# Operational Semantics for *Circus Time*

**Kun Wei**

Department of Computer Science
University of York

*Technical Report*

November 2013

## Abstract

The *Circus* family of languages is a collection of process algebras based on a combination of Z, for data modelling, and CSP, for behavioural modelling. In this paper we develop a symbolic transition system for *Circus Time*, a timed version of *Circus*, whose soundness is proved with respect to a denotational semantics. Our contribution is credited not only to a combination of separate operational semantics for process algebra and for sequential programming in the UTP, but also to a full consideration of feasibility checking for *Circus Time* programs. We believe that this brand-new operational semantics can be easily applied to other state-rich process algebra.

# Contents

## 1 Introduction

An operational semantics of a programming language defines a set of possible individual rules that instruct a machine how to execute a program step by step. Therefore, it concentrates on the language constructs by transition relations, in contrast with a denotational semantics, which often offers an overall effect of the program by a set of observable variables, and with an algebraic semantics, which derives the equivalence of different processes from a set of algebraic laws. Traditionally, for sequential programming languages, we use a sequence of individual steps, which of them consists of a machine state and the remainder of a program, to handle a computation of the program; for process algebras like CSP [4, 7], we use a labelled transition system to describe the communication and concurrency between different components of a program. However, for state-rich process algebras like *Circus* [15, 1], which is a compound of abstract imperative programming and CSP, we need to combine the two kinds of operational semantics. And it is not so easy to achieve this combination.

*Circus* is a comprehensive combination of Z [14], CSP and Morgan's refinement calculus [6], so that it can define both data and behavioural aspects of a system. Over the years, *Circus* has developed into a family of languages for specification, programming and verification. The semantics of the *Circus* languages is based on Hoare and He's Unifying Theories of Programming (UTP) [5], in which they use the alphabetised relational calculus to describe and reason about a wide variety of programming paradigms. *Circus Time* [10, 11] is an extension of *Circus* with some time operators. The theory of *Circus Time* is a discrete time model, and the time operators are very similar to those in Timed CSP [9]. The denotational semantics of *Circus Time* has been well developed particularly in [11] where a pre/postcondition-style semantics is proposed so that it can support contract-based reasoning about models, and also simplify proof of *Circus Time* programs.

The operational semantics of *Circus* has been proposed in [16, 3], and the underlying automata theory and its properties have been formalised in [3] using the Z/Eves theorem prover [8]. The original work in [16] inspires a new symbolic transition system [12] for CML [13], which is a combination of VDM [2] and CSP for modelling systems of systems. The work for the operational semantics of CML is still ongoing, since, so far, it has considered an untimed model only. The novelty in [12] is to use loose constants to express the values of program variables. The loose constants must satisfy certain constraints, which are explicitly given within each transition rule. The collection of the loose constants and their constraints are carried with the execution of a program.

In the paper, based on the new idea in [12], we develop an operational semantics for *Circus Time*, which not only contains the evolution of time in transition rules, but also fully considers the feasibility both for CSP actions and data operations, because *Circus Time* programs can use infeasible actions in the action constructs. To understand the proof for feasibility checking in this paper, we assume the basic knowledge of CSP and the UTP. We briefly describe the *Circus Time* model and the related theories in Section 2. For the detailed introduction to *Circus Time* in the UTP, the reader is referred to [11]. In Section 3, we discuss the new symbolic transition system for *Circus Time*. Infeasible programs cannot be executed by a machine, and therefore, in Section 4, we present how to check the feasibility for a *Circus Time* program, and explicitly give the proviso in related transition rules. In Section 5, we focus on the transition rules of some key operators, such as prefix, external choice, parallel composition and so on, which basically include the essence of the operational semantics of *Circus Time*. Finally, we conclude the paper in Section 6.

## 2 *Circus Time* and UTP theories

The UTP uses the alphabetised relational calculus to supports refinement-based reasoning in the context of a variety of programming paradigms. In the UTP, a relation $P$ is a predicate with an alphabet $\alpha P$, composed of *undashed* variables $(a, b, ...)$ and *dashed* variables $(a', x', ...)$. The former, denoted as $in\alpha P$, stands for initial observations, and the latter, $out\alpha P$, for intermediate or final observations. A relation is called *homogeneous* if $out\alpha P = in\alpha P'$, where $in\alpha P'$ is obtained by dashing all the variables of $in\alpha P$. A *condition* has an empty output alphabet.

