructs in ZRM. It is only syntactical transformation. Only the transformation rules used in this paper are shown.

Ω_2 Rule 1 (Schema Decoration)

$$\Omega_2(S') = S' \qquad \Omega_2((S)') = S'$$

Ω_2 Rule 2 (Horizontal Schema) *In ISO standard Z,* $==$ *is used for horizontal schema but* $\mathrel{\widehat=}$ *in ZRM. Thus,*

$$\Omega_2(==) = \ \mathrel{\widehat=}$$

4.3.3 ZRM to B machine - Ω_3

Our Ω_3 function, which translates from Z in ZRM to B machine, uses the implementation of ProZ [41] in ProB. Since ProB is the model checking tool for *CSP* $\|$ *B* specification, our solution is to translate *Circus* to Z in ZRM and CSP specifications, then supply them to ProB. Eventually, ProB translates Z to B by ProZ and model-checks it as *CSP* $\|$ *B* specification.

Furthermore, because only a considerable subset of Z is implemented in ProB and others [41] shown below are not supported, our solution is accordingly restricted.

- *Generic* definitions cannot be supported. Therefore, genericity in *Circus* is not supported.
- Reflexive-transitive closure construct is not supported.

4.4 *Circus* behaviour to CSP and Z - Φ

The function Φ transforms the behavioural part of a *Circus* specification to CSP and possibly Z.

Φ Rule 1 (Types, Expressions and Operators) *The translation rules for only a very small number of types and expressions are shown below.*

- $\Phi(\mathbb{N}) = Nat$ *where* $Nat = \{0..MAXINT\}$ *and MAXINT is a constant declared in the beginning of CSP specification.*
- $\Phi(n..m) = \{n..m\}$.
- $\Phi(T_1 \times T_2) = \Phi(T_1).\Phi(T_2)$ *if Cartesian product is used in the channel expression.*
- $\Phi(T_1 \times T_2) = cross(\Phi(T_1), \Phi(T_2))$ *if Cartesian product is used in other places.*
- $\Phi(\text{seq}\ T) = fseq(\Phi(T))$ *because Seq function in* CSP_M *is an infinite set of finite sequence, it cannot be the type of channel in CSP of ProB. Otherwise, it results in the infinite enumeration error. Our solution is to treat* seq T *as a partial function* $\mathbb{Z} \pfun T$ *but with extra restriction of maximum number of elements in its domain. Therefore, fseq function is defined in Figure 8. MAXINS denotes the maximum number of instances for model checking and it is put in the beginning of CSP specification like MAXINT.*
  - *The cardinality of* $fseq(s)$ *is equal to*

    $$\sum_{n=0}^{MAXINS} (card(s))^n$$

    and if the size of s is 4 and MAXINS is 5, then the size of $fseq(s)$ *is 1365.*
  - *MAXINS is set by users but what is its optimum value highly depends on the programs to be checked and the computer that ProB runs on. On a powerful computer, it can be set to a higher value but still maintain reasonable model checking resources (memory, CPU and time) consumption.*
- *Abbreviation definition* $(AbbrDef == Expr)$ *is linked to* $(nametype\ AbbrDef = \Phi(Expr))$ *in CSP.*

Φ Rule 2 (Axiomatic Definition) *An axiomatic definition*

$$\begin{array}{|l} x : T \\ \hline p \end{array}$$

```
MAXINS = 3
-- Cartesian Product
cross(X, Y) = {(x,y) | x <- X, y <- Y}
-- relation
rel(X, Y) = Set(cross(X, Y))
-- partial function
pfun(X, Y) = { s | s<-rel(X,Y), empty({x1 | (x1,y1)
    <-s, (x2,y2)<-s, x1 == x2 and y1 != y2})}
squash(s) = let
    pick({x}) = x
    below(b) = card({ x | (x,y)<-s, x <= b })
    pairs = { (y, below(x)) | (x,y)<-s }
    select(i) = pick({ y | (y,n)<-pairs, i==n })
  within < select(i) | i <- <1..card(s)> >
fseq(s) = {squash(ss) | ss <- pfun({1..MAXINS}, s) }
```

Fig. 8: *fseq* function

is translated to x = c, where c is an instance from the set `{x | x<-T, p}` *and shall be assigned manually before model checking.* ***Notes****: c shall match the value of constant x in Z.*

Φ Rule 3 (Channel Declaration) *The link of the channel declaration in Circus to that in CSP is direct and straightforward.*

$\Phi(\textbf{channel}\ chn\_name) = channel\ chn\_name$

$\Phi(\textbf{channel}\ chn\_name : T) = channel\ chn\_name{:}\ \Phi(T)$

Φ Rule 4 (Channel Set)

$\Phi(\{\!|\,|\!\}) = \{||\}$

$\Phi(\{\!|\,c_1, c_2, \cdots, c_n\,|\!\}) = \{|c_1, c_2, \cdots, c_n|\}$

Φ Rule 5 (Channel Set Declaration)

$\Phi(\textbf{channelset}\ N == \{\!|\,|\!\}) = (N = \{||\})$

$\Phi(\textbf{channelset}\ N == \{\!|\,c_1, c_2, \cdots, c_n\,|\!\}) =$
$(N = \{|c_1, c_2, \cdots, c_n|\})$

4.4.1 Actions

The rules for a subset of *Circus* actions are shown below.

Φ Rule 6 (Basic Actions) *The link of the basic actions in Circus to those in CSP is direct and straightforward.*

$\Phi(\textbf{Stop}) = STOP$

$\Phi(\textbf{Skip}) = SKIP$

$\Phi(\textbf{Chaos}) = \textbf{div}$

Φ Rule 7 (Prefixing) *The link of the prefixing in Circus to that in CSP is direct and straightforward.*

$$\begin{array}{ll}
\Phi(c \rightarrow A) & = c \rightarrow \Phi(A) \\
\Phi(c.e \rightarrow A) & = c.\Phi(e) \rightarrow \Phi(A) \\
\Phi(c!e \rightarrow A) & = c!\Phi(e) \rightarrow \Phi(A) \\
\Phi(c?x \rightarrow A(x)) & = c?x \rightarrow \Phi(A(x)) \\
\Phi(c?x : p \rightarrow A(x)) & = c?x : \{y \mid y \ \texttt{<-}\ \Phi(T_c), \Phi(p)\} \rightarrow \\
 & \quad \Phi(A(x))
\end{array}$$

Φ Rule 8 (Schema Expression as Action) *A schema expression as action $\left(SExp\right)$ is linked to an external choice of the same name event SExp with input and output variables, and another event SExp_f which precondition is the negation of precondition of SExp. Therefore, if the precondition of SExp holds, it engages SExp event; otherwise, it engages SExp_f event and consequently diverges as* **div**. *Finally, these events are hidden from communication by adding both events to HIDE_CSPB. That makes it semantically equal to schema expression as action in Circus.*

$$\Phi\left(\left(SExp\right)\right) = \begin{cases} channel\ SExp{:}\ \Phi(T_i).\Phi(T_o) \\ channel\ SExp\_f{:}\ \Phi(T_i) \\ HIDE\_CSPB = \{|SExp, SExp\_f|\} \\ (SExp!ins?outs \rightarrow SKIP \ \Box\ SExp\_f!ins \rightarrow \textbf{div}) \\ SExp\_f = [\Xi StPar; ins? : T_i \mid \neg\, \textbf{pre}\, SExp] \end{cases}$$

provided *SExp is a schema in Z with input variables ins? and output variables outs!; SExp_f is an additional schema in Z; particularly, its predicate is the negation of the precondition of SExp.*

Φ Rule 9 (Simplified Schema Expression as Action) *If the precondition of SExp always holds such as state component retrieve schema expressions and assignments, the rule 8 is simplified because it is not possible to make its precondition be evaluated to false.*

$$\Phi\left(\left(SExp\right)\right) = \begin{cases} channel\ SExp{:}\ \Phi(T_i).\Phi(T_o) \\ HIDE\_CSPB = \{|SExp|\} \\ \begin{cases} (SExp!ins?outs \rightarrow SKIP)\ \ if\ \left(SExp\right)\ as\ process \\ (SExp!ins?outs)\ \ if\ \left(SExp\right)\ as\ communication \end{cases} \end{cases}$$

Φ Rule 10 (Miscellaneous Actions)

$$\begin{array}{ll}
\Phi\left(\left(g\right) \,\&\, A\right) & = \Phi(g)\ \&\ \Phi(A) \\
\Phi(A_1 ; A_2) & = \Phi(A_1) ; \Phi(A_2) \\
\Phi(A_1 \sqcap A_2) & = \Phi(A_1) \sqcap \Phi(A_2) \\
\Phi\left(A \setminus cs\right) & = \Phi(A) \setminus cs \\
\Phi(\mu X \bullet A(X)) & = let\ X = \Phi(A(x))\ within\ X
\end{array}$$

Φ Rule 11 (External Choice) *External choice of actions in Circus is only resolved by external events of the process or termination. Internal events of the process, such as schema expression as action and assignment, would not resolve it. Thus we restrict the actions that can occur in external choice construct to AA.*

$\Phi(AA_1 \ \Box\ AA_2) = \Phi(AA_1) \ \Box\ \Phi(AA_2)$

where AA can be one of actions below.

- *Basic actions:* **Skip**, **Stop**, *or* **Chaos**
- *Prefixed actions:* $c?x?\ldots!e!\cdots \rightarrow A$

– *Guarded commands:* $(g)\,\&AA$

Furthermore, provided both actions are guarded commands and their conditions (g_1 and g_2) are mutually exclusive, that is, $g_1 = \neg g_2$, then their guarded actions are not restricted.

$$\Phi\left((g_1)\,\&A_1 \,\Box\, (g_2)\,\&A_2\right)$$
$$= \Phi\left((g_1)\,\&A_1\right) \Box\, \Phi\left((g_2)\,\&A_2\right)$$

Φ Rule 12 (Iterated Operator)

$$\Phi\left(;x:T \bullet A(x)\right) = ;_{x:\Phi(T)} \bullet \Phi(A(x))$$
$$\Phi\left(\Box x:T \bullet AA(x)\right) = \Box_{x:\Phi(T)} \bullet \Phi(AA(x))$$
$$\Phi\left(\sqcap x:T \bullet A(x)\right) = \sqcap_{x:\Phi(T)} \bullet \Phi(A(x))$$