The program constructors in the theory of relations include sequential composition $(P\,;Q)$, conditional $(P \lhd b \rhd Q)$, assignment $(x := e)$, nondeterminism $(P \sqcap Q)$ and recursion $(\mu\, X \bullet C(X))$. The correctness of a program $P$ with respect to a specification $S$ is denoted by $S \sqsubseteq P$ ($P$ refines $S$), and is defined as $[P \Rightarrow S]$. Here, the square bracket is universal quantification over all variables in the alphabet. In other words, the correctness of $P$ is proved by establishing that every observation that satisfies $P$ must also satisfy $S$. Moreover, the set of relations with a particular alphabet is a complete lattice under the refinement ordering. Its bottom element is the weakest relation **true**, which models the program that behaves arbitrarily ($\mathbf{true} \sqsubseteq P$), and the top element is the strongest relation **false**, which behaves miraculously and satisfies any specification ($P \sqsubseteq \mathbf{false}$).

There are different theories in the UTP for representing a wide variety of programming paradigms. For example, the theory of designs is sufficient for describing sequential programs, and the theory of reactive processes is for describing communications between programs. The theory of CSP is vital for developing the theory of *Circus Time*.

## 2.1 Reactive designs

The theory of reactive designs is a combination between the theories of designs and reactive processes in the UTP. The reactive design theory provides a contract-based (pre and postconditions) semantics for the theory of CSP, which is traditionally built on the theory of reactive processes by imposing additional healthiness conditions.

A design in the UTP is a relation that can be expressed as a pre-postcondition pair in combination with boolean variables, called $ok$ and $ok'$. In designs, $ok$ records that the program has started, and $ok'$ records that it has terminated. If $P$ and $Q$ are predicates not containing $ok$ and $ok'$, a design with the precondition $P$ and the postcondition $Q$, written as $P \vdash Q$, is defined as

$$P \vdash Q \mathrel{\widehat=} ok \wedge P \Rightarrow ok' \wedge Q$$

which means that if a program starts in a state satisfying $P$, then it must terminate, and whenever it terminates, it must satisfy $Q$.

Healthiness conditions of a theory in the UTP are a collection of some fundamental laws that must be satisfied by relations belonging to the theory. These laws are expressed in terms of monotonic idempotent functions. A predicate is a design, if and only if, it satisfies **H1** and **H2**.

$$\mathbf{H1}(P) = ok \Rightarrow P \qquad \mathbf{H2}(P) = [P[\mathbf{false}/ok'] \Rightarrow P[\mathbf{true}/ok']]$$

The first healthiness condition means that observations of a predicate $P$ can only be made after the program has started. **H2** states that a design cannot require non-termination, since if $P$ is satisfied when $ok'$ is false, it must also be satisfied when $ok'$ is true.

A reactive process is a program whose behaviour may depend on interactions with its environment. To represent intermediate waiting states, a boolean variable $wait$ is introduced to the alphabet of a reactive process. Apart from $ok$, $ok'$, $wait$ and $wait'$, another two pairs of observational variables, $tr$ and $ref$, and their dashed counterparts, are introduced. The variable $tr$ records the events that have occurred until the last observation, and $tr'$ contains all the events including those since last observation. Similarly, $ref$ records the set of events that could be refused in the last observation, and $ref'$ records the set of events that may be refused currently. The reactive identity, $I\!I_{rea}$, is defined as $I\!I_{rea} \mathrel{\widehat=} (\neg\ ok \wedge tr \leq tr') \vee (ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref)$ which states that if its predecessor diverges ($\neg\ ok$), the extension of traces is the only guaranteed observation; otherwise ($ok'$), other variables keep unchanged.

Healthiness conditions of a theory in the UTP are a collection of some fundamental laws that must be satisfied by relations belonging to the theory. These laws are expressed in terms of monotonic idempotent functions. A reactive process must satisfy the following healthiness conditions:

$$\mathbf{R1}(P) = P \wedge tr \leq tr' \quad \mathbf{R2}(P(tr, tr')) = P(\langle\rangle, tr' - tr) \quad \mathbf{R3}(P) = I\!I_{rea} \lhd wait \rhd P$$

If a relation $P$ describes a reactive process, **R1** states that it never changes history. The second, **R2**, states that the history of the trace $tr$ has no influence on the behaviour of the process. The final, **R3**, requires that a process should leave the state unchanged ($I\!I_{rea}$) if it is waiting the termination of its predecessor ($wait = true$). A reactive process is a relation whose alphabet includes $ok$, $wait$, $tr$ and $ref$, and their dashed counterparts, and that satisfies the composition $\mathbf{R}$ where $\mathbf{R} \mathrel{\widehat=} \mathbf{R1} \circ \mathbf{R2} \circ \mathbf{R3}$. In other words, a process $P$ is a reactive process if, and only if, it is a fixed point of $\mathbf{R}$.