Φ Rule 13 (Parallel Composition and Interleaving) *Variables in parallel composition are partitioned to ns_1 and ns_2. Both actions can access the initial value of all variables from ns_1 and ns_2, but they can only modify variables in their own partition ns_1 and ns_2 respectively. Our solution is to declare temporary variables tpv_1 and tpv_2 which are initialized to the initial value of all variables in scope pv_1 and pv_2 for A_1 and A_2. Instead of updating pv_1 and pv_2, we update tpv_1 and tpv_2. Eventually, only variables in ns_1 and ns_2 are updated to the value of corresponding variables in tpv_1 and tpv_2, and others are discarded.*

$$\Phi(A_1 \,[\![ns_1 \mid cs \mid ns_2]\!]\, A_2) =$$
$$\left(\begin{array}{l} \Phi\left(\mathbf{var}\, tpv_1 \bullet \left(\begin{array}{l} tpv_1 := pv_1; \\ (A_1[tpv_1/pv_1]); \\ ns_1 := tpns_1 \end{array}\right)\right) \\ \Big\|_{cs} \\ \Phi\left(\mathbf{var}\, tpv_2 \bullet \left(\begin{array}{l} tpv_2 := pv_2; \\ (A_2[tpv_2/pv_2]); \\ ns_2 := tpns_2 \end{array}\right)\right) \end{array}\right)$$

$$\Phi(A_1 \,|||[ns_1 \mid ns_2]|||\, A_2) =$$
$$\left(\begin{array}{l} \Phi\left(\mathbf{var}\, tpv_1 \bullet \left(\begin{array}{l} tpv_1 := pv_1; \\ (A_1[tpv_1/pv_1]); \\ ns_1 := tpns_1 \end{array}\right)\right) \\ ||| \\ \Phi\left(\mathbf{var}\, tpv_2 \bullet \left(\begin{array}{l} tpv_2 := pv_2; \\ (A_2[tpv_2/pv_2]); \\ ns_2 := tpns_2 \end{array}\right)\right) \end{array}\right)$$

Φ Rule 14 (Parallel Composition and Interleaving (Disjoint Variables in Scope))

$$\Phi(A_1 \,[\![ns_1 \mid cs \mid ns_2]\!]\, A_2) = \Phi(A_1) \Big\|_{cs} \Phi(A_2)$$
$$\Phi(A_1 \,|||[ns_1 \mid ns_2]|||\, A_2) = \Phi(A_1) \,|||\, \Phi(A_2)$$

provided

$$ns_1 = \alpha(A_1) = scpV(A_1)$$
$$ns_2 = \alpha(A_2) = scpV(A_2)$$

where scpV is a function to get a set of all variables in scope in an action.

Φ Rule 15 (Variable Block) *A variable block is linked to replicated internal choice in CSP which declares a set of local variables x and their initial value is arbitrary chosen. We use the memory model [35] (Definition 5) in CSP to maintain local variables. The linked process in CSP is put in parallel with replicated parallel of a set of memory cell processes. For each variable in x, there is a unique memory cell (MemCell process) that is distinguished by i.*

$$\Phi(\mathbf{var}\, x:T \bullet A) = \sqcap_{x:\Phi(T)} \bullet F_{Mem}(\Phi(A), \{x\})$$

Definition 3 (*MemCell* Process) A *MemCell* process defined below is the mechanism in CSP to store the value of a local variable. For each local variable, it shall have a *MemCell* process. Therefore, the process is distinguished by number i which is a unique number for each variable. *MemCell* process is initialized by set_i at first, and after that it will continuously provide update and retrieve of the variable by set_i and get_i channels respectively. Additionally, it is capable of terminating successfully through *end* event.

$$\begin{array}{lll} MemCell_i = & & set_i?x \rightarrow MCell_i(x) \\ MCell_i(x) = & & set_i?y \rightarrow MCell_i(y) \\ & \Box & get_i!x \rightarrow MCell_i(x) \\ & \Box & end \rightarrow SKIP \end{array}$$

Definition 4 (F_{Var} function) $F_{Var}(P,v)$ function makes every access to each local variable l from the set v in CSP process P by $get_i?l$, and every update to l by $set_i!l$. For example,

$$F_{Var}(c?x!y!z \rightarrow P, \{x,y,z\}) = get_i?y \rightarrow get_j?z \rightarrow$$
$$\quad c?x!y!z \rightarrow set_k!x \rightarrow F_{Var}(P, \{x,y,z\})$$
$$F_{Var}(c?x!y!z \rightarrow P, \{y\}) = get_i?y \rightarrow c?x!y!z \rightarrow$$
$$\quad F_{Var}(P, \{y\})$$
$$F_{Var}(P, \{\}) = P$$

Definition 5 (F_{Mem}) The function F_{Mem} gives a memory model for CSP process P to store and retrieve local variables, which are shown in a set v with m elements: $l_1, \cdots, l_m$.

$$F_{Mem}(P,v) =$$
$$\left(\begin{array}{l} (set_1!l_1 \rightarrow \cdots \rightarrow set_m!l_m \rightarrow SKIP; \\ \qquad F_{Var}(P,v)\,; end \rightarrow SKIP) \\ \Big\|_{vs} \left(\Big|\Big|_{\{|end|\}} \{MemCell_i \mid i \in \{1..m\}\}\right) \end{array}\right) \setminus vs$$

where $vs = \{|set_1, get_1, set_2, get_2, \cdots, set_m, get_m, end|\}$

Φ Rule 16 (Action Renaming) *The variable v_{old} is renamed to the v_{new} by the action renaming.*

$$\Phi(A[v_{old} := v_{new}]) = \Phi(A[v_{new}/v_{old}])$$

Φ Rule 17 (Action Invocation) *An action reference is the body of the action.*

$$\Phi(A) = \Phi(B(A)) \textbf{ provided } A \mathrel{\widehat=} B(A)$$

Φ Rule 18 (Parametrised Action) *A parametrised action invocation is the body of the parametrised action with the parameters x substituted by the expressions e.*

$$\Phi(A(e)) = \Phi(B(A)[e/x]) \textbf{ provided } A \mathrel{\widehat=} x : T \bullet B(A)$$

4.4.2 Processes

Φ Rule 19 (Explicitly Defined Process) *For an explicitly defined process, its main action is linked to a CSP process. Its state schema and operational schemas are linked to Z and finally B by the Ω function in Sect. 4.3.*

$$\Phi\left(\begin{array}{l}\textbf{process } P \mathrel{\widehat=} \textbf{begin}\\ \quad \textbf{state } StPar == [decl \mid pred]\\ \quad Pars == [\cdots]\\ \quad APars \mathrel{\widehat=} B(APars)\\ \quad \bullet A\\ \textbf{end}\end{array}\right) = (P = \Phi(A))$$

Φ Rule 20 (Process Invocation) *A process invocation is the process itself.*

$$\Phi(P) = P$$

Φ Rule 21 (Compound Process) *For a compound process defined in terms of CSP operators, it is simply an operator expansion.*

$$\begin{array}{ll}\Phi(P;Q) & = \Phi(P);\Phi(Q)\\ \Phi(P \,\Box\, Q) & = \Phi(P) \,\Box\, \Phi(Q)\\ \Phi(P \sqcap Q) & = \Phi(P) \sqcap \Phi(Q)\\ \Phi(P \,[\![cs]\!]\, Q) & = \Phi(P) \parallel \Phi(Q)\\ \Phi(P \mid\mid\mid Q) & = \Phi(P) \overset{cs}{\mid\mid\mid} \Phi(Q)\\ \Phi(P \setminus cs) & = \Phi(P) \setminus cs\end{array}$$

Φ Rule 22 (Iterated Process) *For a process defined in terms of an iterated operator in CSP, it is simply an operator expansion.*

$$\begin{array}{ll}\Phi(;x : T \bullet P(x)) & = ;_{x:\Phi(T)} \bullet \Phi(P(x))\\ \Phi(\Box x : T \bullet P(x)) & = \Box_{x:\Phi(T)} \bullet \Phi(P(x))\\ \Phi(\sqcap x : T \bullet P(x)) & = \sqcap_{x:\Phi(T)} \bullet \Phi(P(x))\\ \Phi([\![CS]\!] x : T \bullet P(x)) & = \Big\|_{x:\Phi(T)}^{CS} \bullet \Phi(P(x))\\ \Phi(|||x : T \bullet P(x)) & = |||_{x:\Phi(T)} \bullet \Phi(P(x))\end{array}$$

Φ Rule 23 (Parametrised Process Invocation) *For the parametrised process invocation, it is simply linked to its corresponding explicitly defined process after rewriting.*

$$\begin{array}{lll}\Phi(PP(const)) & = & PP\_const\\ \Phi(PP(x)) & = & (x == \Phi(x_1)) \;\&\; PP\_x_1\\ & & \Box\, (x == \Phi(x_2)) \;\&\; PP\_x_2\\ & & \cdots\\ & & \Box\, (x == \Phi(x_n)) \;\&\; PP\_x_n\end{array}$$

where const denotes a constant.

Φ Rule 24 (Indexed Process Invocation) *For the indexed process invocation, it is simply linked to its corresponding explicitly defined process after rewriting.*

$$\begin{array}{lll}\Phi(IP\lfloor const \rfloor) & = & IP\_const\\ \Phi(IP\lfloor x \rfloor) & = & (x == \Phi(x_1)) \;\&\; IP\_x_1\\ & & \Box\, (x == \Phi(x_2)) \;\&\; IP\_x_2\\ & & \cdots\\ & & \Box\, (x == \Phi(x_n)) \;\&\; IP\_x_n\end{array}$$

5 Case Study: Reactive Buffer

This section shows how the specification of a buffer and its implementation, a distributed reactive buffer, from the paper [12] can be linked to *CSP* $\parallel$ *B* by the links defined. Eventually, we model-check them by ProB. Particularly, the implementation is checked to be both a trace refinement and a failure refinement of the specification.

5.1 Buffer Specification

The specification of *BufferSpec* in *Circus* is shown in Figure 2.

5.1.1 Rewriting by R_{wrt}

According to our link definitions in Sect. 3, a *Circus* program such as *BufferSpec* is linked to a *CSP* $\parallel$ *B* program by the function Υ.

Firstly, it is transformed by the R_{wrt} function to get a rewritten program *RewrittenBufferSpec*. We add two schemas named *Op_buff* and *Op_size* to retrieve state components *buff* and *size* respectively by the R_{wrt} Rule 5. Then we rename state components, schemas, actions and their references by prefixing *Buffer_* by the R_{wrt} Rule 6. The only exception is the references to state components *size* and *buff* in the action. Finally we rewrite the main action of the process *Buffer* by the R_{wrt} Rule 7. After that, we get the rewritten program as shown in Figure 9.

section *BufferSpec* **parents** *circus_toolkit*
| $maxbuff : \mathbb{N}_1$
channel $input, output : \mathbb{N}$
process *Buffer* $\widehat{=}$ **begin**
$Buffer\_State == [Buffer\_buff : \mathrm{seq}\,\mathbb{N}; Buffer\_size : 0\,..\,maxbuff \mid Buffer\_size = \#Buffer\_buff \leq maxbuff]$
$Buffer\_Init == [(Buffer\_State)' \mid Buffer\_buff' = \langle\rangle \land Buffer\_size' = 0]$
$Buffer\_InputCmd == [\Delta Buffer\_State; x? : \mathbb{N} \mid Buffer\_size < maxbuff \land Buffer\_buff' = Buffer\_buff^\frown\langle x?\rangle \land Buffer\_size' = Buffer\_size + 1]$
$Buffer\_OutputCmd == [\Delta Buffer\_State \mid Buffer\_size > 0 \land Buffer\_buff' = tail\,Buffer\_buff \land Buffer\_size' = Buffer\_size - 1]$
$Buffer\_Op\_buff == [\Xi Buffer\_State; buff! : \mathrm{seq}\,\mathbb{N} \mid buff! = Buffer\_buff]$
$Buffer\_Op\_size == [\Xi Buffer\_State; size! : 0\,..\,maxbuff \mid size! = Buffer\_size]$

$$\bullet\,(Buffer\_Init);\mu X \bullet \left(\begin{array}{l}(Buffer\_Op\_size) \rightarrow (Buffer\_Op\_buff) \rightarrow \\ \left(\begin{array}{l}(size < maxbuff)\ \&\ input?x \rightarrow (Buffer\_InputCmd) \\ \Box \\ (size > 0)\ \&\ output!(head\,buff) \rightarrow (Buffer\_OutputCmd)\end{array}\right)\end{array}\right);X$$

end

Fig. 9: Rewritten specification of Buffer

5.1.2 The Behavioural Part

Then the behavioural part of the rewritten program is translated by the Φ function ($\Phi(RewrittenBufferSpec)$) to get a CSP specification illustrated in Figure 10. The rules of the Φ function that are applied sequentially are the channel declaration Φ Rule 3, the explicitly defined process Φ Rule 19, the type and expression Φ Rule 1, the sequential composition Φ Rule 10, the simplified schema expression as action Φ Rule 9, the basic action Φ Rule 3, the prefixing Φ Rule 7, the external choice Φ Rule 11, the guarded command Φ Rule 10, and the schema expression as action Φ Rule 8.

5.1.3 The State Part

Note that the behavioural translation of the schema expressions *Buffer_InputCmd* and *Buffer_OutputCmd* in Figure 10 produces two additional schemas—*Buffer_InputCmd_f* and *Buffer_OutputCmd_f*—according to the schema expression as action Φ Rule 8, and they are added in the rewritten program *RewrittenBufferSpec* before the translation of the state part to get *RewrittenBufferSpec*1.

Eventually the state part of *RewrittenBufferSpec*1 is linked by Ω function to get a Z specification in Figure 11, where the Ω_3 function is applied in the later stage within ProB.

maxbuff = c
$channel\ input, output : Nat$
$channel\ Buffer\_Init$
$channel\ Buffer\_Op\_size : \{0..maxbuff\}$
$channel\ Buffer\_Op\_buff : Seq(Nat)$
$channel\ Buffer\_InputCmd : Nat$
$channel\ Buffer\_InputCmd\_f : Nat$
$channel\ Buffer\_OutputCmd$
$channel\ Buffer\_OutputCmd\_f$
$HIDE\_CSPB = \{|Buffer\_Init, Buffer\_Op\_size, Buffer\_Op\_buff, Buffer\_InputCmd, Buffer\_InputCmd\_f, Buffer\_OutputCmd, Buffer\_OutputCmd\_f|\}$
$Buffer = Buffer\_Init \rightarrow SKIP$; let $X =$

$$\left(\begin{array}{l}\left(\begin{array}{l}Buffer\_Op\_size?size \rightarrow Buffer\_Op\_buff?buff \rightarrow \\ \left(\begin{array}{l}(size < maxbuff)\ \&\ input?x \rightarrow \\ \quad\left(\begin{array}{l}Buffer\_InputCmd!x \rightarrow SKIP \\ \Box \\ Buffer\_InputCmd\_f!x \rightarrow \mathbf{div}\end{array}\right) \\ \Box \\ (size > 0)\ \&\ output!(head(buff)) \rightarrow \\ \quad\left(\begin{array}{l}Buffer\_OutputCmd \rightarrow SKIP \\ \Box \\ Buffer\_OutputCmd\_f \rightarrow \mathbf{div}\end{array}\right)\end{array}\right)\end{array}\right)\end{array}\right);X$$

within X

Fig. 10: The Behavioural Part Translation

5.2 Distributed Reactive Buffer

The distributed cached-head ring buffer [12], an implementation of the buffer in Sect. 5.2, is a result of final development of refinement strategies. It is composed of the process *Controller* [12, Fig. 4], the process *RingCell* [12, Sect. 7.6], the indexed ring cell process *IRCell* [12, Sect. 7.6], and the process *Ring* [12, Sect. 7.6]. Finally the process *Buffer* [12, Sect. 7.4] is a parallel composition of the process *Ring* and the process *Controller*. Obviously, this program shall have the definition of *maxbuff* and *maxring*, and the declaration of channels. In addition, we name this distributed cached-head ring buffer as *DisBuffer*, and define additional abbreviation *RingIndex*. These definitions and declarations are shown in Figure 12. Note that the original *RingCell* has no initialisation schema and we explicitly add a schema named *Init*, that has the predicate *true*. The *Init* is the first event in the main action of the *RingCell*.

$$
\begin{array}{l}
\Omega\,(RewrittenBufferSpec1) \\
= \Omega_3 \left(\Omega_2 \left(\Omega_1 \left(\begin{array}{l} \ldots \\ \textbf{process}\ Buffer \mathrel{\widehat=} \textbf{begin} \\ \ldots \\ Buffer\_InputCmd\_f == [\Xi Buffer\_State; x? : \mathbb{N} \mid \neg\, \textbf{pre}\, Buffer\_InputCmd] \\ Buffer\_OutputCmd\_f == [\Xi Buffer\_State \mid \neg\, \textbf{pre}\, Buffer\_OutputCmd] \\ \textbf{end} \end{array} \right) \right) \right) \\
= \Omega_3 \left(\Omega_2 \left(\begin{array}{l} \textbf{section}\ BufferSpec\ \textbf{parents}\ circus\_toolkit \\ |\quad maxbuff : \mathbb{N}_1 \\ Buffer\_State == [Buffer\_buff : \mathrm{seq}\,\mathbb{N}; Buffer\_size : 0\,..\,maxbuff \mid Buffer\_size = \#Buffer\_buff \leq maxbuff] \\ Init == [(Buffer\_State)' \mid Buffer\_buff' = \langle\rangle \land Buffer\_size' = 0] \\ Buffer\_InputCmd == [\Delta Buffer\_State; x? : \mathbb{N} \mid Buffer\_size < maxbuff \land Buffer\_buff' = Buffer\_buff^\frown\langle x?\rangle \land \\ \qquad Buffer\_size' = Buffer\_size + 1] \\ Buffer\_InputCmd\_f == [\Xi Buffer\_State; x? : \mathbb{N} \mid \neg\, \textbf{pre}\, Buffer\_InputCmd] \\ Buffer\_OutputCmd == [\Delta Buffer\_State \mid Buffer\_size > 0 \land Buffer\_buff' = tail\, Buffer\_buff \land Buffer\_size' = Buffer\_size - 1] \\ Buffer\_OutputCmd\_f == [\Xi Buffer\_State \mid \neg\, \textbf{pre}\, Buffer\_OutputCmd] \\ Buffer\_Op\_buff == [\Xi Buffer\_State; buff! : \mathrm{seq}\,\mathbb{N} \mid buff! = Buffer\_buff] \\ Buffer\_Op\_size == [\Xi Buffer\_State; size! : 0\,..\,maxbuff \mid size! = Buffer\_size] \end{array} \right) \right) \\
\qquad\qquad [\Omega_1\ \text{Rule 1}] \\
= \Omega_3 \left(\begin{array}{l} |\quad maxbuff : \mathbb{N}_1 \\ Buffer\_State \mathrel{\widehat=} [Buffer\_buff : \mathrm{seq}\,\mathbb{N}; Buffer\_size : 0\,..\,maxbuff \mid Buffer\_size = \#Buffer\_buff \leq maxbuff] \\ Init \mathrel{\widehat=} [Buffer\_State' \mid Buffer\_buff' = \langle\rangle \land Buffer\_size' = 0] \\ Buffer\_InputCmd \mathrel{\widehat=} [\Delta Buffer\_State; x? : \mathbb{N} \mid Buffer\_size < maxbuff \land Buffer\_buff' = Buffer\_buff^\frown\langle x?\rangle \land \\ \qquad Buffer\_size' = Buffer\_size + 1] \\ Buffer\_InputCmd\_f \mathrel{\widehat=} [\Xi Buffer\_State; x? : \mathbb{N} \mid \neg\, \textbf{pre}\, Buffer\_InputCmd] \\ Buffer\_OutputCmd \mathrel{\widehat=} [\Delta Buffer\_State \mid Buffer\_size > 0 \land Buffer\_buff' = tail\, Buffer\_buff \land Buffer\_size' = Buffer\_size - 1] \\ Buffer\_OutputCmd\_f \mathrel{\widehat=} [\Xi Buffer\_State \mid \neg\, \textbf{pre}\, Buffer\_OutputCmd] \\ Buffer\_Op\_buff \mathrel{\widehat=} [\Xi Buffer\_State; buff! : \mathrm{seq}\,\mathbb{N} \mid buff! = Buffer\_buff] \\ Buffer\_Op\_size \mathrel{\widehat=} [\Xi Buffer\_State; size! : 0\,..\,maxbuff \mid size! = Buffer\_size] \end{array} \right) \\
\qquad\qquad [\Omega_2\ \text{Rule 4.3.2}]
\end{array}
$$

Fig. 11: The State Part Translation

section *DisBufferSpec* **parents** *circus_toolkit*

$$
\begin{array}{|l}
maxbuff : \mathbb{N}_1 \\
maxring : \mathbb{N}_1 \\
\hline
maxring = maxbuff - 1
\end{array}
$$

$$RingIndex == 1\,..\,maxring$$

channel $input, output : \mathbb{N}$
channel $read, write : (RingIndex) \times \mathbb{N}$
channel $rd, wrt : \mathbb{N}$
channel $rd\_i, wrt\_i : (RingIndex) \times \mathbb{N}$

Fig. 12: The preamble of *DisBufferSpec*

5.2.1 Rewriting by R_{wrt}

Firstly, the process *IRCell* is rewritten by the R_{wrt} Rule. Particularly, the set *RingIndex* shall be determined before rewriting the *IRCell*. We assume *maxring* is equal to 3 and thus $RingIndex = 1\,..\,3$.

$$
\begin{array}{l}
R_{wrt} \left(\begin{array}{l} \textbf{process}\ IRCell \mathrel{\widehat=} \\ (i : RingIndex \odot RingCell) \\ \quad [rd\_i, wrt\_i := read, write] \end{array} \right) \\
= \left(\begin{array}{l} R_{wrt} \left(\begin{array}{l} \textbf{process}\ IRCell\_1 \mathrel{\widehat=} \\ \quad (RingCell[rd, wrt := read.1, write.1]) \end{array} \right) \\ R_{wrt} \left(\begin{array}{l} \textbf{process}\ IRCell\_2 \mathrel{\widehat=} \\ \quad (RingCell[rd, wrt := read.2, write.2]) \end{array} \right) \\ R_{wrt} \left(\begin{array}{l} \textbf{process}\ IRCell\_3 \mathrel{\widehat=} \\ \quad (RingCell[rd, wrt := read.3, write.3]) \end{array} \right) \end{array} \right) \\
\qquad\qquad [\text{Renaming Operator } R_{wrt}\ \text{Rule 4}]
\end{array}
$$

The *IRCell* is expanded to three explicitly defined processes *IRCell_1*, *IRCell_2*, and *IRCell_3*. They are the same to the *RingCell* except that the channels *rd* and *wrt* in the *RingCell* are renamed.