In the UTP, the theory of CSP is built by applying additional healthiness conditions to reactive processes. For example, a reactive process is also a CSP process if and only if, it satisfies the following healthiness conditions:

$$\mathbf{CSP1} \quad P = P \lor (\neg\, ok \land tr \le tr') \qquad \mathbf{CSP2} \quad P = P \,;\, J$$

where $J = (ok \Rightarrow ok') \land wait' = wait \land tr' = tr \land ref' = ref$. The first healthiness condition requires that, in whatever situation, the trace can only be increased. The second one means that $P$ cannot require non-termination, so that it is always possible to terminate.

A CSP process can also be obtained by applying the healthiness condition $\mathbf{R}$ to a design. This follows from the theorem in [5], that, for every CSP process $P$, $P = \mathbf{R}(\neg\, P^f_f \vdash P^t_f)$, where $P^a_b$ is an abbreviation of $P[a, b/ok', wait]$, and it is often used in this paper. This theorem gives a new style of specification for CSP processes in which a design describes the behaviour when its predecessor has terminated and not diverged, and the other situations of its behaviour are left to $\mathbf{R}$. The importance of this reactive design semantics is that it exposes the pre-postcondition semantics so as to support contract-based reasoning about models.

## 2.2 *Circus Time*

In the theory of *Circus Time*, a CSP action is described as an alphabetised predicate whose observational variables include $ok$, $wait$, $tr$, $ref$, $state$ and their dashed counterparts. Here, $ok$, $ok'$, $wait$ and $wait'$ are the same variables used in the theory of reactive processes. The traces, $tr$ and $tr'$, are defined to be non-empty sequences ($\mathrm{seq}_1(\mathrm{seq}\ Event)$), and each element in the trace represents a sequence of events that have occurred over one time unit. Also, $ref$ and $ref'$ are non-empty sequences ($\mathrm{seq}_1(\mathbb{P}\, Event)$) where each element is a refusal at the end of a time unit. Thus, time is actually hidden in the length of traces. In addition, $state$ and $state'$ ($N \nrightarrow Value$) records a set of local variables and their values. $N$ is the set of valid names of these variables.

We explain the details of the notation at the points where they are firstly used. For sequences, we use $head$, $tail$, $front$, $last$, $\#$(length), $^\frown$(concatenation) and $^\frown/$(flattening). An expanding relation between traces is defined as

$$tr \preccurlyeq tr' \mathrel{\widehat=} front(tr) \le tr' \land last(tr) \le tr'(\#tr)$$

which, for example, states that $\langle\langle a\rangle, \langle b\rangle\rangle$ is expanding $\langle\langle a\rangle, \langle b, c\rangle\rangle$.

An action in *Circus Time* must satisfy the healthiness conditions, $\mathbf{R1_{ct}}$-$\mathbf{R3_{ct}}$ and $\mathbf{CSP1_{ct}}$-$\mathbf{CSP5_{ct}}$. These healthiness conditions have similar meanings to those in the CSP theory, but are changed to accommodate discrete time. For the sake of a simpler proof, we focus on the healthiness conditions, $\mathbf{R1_{ct}}$ and $\mathbf{R3_{ct}}$, since the properties including other healthiness conditions are usually straightforward to be proven. A detailed introduction to other healthiness conditions can be found in [11].

$$\mathbf{R1_{ct}}(X) \mathrel{\widehat=} X \land RT \qquad \mathbf{R3_{ct}}(X) \mathrel{\widehat=} I\!\!I_{ct} \lhd wait \rhd X$$

where the predicate $RT$, the difference of two traces ($diff$), the relational identity ($I\!\!I$) and the timed reactive identity $I\!\!I_{ct}$ are given as

$$RT \mathrel{\widehat=} tr \preccurlyeq tr' \land front(ref) \le ref' \land \#diff(tr', tr) = \#(ref' - front(ref)) \tag{1}$$

$$diff(tr', tr) \mathrel{\widehat=} \langle tr'(\#tr) - last(tr)\rangle ^\frown tail(tr' - front(tr)) \tag{2}$$

$$I\!\!I \mathrel{\widehat=} \big(\ ok' = ok \land tr' = tr \land ref' = ref \land wait' = wait \land state' = state\ \big) \tag{3}$$

$$I\!\!I_{ct} \mathrel{\widehat=} (\neg\, ok \land RT) \lor (ok' \land I\!\!I) \tag{4}$$