Along with the *Controller* and *RingCell*, now we have five explicitly defined processes. According to the R_{wrt} rule in Figure 4, five additional schemas for the *Controller* and one for the *RingCell* are added within them to access each

$\ldots$
process *Controller* $\widehat{=}$ **begin**
$\ldots$
$Op\_size == [\,\Xi ControllerState\,;size! : 0\,..\,maxbuff \mid size! = size\,]$
$Op\_ringsize == [\,\Xi ControllerState\,;ringsize! : 0\,..\,maxring \mid ringsize! = ringsize\,]$
$Op\_cache == [\,\Xi ControllerState\,;cache! : \mathbb{N} \mid cache! = cache\,]$
$Op\_top == [\,\Xi ControllerState\,;top! : RingIndex \mid top! = top\,]$
$Op\_bot == [\,\Xi ControllerState\,;bot! : RingIndex \mid bot! = bot\,]$
$\ldots$
end
process *RingCell* $\widehat{=}$ **begin**
$\ldots$
$Op\_v \ == \ [\,\Xi CellState\,;v! : \mathbb{N} \mid v! = v\,]$
$\ldots$
end
process *IRCell_1* $\widehat{=}$ **begin**
$\ldots$
$Op\_v \ == \ [\,\Xi CellState\,;v! : \mathbb{N} \mid v! = v\,]$
$\ldots$
end
$\ldots$
process *IRCell_3* $\widehat{=}$ **begin**
$\ldots$
$Op\_v \ == \ [\,\Xi CellState\,;v! : \mathbb{N} \mid v! = v\,]$
$\ldots$
end

Fig. 13: Additional Schemas for State Retrieve

state component. For *IRCell_1*, *IRCell_2*, and *IRCell_3*, they are similar. That is shown in Figure 13.

Then we rename state components, schemas, actions and their references of the *Contoller* and *RingCell* by prefixing *Controller_* and *RingCell_* respectively by the R_{wrt} Rule in Figure 5. The renamed *Controller* and *RingCell* are shown in Figure 14 and Figure 15 separately. It is also the similar case for *IRCell_1*, *IRCell_2*, and *IRCell_3*.

In the end, the main action of the *Controller* and the *RingCell* is rewritten by the R_{wrt} Rule in Figure 6. To begin with, the main action of the *Controller* is rewritten.

$$R_{wrt}\left(\begin{array}{l}(Controller\_ControllerInit)\,;\mu X \bullet \\ \quad\left(\begin{array}{l}Controller\_InputController\\ \Box\\ Controller\_OutputController\end{array}\right);X\end{array}\right)$$

$$=\left(\begin{array}{l}R_{wrt}\left(\left(Controller\_ControllerInit\right)\right);\\ R_{wrt}\left(\mu X \bullet \left(\begin{array}{l}Controller\_InputController\\ \Box\\ Controller\_OutputController\end{array}\right);X\right)\end{array}\right)$$

[Sequential Composition]

process *Controller* $\widehat{=}$ **begin**
state $Controller\_ControllerState == [$
$Controller\_size : 0\,..\,maxbuff;$
$Controller\_ringsize : 0\,..\,maxring;$
$Controller\_cache : \mathbb{N};$
$Controller\_top, Controller\_bot : RingIndex \mid$
$Controller\_ringsize \bmod maxring =$
$(Controller\_top - Controller\_bot) \bmod maxring \wedge$
$Controller\_ringsize = max\,\{\,0, Controller\_size - 1\,\}\,]$
$Controller\_Op\_size == [\,\Xi Controller\_ControllerState;$
$size! : 0\,..\,maxbuff \mid size! = Controller\_size\,]$
$Controller\_Op\_ringsize == [\,\Xi Controller\_ControllerState;$
$ringsize! : 0\,..\,maxring \mid ringsize! = Controller\_ringsize\,]$
$Controller\_Op\_cache == [\,\Xi Controller\_ControllerState;$
$cache! : \mathbb{N} \mid cache! = Controller\_cache\,]$
$Controller\_Op\_top == [\,\Xi Controller\_ControllerState;$
$top! : RingIndex \mid top! = Controller\_top\,]$
$Controller\_Op\_bot == [\,\Xi Controller\_ControllerState;$
$bot! : RingIndex \mid \quad bot! = Controller\_bot\,]$
$Controller\_ControllerInit == [\,(Controller\_ControllerState)' \mid$
$Controller\_top' = 1 \wedge Controller\_bot' = 1$
$\wedge\ Controller\_size' = 0\,]$
$Controller\_CacheInput == [\,\Delta Controller\_ControllerState;$
$x? : \mathbb{N} \mid Controller\_size = 0 \wedge Controller\_size' = 1 \wedge$
$Controller\_cache' = x? \wedge Controller\_bot' = Controller\_bot$
$\wedge\ Controller\_top' = Controller\_top\,]$
$Controller\_StoreInputController == [$
$\Delta Controller\_ControllerState \mid 0 < Controller\_size \wedge$
$Controller\_size < maxbuff \wedge$
$Controller\_size' = Controller\_size + 1 \wedge$
$Controller\_cache' = Controller\_cache \wedge$
$Controller\_bot' = Controller\_bot \wedge$
$Controller\_top' = (Controller\_top \bmod maxring) + 1\,]$
$Controller\_InputController \mathrel{\widehat{=}} (size < maxbuff)\ \&\ input?x \rightarrow$
$$\left(\begin{array}{l}(size = 0)\ \&\ (Controller\_CacheInput)\\ \Box\\ (size > 0)\ \&\ write.top!x \rightarrow\\ \qquad (Controller\_StoreInputController)\end{array}\right)$$
$Controller\_NoNewCache == [\,\Delta Controller\_ControllerState \mid$
$Controller\_size = 1 \wedge Controller\_size' = 0 \wedge$
$Controller\_cache' = Controller\_cache \wedge$
$Controller\_bot' = Controller\_bot \wedge$
$Controller\_top' = Controller\_top\,]$
$Controller\_StoreNewCacheController == [$
$\Delta Controller\_ControllerState\,; x? : \mathbb{N} \mid Controller\_size > 1$
$\wedge\ Controller\_size' = Controller\_size - 1 \wedge$
$Controller\_cache' = x? \wedge$
$Controller\_bot' = (Controller\_bot \bmod maxring) + 1 \wedge$
$Controller\_top' = Controller\_top\,]$
$Controller\_OutputController \mathrel{\widehat{=}} (size > 0)\ \&\ output!(cache)$
$$\rightarrow \left(\begin{array}{l}(size > 1)\ \&\ read.bot?x \rightarrow\\ \qquad (Controller\_StoreNewCacheController)\\ \Box\\ (size = 1)\ \&\ (Controller\_NoNewCache)\end{array}\right)$$
$$\bullet\ (Controller\_ControllerInit)\,;\mu X \bullet \left(\begin{array}{l}Controller\_InputController\\ \Box\\ Controller\_OutputController\end{array}\right);X$$
end

Fig. 14: Renaming of *Controller*

process $RingCell \mathrel{\widehat=}$ **begin**
state $RingCell\_CellState == [RingCell\_v : \mathbb{N} \mid true]$
$RingCell\_Op\_v == [\Xi RingCell\_CellState; v! : \mathbb{N} \mid v! = RingCell\_v]$
$RingCell\_Init == [(RingCell\_CellState)' \mid true]$
$RingCell\_CellWrite == [\Delta RingCell\_CellState; x? : \mathbb{N} \mid RingCell\_v' = x?]$
$RingCell\_Read \mathrel{\widehat=} rd!v \rightarrow \mathbf{Skip}$
$RingCell\_Write \mathrel{\widehat=} wrt?x \rightarrow (RingCell\_CellWrite)$
$\bullet \left(RingCell\_Init\right); \mu X \bullet \begin{pmatrix} RingCell\_Read \\ \Box \\ RingCell\_Write \end{pmatrix}; X$
end

Fig. 15: Renaming of *RingCell*

$$= \left(\begin{array}{l} \left(Controller\_ControllerInit\right); \mu X \bullet \\ R_{wrt}\left(\begin{pmatrix} Controller\_InputController \\ \Box \\ Controller\_OutputController \end{pmatrix}; X \right) \end{array} \right)$$

[Schema Expression and Recursion]

$$= \left(\cdots R_{wrt} \begin{pmatrix} Controller\_InputController \\ \Box \\ Controller\_OutputController \end{pmatrix}; R_{wrt}(X) \right)$$

[Sequential Composition]

$$= \left(\begin{array}{l} \cdots R_{mrg} \begin{pmatrix} R_{pre}(Controller\_InputController), \\ R_{pre}(Controller\_OutputController) \end{pmatrix} \rightarrow \\ \begin{pmatrix} R_{post}(Controller\_InputController) \\ \Box \\ R_{post}(Controller\_OutputController) \end{pmatrix}; X \end{array} \right)$$

[External Choice and Action Invocation]

$$= \left(\begin{array}{l} \cdots R_{mrg} \begin{pmatrix} \left(Controller\_OP\_size\right), \\ \left(Controller\_Op\_size\right) \rightarrow \\ \quad \left(Controller\_Op\_cache\right) \end{pmatrix} \rightarrow \\ \begin{pmatrix} \left(\left(size < maxbuff\right) \,\&\, input?x \rightarrow \cdots\right) \\ \Box \\ \left(\left(size > 0\right) \,\&\, output!(cache) \rightarrow \cdots\right) \end{pmatrix}; X \end{array} \right)$$

[Equation (9) and (12)]

$$= \left(\begin{array}{l} \cdots \left(Controller\_Op\_size\right) \rightarrow \left(Controller\_Op\_cache\right) \\ \rightarrow \\ \begin{pmatrix} \left(\left(size < maxbuff\right) \,\&\, input?x \rightarrow \cdots\right) \\ \Box \\ \left(\left(size > 0\right) \,\&\, output!(cache) \rightarrow \cdots\right) \end{pmatrix}; X \end{array} \right)$$

[Definition 2 of R_{mrg}]

Among them, the action *Controller_InputController* is rewritten to

$$R_{wrt}(Controller\_InputController)$$

(9)

$$= R_{wrt} \left(\begin{array}{l} \left(size < maxbuff\right) \,\&\, input?x \rightarrow \\ \left(\begin{array}{l} \left(size = 0\right) \,\&\, \left(Controller\_CacheInput\right) \\ \Box \\ \left(size > 0\right) \,\&\, write.top!x \rightarrow \\ \quad \left(Controller\_StoreInputController\right) \end{array} \right) \end{array} \right)$$

[Action Invocation]

$$= \left(\begin{array}{l} R_{pre}\left(size < maxbuff\right) \rightarrow R_{pre}\left(input?x \rightarrow (\cdots)\right) \\ \rightarrow \left(size < maxbuff\right) \,\&\, R_{post}\left(input?x \rightarrow (\cdots)\right) \end{array} \right)$$

[Guarded Command]

$$= \left(\begin{array}{l} \left(Controller\_OP\_size\right) \rightarrow \left(size < maxbuff\right) \,\&\, \\ R_{post}\left(input?x \rightarrow (\cdots)\right) \end{array} \right)$$

[Definition 1 of R_{pre} and R_{post}]

$$= \left(\begin{array}{l} \left(Controller\_OP\_size\right) \rightarrow \left(size < maxbuff\right) \,\&\, input?x \\ \rightarrow R_{wrt} \begin{pmatrix} \left(size = 0\right) \,\&\, \left(Controller\_CacheInput\right) \\ \Box \\ \left(size > 0\right) \,\&\, write.top!x \rightarrow \\ \quad \left(Controller\_StoreInputController\right) \end{pmatrix} \end{array} \right)$$

[Prefixing]

$$= \left(\begin{array}{l} \cdots \\ R_{mrg} \begin{pmatrix} R_{pre}\left(\left(size = 0\right) \,\&\, \left(\cdots\right)\right), \\ R_{pre}\left(\left(size > 0\right) \,\&\, write.top!x \rightarrow \left(\cdots\right)\right) \end{pmatrix} \\ \rightarrow \begin{pmatrix} R_{post}\left(\left(size = 0\right) \,\&\, \left(\cdots\right)\right) \\ \Box \\ R_{post}\left(\left(size > 0\right) \,\&\, write.top!x \rightarrow \left(\cdots\right)\right) \end{pmatrix} \end{array} \right)$$

[External Choice]

$$= \left(\begin{array}{l} \cdots \\ R_{mrg} \begin{pmatrix} \left(Controller\_OP\_size\right), \\ \left(Controller\_OP\_size\right) \rightarrow \\ \quad \left(Controller\_OP\_top\right) \end{pmatrix} \\ \rightarrow \begin{pmatrix} \left(size = 0\right) \,\&\, \left(Controller\_CacheInput\right) \\ \Box \\ \left(size > 0\right) \,\&\, write.top!x \rightarrow \\ \quad \left(Controller\_StoreInputController\right) \end{pmatrix} \end{array} \right)$$

[Equation (10) and (11)]

$$= \left(\begin{array}{l} \left(Controller\_OP\_size\right) \rightarrow \left(size < maxbuff\right) \,\& \\ input?x \rightarrow \left(Controller\_OP\_size\right) \rightarrow \\ \left(Controller\_OP\_top\right) \rightarrow \\ \begin{pmatrix} \left(size = 0\right) \,\&\, \left(Controller\_CacheInput\right) \\ \Box \\ \left(size > 0\right) \,\&\, write.top!x \rightarrow \\ \quad \left(Controller\_StoreInputController\right) \end{pmatrix} \end{array} \right)$$

[Definition 2 of R_{mrg}]

where

$$R_{wrt}\left((size=0)\,\&\,\left(Controller\_CacheInput\right)\right)$$
$$=\left(\begin{array}{l}\left(Controller\_OP\_size\right)\rightarrow(size=0)\,\&\\ R_{post}\left(\left(Controller\_CacheInput\right)\right)\end{array}\right)$$

[Guarded Command and Definition 1 of R_{pre} and R_{post}]

$$=\left(\begin{array}{l}\left(Controller\_OP\_size\right)\rightarrow(size=0)\,\&\\ \left(Controller\_CacheInput\right)\end{array}\right)$$

[Schema Expression]

(10)

and

$$R_{wrt}\left(\begin{array}{l}(size>0)\,\&\,write.top!x\rightarrow\\ \quad\left(Controller\_StoreInputController\right)\end{array}\right)$$
$$=\left(\begin{array}{l}\left(Controller\_OP\_size\right)\rightarrow R_{pre}\,(write.top!x\rightarrow\cdots)\\ (size>0)\,\&\,R_{post}\left(write.top!x\rightarrow\left(\cdots\right)\right)\end{array}\right)$$

[Guarded Command and Definition 1 of R_{pre} and R_{post}]

$$=\left(\begin{array}{l}\left(Controller\_OP\_size\right)\rightarrow\left(Controller\_OP\_top\right)\\ \rightarrow(size>0)\,\&\\ write.top!x\rightarrow\left(Controller\_StoreInputController\right)\end{array}\right)$$

[Prefixing and Schema Expression]

(11)

Analogous to the rewriting of *Controller_InputController*, *Controller_OutputController* is rewritten to

$$R_{wrt}\,(Controller\_OutputController)$$
$$=\left(\begin{array}{l}\left(Controller\_Op\_size\right)\rightarrow\left(Controller\_Op\_cache\right)\rightarrow\\ (size>0)\,\&\,output!(cache)\rightarrow\left(Controller\_Op\_size\right)\\ \quad\rightarrow\left(Controller\_Op\_bot\right)\\ \quad\rightarrow\left(\begin{array}{l}(size>1)\,\&\,read.bot?x\rightarrow\\ \quad\left(Controller\_StoreNewCacheController\right)\\ \Box\\ (size=1)\,\&\left(Controller\_NoNewCache\right)\end{array}\right)\end{array}\right)$$

(12)

In addition, it is the similar case for the rewriting of the main action of the *RingCell*, *IRCell_1*, *IRCell_2* and *IRCell_3* as well. Eventually, their main actions after rewriting are illustrated in Figure 16.

5.2.2 The Behavioural Part

The behavioural part of the rewritten program is translated by the Φ function to get a CSP specification.

process *Controller* $\widehat{=}$ **begin**

$$\left(\begin{array}{l}\ldots\\ \bullet\left(Controller\_ControllerInit\right);\mu X\bullet\left(Controller\_Op\_size\right)\\ \rightarrow\left(Controller\_Op\_cache\right)\rightarrow\\ \left(\begin{array}{l}\left(size<maxbuff\right)\,\&\,input?x\rightarrow\left(Controller\_Op\_size\right)\\ \quad\rightarrow\left(Controller\_Op\_top\right)\\ \quad\rightarrow\left(\begin{array}{l}\left(size=0\right)\,\&\left(Controller\_CacheInput\right)\\ \Box\\ \left(size>0\right)\,\&\,write.top!x\rightarrow\\ \quad\left(Controller\_StoreInputController\right)\end{array}\right)\\ \Box\\ \left(size>0\right)\,\&\,output!(cache)\rightarrow\left(Controller\_Op\_size\right)\\ \quad\rightarrow\left(Controller\_Op\_bot\right)\\ \quad\rightarrow\left(\begin{array}{l}\left(size>1\right)\,\&\,read.bot?x\rightarrow\\ \quad\left(Controller\_StoreNewCacheController\right)\\ \Box\\ \left(size=1\right)\,\&\left(Controller\_NoNewCache\right)\end{array}\right)\end{array}\right);X\end{array}\right)$$

end

process *RingCell* $\widehat{=}$ **begin**

$$\left(\begin{array}{l}\ldots\\ \bullet\left(RingCell\_Init\right);\mu X\bullet\left(RingCell\_Op\_v\right)\rightarrow\\ \left(\begin{array}{l}rd!v\rightarrow\textbf{Skip}\\ \Box\\ wrt?x\rightarrow\left(RingCell\_CellWrite\right)\end{array}\right);X\end{array}\right)$$

end

process *IRCell_1* $\widehat{=}$ **begin**

$$\left(\begin{array}{l}\ldots\\ \bullet\left(IRCell\_1\_Init\right);\mu X\bullet\left(IRCell\_1\_Op\_v\right)\rightarrow\\ \left(\begin{array}{l}read.1!v\rightarrow\textbf{Skip}\\ \Box\\ write.1?x\rightarrow\left(IRCell\_1\_CellWrite\right)\end{array}\right);X\end{array}\right)$$

end

process *IRCell_2* $\widehat{=}$ **begin**

$$\left(\begin{array}{l}\ldots\\ \bullet\left(IRCell\_2\_Init\right);\mu X\bullet\left(IRCell\_2\_Op\_v\right)\rightarrow\\ \left(\begin{array}{l}read.2!v\rightarrow\textbf{Skip}\\ \Box\\ write.2?x\rightarrow\left(IRCell\_2\_CellWrite\right)\end{array}\right);X\end{array}\right)$$

end

process *IRCell_3* $\widehat{=}$ **begin**

$$\left(\begin{array}{l}\ldots\\ \bullet\left(IRCell\_3\_Init\right);\mu X\bullet\left(IRCell\_3\_Op\_v\right)\rightarrow\\ \left(\begin{array}{l}read.3!v\rightarrow\textbf{Skip}\\ \Box\\ write.3?x\rightarrow\left(IRCell\_3\_CellWrite\right)\end{array}\right);X\end{array}\right)$$

end

Fig. 16: Rewrite of the main actions

First of all, the axiomatic definition of the *maxbuff* and the *maxring* is translated to

$$maxbuff = c$$
$$maxring = c - 1$$

by the Φ Rule 2, where c is a constant that is manually assigned before the model checking. For example, $c = 3$.

The abbreviation definition *RingIndex* is transformed to

$$\text{nametype } RingIndex = \{1..maxring\}$$

by the Φ Rule 1.

The channel declarations are transformed to

$$\text{channel } input, output : Nat$$
$$\text{channel } read, write : RingIndex.Nat$$
$$\text{channel } rd, wrt : Nat$$
$$\text{channel } rd\_i, wrt\_i : RingIndex.Nat$$

by the Φ Rule 3 and their expressions are transformed by the Φ Rule 1.

The behaviour of the *Controller* process is specified by its main action $ma(Controller)$ in Figure 16.

$$\Phi(\textbf{process } Controller \mathrel{\widehat{=}} \dots) =$$
$$Controller = \Phi(ma(Controller)) \qquad [\Phi \text{ Rule 19}]$$

Then its main action is translated. Here, the additional channel declarations and events added in HIDE_CSPB are omitted. In addition, when a schema expression is linked by Φ Rule 8, an additional schema is added in the state part. For example, $\Phi\left(\left(Controller\_CacheInput\right)\right)$ generates the schema $Controller\_CacheInput\_f$. This is omitted here as well.

$$\Phi(ma(Controller))$$
$$= \Phi\left(\left(Controller\_ControllerInit\right)\right); \Phi(\mu X \bullet \dots)$$
[Sequential Composition Φ Rule 10]
$$= Controller\_ControllerInit \rightarrow SKIP; \Phi(\mu X \bullet \dots)$$
[Schema Expression Φ Rule 9, Basic Actions Φ Rule 6]
$$= \cdots \text{let } X = \Phi \begin{pmatrix} \left(Controller\_Op\_size\right) \rightarrow \\ \left(Controller\_Op\_cache\right) \rightarrow \\ \cdots \end{pmatrix} \text{ within } X$$
[Recursion Φ Rule 10]
$$= \cdots \begin{pmatrix} Controller\_Op\_size?size \rightarrow \\ Controller\_Op\_cache?cache \rightarrow \\ \Phi(\cdots \Box \cdots) \end{pmatrix}$$
[Schema Expression Φ Rule 9, Prefixing Φ Rule 7]
$$= \cdots \Phi\left(\left(size < maxbuff\right) \,\&\, \cdots\right) \Box \Phi\left(\left(size > 0\right) \,\&\, \cdots\right)$$
[External Choice Φ Rule 11]
$$= \cdots ((size < maxbuff) \;\&\; \Phi(input?x \rightarrow \cdots) \Box \cdots)$$
[Guarded Command Φ Rule 10]
$$= \cdots \begin{pmatrix} input?x \rightarrow Controller\_Op\_size?size \rightarrow \\ Controller\_Op\_top?top \rightarrow \Phi(\cdots) \end{pmatrix} \Box \cdots$$
[Prefixing Φ Rule 7, Schema Expression Φ Rule 9]
$$= \begin{pmatrix} \cdots \\ \begin{pmatrix} (size == 0) \;\&\; \Phi\left(\left(Controller\_CacheInput\right)\right) \\ \Box \\ (size > 0) \;\&\; write.top!x \rightarrow \\ \quad \Phi\left(\left(Controller\_StoreInputController\right)\right) \end{pmatrix} \\ \Box \cdots \end{pmatrix}$$
[Guarded Command Φ Rule 10, Expression, Prefixing]
$$= \begin{pmatrix} \cdots \\ \begin{pmatrix} \cdots \begin{pmatrix} Controller\_CacheInput!x \rightarrow SKIP \\ \Box \\ Controller\_CacheInput\_f!x \rightarrow \textbf{div} \end{pmatrix} \\ \Box \\ \cdots \begin{pmatrix} Controller\_StoreInputController \rightarrow SKIP \\ \Box \\ Controller\_StoreInputController\_f \rightarrow \textbf{div} \end{pmatrix} \end{pmatrix} \\ \Box \cdots \end{pmatrix}$$
[Schema Expression Φ Rule 8]
$$= \cdots \Box \Phi\left(\left(size > 0\right) \,\&\, \cdots\right) \qquad \text{[Similar to previous steps]}$$

The behaviour of the *RingCell* process is given by its main action $ma(RingCell)$ in Figure 16. Its translation is similar to that of the *Controller* process.

$$\Phi(ma(RingCell))$$
$$= \cdots$$
$$= \begin{pmatrix} RingCell\_Init \rightarrow SKIP; \text{let } X = \\ \begin{pmatrix} RingCell\_Op\_v?v \rightarrow \\ \begin{pmatrix} rd!v \rightarrow SKIP \\ \Box \\ wrt?x \rightarrow \\ \quad \begin{pmatrix} RingCell\_CellWrite!x \rightarrow SKIP \\ \Box \\ RingCell\_CellWrite\_f!x \rightarrow \textbf{div} \end{pmatrix} \end{pmatrix} \\ ;X \end{pmatrix} \\ \text{within } X \end{pmatrix}$$
[Φ Rules]

The behaviour of *IRCell_1*, *IRCell_2* and *IRCell_3* is their main actions as well. Only the translation of *IRCell_1* is displayed below.

$$\Phi(ma(IRCell\_1))$$

$$= \cdots$$

$$= \left(\begin{array}{l} IRCell\_1\_Init \rightarrow SKIP ; \text{let } X = \\ \quad \left(\begin{array}{l} IRCell\_1\_Op\_v?v \rightarrow \\ \quad \left(\begin{array}{l} read.1!v \rightarrow SKIP \\ \quad \Box \\ \quad write.1?x \rightarrow \\ \qquad \left(\begin{array}{l} IRCell\_1\_CellWrite!x \rightarrow SKIP \\ \quad \Box \\ IRCell\_1\_CellWrite\_f!x \rightarrow \textbf{div} \end{array} \right) \end{array} \right) \\ \quad ; X \end{array} \right) \\ \text{within } X \end{array} \right)$$

[Φ Rules]

The *Ring* process is the interleaving of the indexed *IRCell* processes. It is link to

$$\Phi \left(Ring \mathrel{\widehat=} \left(\left|\left|\left| i : RingIndex \bullet IRCell\lfloor i \rfloor \right.\right.\right) \right)$$

$$= \left(Ring = \left|\left|\left|_{i:RingIndex} \bullet \Phi \left(IRCell\lfloor i \rfloor \right) \right.\right.\right)$$

[Iterated Interleave Φ Rule 22]

$$= \left(Ring = \left|\left|\left|_{i:RingIndex} \bullet \left(\begin{array}{l} (i == 1) \ \& \ IRCell\_1 \\ \Box \ (i == 2) \ \& \ IRCell\_2 \\ \Box \ (i == 3) \ \& \ IRCell\_3 \end{array} \right) \right.\right.\right)$$

[Indexed Process Invocation Φ Rule 24]

.

The *Buffer* process is translated to the *Buffer* process in CSP below.

$$\Phi \left(\begin{array}{l} \textbf{process } Buffer \mathrel{\widehat=} \\ (\, Controller \, [\![\, \{\!| read, write |\!\} \,]\!] \, Ring \,) \setminus \{\!| read, write |\!\} \end{array} \right)$$

$$= \left(\begin{array}{l} Buffer = \\ \Phi \left(Controller \underset{\{|read,write|\}}{\parallel} Ring \right) \setminus \{|read, write|\} \end{array} \right)$$

[Process hide Φ Rule 21, Channel Set Φ Rule 4]

$$= \left(\begin{array}{l} Buffer = \\ \left(Controller \underset{\{|read,write|\}}{\parallel} Ring \right) \setminus \{|read, write|\} \end{array} \right)$$

[Parallel Φ Rule 21, Process Invocation Φ Rule 20]

The *Buffer* process is the main process of the *DisBufferSpec* program. Thus when translated to CSP, the *MAIN* process is the *Buffer*.

$$MAIN = Buffer$$

5.2.3 The State Part

After the rewriting in Sect. 5.2.1 and the behaviour translation in Sect. 5.2.2, we have five explicitly defined processes: *Controller*, *RingCell*, *IRCell_1*, *IRCell_2*, and *IRCell_3*.

For the *Controller*, additional schemas are added by the schema expression as action Φ Rule 8 during the link of the behaviour to CSP.

- *Controller_CacheInput_f*
- *Controller_StoreInputController_f*
- *Controller_NoNewCache_f*
- *Controller_StoreNewCacheController_f*

For the *RingCell*, *IRCell_1*, *IRCell_2*, and *IRCell_3*, one additional schema is added for each process. They are *RingCell_CellWrite_f*, *IRCell_1_CellWrite_f*, *IRCell_2_CellWrite_f*, and *IRCell_3_CellWrite_f* separately.

The state part of the program is linked to a B machine by the Ω function that is composed of the Ω_1, the Ω_2 and the Ω_3. The Ω_3 is applied within ProB and so it is skipped here.

To begin with, the axiomatic definition and the abbreviation *RingIndex* are moved to the Z program directly.

Then the states and schemas from five explicitly defined processes are merged by the Ω_1 Rule 1. After that, the Ω_2 function is simply applied. Finally the state part is linked to a Z program.

The state schemas are merged as shown.

$$\begin{array}{l} Controller\_ControllerState \mathrel{\widehat=} [\\ \quad Controller\_size : 0 \,..\, maxbuff ; \\ \quad Controller\_ringsize : 0 \,..\, maxring ; \\ \quad Controller\_cache : \mathbb{N} ; \\ \quad Controller\_top, Controller\_bot : RingIndex \mid \\ \quad Controller\_ringsize \bmod maxring = \\ \qquad (Controller\_top - Controller\_bot) \bmod maxring \land \\ \quad Controller\_ringsize = max \{ 0, Controller\_size - 1 \} \\ ] \\ RingCell\_CellState \mathrel{\widehat=} [RingCell\_v : \mathbb{N} \mid true] \\ IRCell\_1\_CellState \mathrel{\widehat=} [RingCell\_1\_v : \mathbb{N} \mid true] \\ IRCell\_2\_CellState \mathrel{\widehat=} [RingCell\_2\_v : \mathbb{N} \mid true] \\ IRCell\_3\_CellState \mathrel{\widehat=} [RingCell\_3\_v : \mathbb{N} \mid true] \\ State \mathrel{\widehat=} Controller\_ControllerState \land \\ \quad RingCell\_CellState \land IRCell\_1\_CellState \land \\ \quad IRCell\_2\_CellState \land IRCell\_3\_CellState \end{array}$$

The initialisation schemas are merged.

$$\begin{array}{l} Init \mathrel{\widehat=} [State' \mid Controller\_top' = 1 \land \\ \quad Controller\_bot' = 1 \land Controller\_size' = 0 \land true] \end{array} \quad (13)$$

All other schemas are merged as well. However for a schema from one process, it shall include the state schemas from other processes in its declaration part by Ξ notation to make sure no change is made to the state components from others. For example, the *Controller_Op_size* schema from the *Controller* becomes

$$\begin{aligned}&Controller\_Op\_size \mathrel{\widehat=} [\Xi Controller\_ControllerState;\\ &\quad \Xi RingCell\_CellState; \Xi RingCell\_1\_CellState;\\ &\quad \Xi RingCell\_2\_CellState; \Xi RingCell\_3\_CellState;\\ &\quad size! : 0 \mathinner{..} maxbuff \mid size! = Controller\_size]\end{aligned}$$

, *Controller_CacheInput* becomes

$$\begin{aligned}&Controller\_CacheInput \mathrel{\widehat=} [\Delta Controller\_ControllerState;\\ &\quad \Xi RingCell\_CellState; \Xi RingCell\_1\_CellState;\\ &\quad \Xi RingCell\_2\_CellState; \Xi RingCell\_3\_CellState;\\ &\quad x? : \mathbb{N} \mid Controller\_size = 0 \land\\ &\quad Controller\_size' = 1 \land Controller\_cache' = x? \land\\ &\quad Controller\_bot' = Controller\_bot \land\\ &\quad Controller\_top' = Controller\_top]\end{aligned}$$

, and *Controller_CacheInput_f* is transformed to

$$\begin{aligned}&Controller\_CacheInput\_f \mathrel{\widehat=} [\\ &\quad \Xi Controller\_ControllerState;\\ &\quad \Xi RingCell\_CellState; \Xi RingCell\_1\_CellState;\\ &\quad \Xi RingCell\_2\_CellState; \Xi RingCell\_3\_CellState;\\ &\quad x? : \mathbb{N} \mid \neg\, \mathbf{pre}\, Controller\_CacheInput]\end{aligned}$$

The translation of other schemas are very similar and thus skipped.

5.3 Model Checking By ProB

Now we have got the final CSP program (Figure 10) and the Z program (Figure 11) for the buffer specification, and the CSP program (Sect. 5.2.2) and the Z program (Sect. 5.2.3) for the distributed buffer. Both of them can be model-checked by ProB. But before performing model checking, the value of the constants *MAXINT*, *MAXINS* and *maxbuff* shall be considered at first.

5.3.1 Maximum Instances *MAXINS* and Maximum Size of Buffer *maxbuff*

For the buffer specification, the type of buffer is $seq\,\mathbb{N}$. When linked to $CSP \parallel B$, we use *fseq* (Figure 8) that introduces the bound constant *MAXINS*. Finally the size of the set of finite sequences by *fseq* highly relies on the value of *MAXINS* as well as the data set *s*. The defined *fseq* computes the result relied on several intermediate functions (*squash*, *pfun*, *rel* and *cross*) which are defined in the functional language as well. The consumption of resources during resolution is still high. If the size of *s* is big and *MAXINT* is large, ProB will take longer time to compute all possible finite sequences. For an instance, on the system having 2GB RAM and 2.5 GHz CPU, and running Ubuntu 12.04, it takes approximately thirty minutes for ProB to load the CSP program when the size of *s* is 4 (*MAXINT* is set to 3) and *MAXINS* is 5. However if the size of *s* is reduced to 2, we can increase *MAXINS* to 9 to make ProB load the CSP program still in a shorter time. Alternatively, instead of using the functional language to resolve *fseq*, we can compute all finite sequences in advance by another language, let's say Perl, then include them explicitly into a set and replace *fseq* in CSP programs by this set. For example, if *s* is $\{0,1\}$ and *MAXINT* is 2, then we can get this set as $\{\langle\rangle, \langle 0\rangle, \langle 1\rangle, \langle 0,0\rangle, \langle 0,1\rangle, \langle 1,0\rangle, \langle 1,1\rangle\}$. By this way, it can reduce the program load time tremendously. But the loss of flexibility is a side effect because we have to compute this set in advance and externally (out of ProB).

For the buffer specification the value of *maxbuff* shall be less than or equal to *MAXINS*.

5.3.2 Data Independence and *MAXINT*

The type of data (T) in both the buffer specification and the buffer implementation is $\mathbb{N}$. According to the definition of data independence [28, Sect. 2.7] and [43, Sect. 15.3.2], both the linked buffer programs in $CSP \parallel B$ are data-independent because they input values of $\mathbb{N}$ along their *input* channels, store them in a sequence or a set of ring cells, and then output values in order along their *output* channels without any computations. And they do not perform any explicit and implicit equality tests over T, therefore they satisfy **NoEqT** [28, 42]. In addition, the *Buffer* process in the linked buffer specification (Figure 10) satisfies **Norm** [28, 42]. According to Theorem 17.2 [42], the threshold of the size of T such that the implementation is a refinement of the specification in terms of traces, failures and failures-divergences is 2. There is a similar conclusion in the book [42, p.397] that the threshold of an N-bounded buffer for any N is 2. So for the refinement check, we can set *MAXINT* to 1 as there are two elements $\{0,1\}$ and set *maxbuff* to 3. Actually, we also checked the refinement when *MAXINT* is increased to 3.

5.3.3 Model Checking of Buffer Specification

When model-checking this case by ProB, we notice ProB kernel treats seq T as $set(couple(integer, T))$ in Z and B. But it fails to match the sequence type in CSP. Therefore, it generates an incompatible type error. We change the implementation of predicate `type_ok` and `is_csp_set_type`

Table 1: Model Checking Performance Comparison (Buffer Specification)

MAXINT	*MAXINS*	*maxbuff*	Time (*ms*)	Memory (MB)
3	3	1	122	38
3	3	2	538	38
3	3	3	2,152	39
3	4	4	28,022	40
3	5	5	443,306	78
1	9	9	16,802	68

in `specfile.pl` of ProB kernel source code to make $set(couple(integer, T))$ match the sequence type in CSP.

Additionally, **div**, the most divergent process, is not available in CSP_M as well as ProB. We define a process DIV as $div \rightarrow STOP$, where *div* is a special event, in CSP for **div**. Though DIV is not a divergent process, we can check deadlock of combination of CSP and Z specifications to achieve divergence checking. We use the deadlock checking to find this kind of divergence because it is a more direct checking in ProB. In case that a deadlock is found, we check the counterexample to see if the last event is *div* or not. If the last event is *div*, it means the original *Circus* specification can lead to divergence. Alternatively, LTL formula checking can be used to check deadlock as well. For example, the LTL formula (`not F e(div)`), which denotes the statement that finally *div* event is enabled, is not true. When it comes to this case, if we remove guarded conditions in *Input* or *Output* action in Figure 2, the specification diverges because the precondition of *InputCmd* and *OutputCmd* may not hold. In the final CSP specification in Figure 10, the corresponding boolean guard is removed as well. Using ProB, we can easily find the deadlock and the last event is *div*, therefore it finds divergence.

Deadlock and Invariant Violation Checking Finally, we can model-check the combination of CSP and Z specifications and there is no deadlock found. A comparison of the model checking performance for different configuration of constants is shown in Table 1. This experiment was undertaken on ProB Linux version, which is modified based on ProB 1.5.0-Beta, on Ubuntu.

Deadlock and Divergence Checking by CSP Assertions ProB is capable of deadlock and divergence checking through CSP assertions as well. By adding the following three asserts to the CSP program (Figure 10), we checked the deadlock free and divergence free of the *Buffer* process with the combination of constants in Table 1 successfully.

```
assert Buffer :[ deadlock [F] ]
assert Buffer :[ deadlock [FD] ]
assert Buffer :[ livelock free ]
```

5.3.4 Model Checking of Distributed Reactive Buffer

One issue we found is about the well-definedness of the modulo operation in Z when it is translated to the counterpart in B. In Z, the modulo operation is defined on the integer dividend and the non-zero integer divisor [47]. However it is defined on the natural number dividend and the non-zero natural number divisor in B machine. Therefore, when model-checking this case by ProB that translates the modulo to the modulo operation in B, it triggers an error about the well-definedness of the modulo because the dividend of the modulo in Z is possibly less than 0. Thus, we modified the implementation of the modulo operation in ProB to use the modulo operation *mod* in SICStus Prolog. Because the modulo operation in Z uses truncation towards minus infinity [47] and in Prolog it is the integer remainder after floored division [2], they use the same definition of modulo—floored division [29]. Hence, the well-definedness of mod in Z is retained.

In addition, ProB uses the build-in command time in Tcl to measure the elapsed time for the model checking task. It can count up to 4,294,967,295 microseconds, approximately 72 minutes, for a task in a 32-bit machine, otherwise it will cause the overflow. For the model checking of this case with the *maxbuff* larger than 3, it requires longer time and causes the overflow. Therefore, the output result about the time is not useful. We record the timestamp before the task execution and the timestamp after the completion of the task by `clock milliseconds` in Tcl, then calculate the difference between two timestamps. This is the model checking time.

Deadlock and Invariant Violation Checking There is no deadlock or divergence found. A comparison of the model checking performance is shown in Table 2. Note that due to the state space exploration and resource limitation, we are not able to model check this case if *maxbuff* is larger than 3 and *MAXINT* is 3 because ProB runs out of memory on Linux with 3 GB memory. We can set the *MAXINT* to 1 to reduce the state space. The result is shown in the third row. Further methods like more specific initialisation can be used to tremendously decrease the size of the state space. For an instance, if we substitute (13) by the initialisation schema (14), the model checking result is displayed in the fourth row.

$$\begin{aligned} Init \mathrel{\widehat=} [State' \mid & Controller\_top' = 1 \land Controller\_bot' = 1 \\ & \land Controller\_size' = 0 \land Controller\_ringsize' = 0 \\ & \land Controller\_cache' = 0 \land RingCell\_v' = 0 \\ & \land IRCell\_1\_v' = 0 \land IRCell\_2\_v' = 0 \\ & \land IRCell\_3\_v' = 0 \land IRCell\_4\_v' = 0] \end{aligned} \tag{14}$$

Deadlock and Divergence Checking by CSP Assertions By adding the following three asserts to the CSP program (Sect.

Table 2: Model Checking Performance Comparison (Buffer Implementation)

MAXINT	*maxbuff*	*maxring*	Time (*ms*)	Memory (MB)
3	2	1	38,039	53
3	3	2	2,582,944	837
1	4	3	951,593	913
1	4	3	236,532	318 [a]

[a] This row is the result with substituted initialisation schema.

5.2.2), we checked the deadlock free and divergence free of the *Buffer* process with the combination of constants in Table 2 successfully.

```
assert Buffer :[ deadlock [F] ]
assert Buffer :[ deadlock [FD] ]
assert Buffer :[ livelock free ]
```

5.3.5 Refinement Checking

In addition, ProB can check if an implementation in $CSP \parallel B$ is a trace refinement of a specification in $CSP \parallel B$ [30]. When checking the trace refinement, an issue we got in ProB for our case, after inspecting source code, is that ProB refinement checker compares the traces of both the specification and the implementation according to their transitions in the same state space. That works for the refinement of two processes in the same CSP program for the CSP model, or the refinement of two processes in the same CSP program for the $CSP \parallel B$ model. But for our case, the specification and the implementation have the different Z programs and it is impossible to put their CSP programs into one CSP file. Thus we modified the `refinement_checker.pl` of ProB to search the traces by the transitions from their own separate state space. After model-checking the buffer specification, we save its state space for later refinement check to a file. Then we load the buffer implementation to ProB, and select "trace refinement check" function, open the saved state space file for the specification. Finally ProB will show the result: the implementation is a trace refinement of the specification; or if not a trace refinement, a counter example is provided.

We checked the trace refinement between the buffer specification and the buffer implementation, and finally got the result the distributed reactive buffer is a trace refinement of the buffer specification for *MAXINT* and *maxbuff* equal to 3 and 3 separately. Furthermore, ProB has an option to check failures. We checked the failure refinement between the specification and the implementation as well, and finally found the distributed reactive buffer is also a failure refinement of the buffer specification with the same constants. The refinement checking performance is shown in Table 3. According to Sect. 5.3.2, we can conclude the buffer implementation is a failure refinement of the buffer specification.

Table 3: Refinement Check Performance

Model	*MAXINT*	*maxbuff*	*maxring*	Time (*ms*)
Traces	1	3	2	109,180
Failures	1	3	2	122,440
Traces	3	3	2	342,440
Failures	3	3	2	355,320

However, if *maxbuff* between the specification and the implementation is not equal, ProB gives an error with a counterexample provided.

6 Conclusions and Future Work

Related work Model checking and animation are regarded as a very important tool support for the application of formalisms in both academia and industry. There are three existing solutions for implementing or model-checking *Circus* programs. The first solution is *JCircus* [21, 36, 40] which translates a concrete *Circus* program to a Java program with JCSP [48]. After that, linking *Circus* to CSP [7] aims to translate *Circus* to CSP_M then use FDR2 [1] to model-check CSP specification. The last one is mapping *Circus* processes and refinement to CSP processes and refinement [31,39] that transforms stateful *Circus* to stateless *Circus* first by introducing the memory model, and then converts stateless *Circus* to CSP. The first is not a model checking solution but implementation instead, and it is restricted to executable *Circus* programs because Java is an imperative programming language and not a high-level specification language. Therefore, before supplying the *Circus* program to *JCircus*, it has to be refined to a concrete program. For the second solution, it is not clear how to connect the data part to the behavioural part of a *Circus* program. The third solution transforms both state and behavioural parts to CSP specification, which means all states are maintained in CSP. It is restricted to divergence-free *Circus*. Furthermore, it is not convenient and capable in CSP to maintain very complex states, and rather difficult to understand the final CSP specification if it contains a lot of state operations.

Our work Comparatively, our work links *Circus* to $CSP \parallel B$ to express the behavioural and state parts, which maintains the high-level abstraction. And using B to specify the state part is more straightforward and easier than using CSP. The capability of linking most of constructs in *Circus* to $CSP \parallel B$ is another advantage because *Circus* itself consists of a large amount of syntactic constructs. Additionally, the capabilities of deadlock checking, LTL formula checking, refinement checking, automatic and manual animations are very important as well. Last but not least, we achieve the divergence checking of original *Circus* program by deadlock checking of $CSP \parallel B$.

However, this work has some limitations as well. *Generics* in *Circus* are not supported; for *CSP* ‖ *B*, its state is composed of the state from B and the state of the process from CSP, and is expressed as a pair [9], thus its state space is more complex for exploration; limited actions can occur in both sides of external choice (Φ Rule 11) due to the semantics of external choice [37]; the rule of parallel composition of actions (Φ Rule 13) achieves semantical equality but is difficult to implement because we need to keep a copy of temporary variables in CSP for both state variables in B and local variables, which makes the state maintained in CSP temporarily before merge and is against our strategy that state and behaviour are separated in B and CSP; how to trace *CSP* ‖ *B* back to *Circus* is an issue.

Future work By now, we have defined the translation rules for a large subset of constructs in *Circus* , given the soundness, and developed a simple translator which can deal with very limited rules. We will continue to study a more complicated case, extend the translator to support approximately all rules defined. Most significantly, we need to minimize the limitations. For instance, we may modify the operational semantics of *CSP* ‖ *B* in ProB to make external choice resolved only by external events and termination but not communication between CSP and B, which makes all actions can occur in external choice. In addition, for link of axiomatic definition to CSP (Φ Rule 2), we will modify the implementation of ProB to make it instantiated to the same value as in Z.

Acknowledgements We thank Leo Freitas and Andrew Butterfield for discussions about the link approach, CZT as well as the insights of difficulties.

References

1. FDR2: a refinement checker for establishing properties of models expressed in CSP. URL www.fsel.com/software.html
2. SICStus Prolog Manual (Arithmetic Expressions). URL https://sicstus.sics.se/sicstus/docs/latest4/html/sicstus.html/ref_002dari_002daex.html#ref_002dari_002daex
3. The ProB Animator and Model Checker. URL http://www.stups.uni-duesseldorf.de/ProB/index.php5/Main_Page
4. ISO/IEC: Information Technology-Z Formal Specification Notation-Syntax, Type System and Semantics. (2002). URL http://standards.iso.org/ittf/PubliclyAvailableStandards/c021573_ISO_IEC_13568_2002(E).zip
5. Abrial, J.R.: The B-book: assigning programs to meanings. Cambridge University Press (2005)
6. Abrial, J.R.: Modeling in Event-B: System and Software Engineering. Cambridge University Press (2010)
7. Beg, A., Butterfield, A.: Linking a state-rich process algebra to a state-free algebra to verify software/hardware implementation. In: FIT, p. 47 (2010)
8. Bergstra, J.A., Klop, J.W.: Process Algebra for Synchronous Communication. Information and Control **60**(1-3), 109–137 (1984)
9. Butler, M., Leuschel, M.: Combining CSP and B for Specification and Property Verification pp. 221–236 (2005)
10. Carlsson, M.: Sicstus PROLOG User's Manual 4.2. Books On Demand - Proquest (2012)
11. Carrington, D.A., Duke, D.J., Duke, R., King, P., Rose, G.A., Smith, G.: Object-Z: An Object-Oriented Extension to Z. In: FORTE, pp. 281–296 (1989)
12. Cavalcanti, A., Sampaio, A., Woodcock, J.: A Refinement Strategy for Circus. Formal Asp. Comput. **15**(2-3), 146–181 (2003)
13. Cavalcanti, A., Woodcock, J.: A Tutorial Introduction to CSP in *unifying theories of programming*. In: PSSE, pp. 220–268 (2004)
14. Clarke Jr., E.M., Grumberg, O., Peled, D.A.: Model checking. MIT Press, Cambridge, MA, USA (1999)
15. Clearsy: B LANGUAGE REFERENCE MANUAL (Version 1.8.6). URL http://www.atelierb.eu/ressources/manrefb1.8.6.uk.pdf
16. Dijkstra, E.W.: A Discipline of Programming. Prentice-Hall (1976)
17. Eclipse: Eclipse Public License - v 1.0. URL http://www.eclipse.org/org/documents/epl-v10.html
18. Fischer, C.: CSP-OZ: A Combination of Object-Z and CSP (1997)
19. Fischer, C.: How to Combine Z with Process Algebra. In: ZUM, pp. 5–23 (1998)
20. Formal Systems (Europe) Ltd: FDR2 User Manual, fdr 2.94 edn. (2012)
21. Freitas, A., Cavalcanti, A.: Automatic Translation from Circus to Java. In: FM, pp. 115–130 (2006)
22. Freitas, L.: Model Checking Circus. Ph.D. thesis (2005)
23. Galloway, A., Stoddart, B.: An Operational Semantics for ZCCS. In: ICFEM, pp. 272– (1997)
24. Hoare, C., He, J.: Unifying theories of programming, vol. 14. Prentice Hall (1998)
25. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall (1985)
26. Jones, C.B.: Systematic software development using VDM (2. ed.). Prentice Hall International Series in Computer Science. Prentice Hall (1991)
27. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
28. Lazic, R.: A Semantic Study of Data Independence with Application to Model Checking. Ph.D. thesis (1999). URL http://citeseer.uark.edu:8080/citeseerx/showciting;jsessionid=344C8C9F9CE25F2A4FE75DDD6362C072?cid=96955&sort=recent
29. Leijen, D.: Division and Modulus for Computer Scientists (2001)
30. Leuschel, M., Butler, M.J.: Automatic Refinement Checking for B. In: ICFEM, pp. 345–359 (2005)
31. Marcel Oliveira Augusto Sampaio, P.A.R.R.A.C.J.W.: Compositional Analysis and Design of CML Models. COMPASS Deliverable D24.1 (2013)
32. Milner, R.: A Calculus of Communicating Systems, *Lecture Notes in Computer Science*, vol. 92. Springer (1980)
33. Morgan, C.C.: Programming from specifications, 2nd Edition. Prentice Hall International series in computer science. Prentice Hall (1994)
34. Mota, A., Sampaio, A.: Model-Checking CSP-Z. In: FASE, pp. 205–220 (1998)
35. Nogueira, S., Sampaio, A., Mota, A.: Test generation from state based use case models. Formal Aspects of Computing pp. 1–50 (2012). DOI 10.1007/s00165-012-0258-z. URL http://dx.doi.org/10.1007/s00165-012-0258-z
36. Oliveira, M., Cavalcanti, A., Woodcock, J.: Formal Development of Industrial-Scale Systems in Circus. ISSE **1**(2), 125–146 (2005)
37. Oliveira, M., Cavalcanti, A., Woodcock, J.: A Denotational Semantics for Circus. Electr. Notes Theor. Comput. Sci. **187**, 107–123 (2007)

38. Oliveira, M., Cavalcanti, A., Woodcock, J.: A UTP Semantics for Circus. Formal Asp. Comput. **21**(1-2), 3–32 (2009)
39. Oliveira, M., Sampaio, A., Conserva Filho, M.: Model-Checking Circus State-Rich Specifications. In: E. Albert, E. Sekerinski (eds.) Integrated Formal Methods, Lecture Notes in Computer Science, pp. 39–54. Springer International Publishing (2014). DOI 10.1007/978-3-319-10181-1_3. URL http://dx.doi.org/10.1007/978-3-319-10181-1_3
40. Oliveira, M.V.: Formal Derivation of State-Rich Reactive Programs using Circus. Ph.D. thesis, University of York (2005)
41. Plagge, D., Leuschel, M.: Validating Z Specifications Using the ProB Animator and Model Checker. In: J. Davies, J. Gibbons (eds.) Integrated Formal Methods, *Lecture Notes in Computer Science*, vol. 4591, pp. 480–500. Springer Berlin Heidelberg (2007). DOI 10.1007/978-3-540-73210-5_25. URL http://dx.doi.org/10.1007/978-3-540-73210-5_25
42. Roscoe, A.: Understanding Concurrent Systems, 1st edn. Springer-Verlag New York, Inc., New York, NY, USA (2010)
43. Roscoe, A.W., Hoare, C.A.R., Bird, R.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
44. Roscoe, A.W., Woodcock, J., Wulf, L.: Non-interference through Determinism. Journal of Computer Security **4**(1), 27–54 (1996)
45. Scattergood, B.: The Semantics and Implementation of Machine-Readable CSP. Ph.D. thesis, University of Oxford (1998)
46. Schneider, S., Treharne, H.: CSP theorems for communicating B machines. Formal Asp. Comput. **17**(4), 390–422 (2005)
47. Spivey, J.M.: Z Notation: A Reference Manual (2. ed.). Prentice Hall International Series in Computer Science. Prentice Hall (1992)
48. Welch, P.H.: Process Oriented Design for Java: Concurrency for All. In: PDPTA (2000)
49. Woodcock, J., Cavalcanti, A.: The Semantics of Circus. In: ZB, pp. 184–203 (2002)
50. Woodcock, J., Cavalcanti, A.: A Tutorial Introduction to Designs in Unifying Theories of Programming. In: IFM, pp. 40–66 (2004)
51. Woodcock, J., Cavalcanti, A., Freitas, L.: Operational Semantics for Model Checking Circus. In: FM, pp. 237–252 (2005)
52. Woodcock, J., Cavalcanti, A., Gaudel, M.C., Freitas, L.: Operational Semantics for Circus. Formal Aspects of Computing (2007). URL https://www.cs.york.ac.uk/ftpdir/pub/leo/utp/journal-pub/circus-operational-semantics.pdf