Note that we impose a restriction, $\#diff(tr', tr) = \#(ref' - front(ref))$, to ensure that there is one refusal for each time unit. However, we are usually not interested in the refusals of the last time unit after an action terminates. Therefore, we use $front(ref) \le ref'$ in $\mathbf{R1_{ct}}$, instead of $ref \le ref'$. In addition, we have proved the following theorem in [11] that any action can be expressed as a reactive design

**Theorem 1.** *For any Circus Time action $P$, $P = \mathbf{R_{ct}}(\neg\, P^f_f \vdash P^t_f)$*

## 3 Symbolic transitions in *Circus Time*

An operational semantics of a program is a computation that describes how the program can be executed by machine. Usually, the computation is considered a sequence of individual steps, which takes the machine from one state to another close one. These steps are defined by a transition relation between individual machine states $(\mathsf{s},\mathsf{P})$. Here, $\mathsf{s}$ is a text, assigning constants to all program variables of the alphabet, and $\mathsf{P}$ is a program text, representing the rest of the program that has not been executed. If $\mathsf{Skip}$ is allowed to denote program termination, the state $(\mathsf{s}, \mathsf{Skip})$ is the last state of any computation, and $\mathtt{s}$ defines the final values of the program variables.

The transition relation relates *syntactic* objects rather than *semantic* ones. However, the connection between the syntactic and semantic objects allows us to verify the soundness of the transition relation by means of the denotational semantics. To distinguish the syntactic texts and their meanings in the operational semantics, we write $\mathtt{s}$ to be the text of the assignment $s$, and $\mathtt{P}$ to be text of the predicate $P$. The definition of the transition relation is represented by the semantic meaning of the machine states and the notion of refinement ($\sqsubseteq$) as follows.

$$(\mathsf{s},\mathsf{P}) \rightarrow (\mathsf{t},\mathsf{Q}) \mathrel{\widehat=} (s;\, P) \sqsubseteq (t;\, Q)$$

Here, $(s;\ P) \sqsubseteq (t;\ Q)$ means that an implementation of the program $(t;\ Q)$ is an improvement of executing the program $(s; P)$. That is, the after-state $(\mathtt{t},\mathtt{Q})$ is one of the next possible states since the state $(\mathtt{s},\mathtt{P})$ before the transition.

This kind of transition relations are sufficient for describing the operational semantics of the nondeterministic sequential programming languages. However, to describe the operational semantics of pure CSP, it is extended by requiring labelled transitions, but no need for representing imperative state. In the UTP, two kinds of transitions are identified as follows.

$$\mathsf{P} \rightarrow \mathsf{Q} \mathrel{\widehat=} P \sqsubseteq Q$$
$$\mathsf{P} \xrightarrow{\mathsf{a}} \mathsf{Q} \mathrel{\widehat=} P \sqsubseteq ((a \rightarrow Q) \Box P)$$

The first transition is for describing internal progress of a process, in which $P \sqsubseteq Q$ means that the result of implementing $\mathsf{Q}$ is equally or more deterministic than the original. The second transition is the action transition, representing that $a$ is a possible initial event for $P$, and $Q$ is the subsequent behaviour. Note that the action transition cannot be simply defined by $P \sqsubseteq a \rightarrow Q$, because the initial observation of $P$ may be constrained by other components other than $a \rightarrow Q$. For example, clearly, $a \rightarrow \mathit{Skip}$ cannot refine $a \rightarrow \mathit{Skip} \Box b \rightarrow \mathit{Skip}$ because of the unrefined initial refusal sets. Similarly, $a \rightarrow \mathit{Skip}$ cannot refine $(a \rightarrow \mathit{Skip} \Box b \rightarrow \mathit{Skip}) \setminus \{b\}$ because $P$ may resolve the internal non-determinism so that it even prevents $a$ from happening.

We use the similar symbolic transition system in CML for concisely expressing the multiple choices of next transitions, such as the communication of channels, nondeterministic delay and so on. For example, in *Circus Time*, we can define $\mathsf{channel\ ch}{:}\{0..9\}$ to represent that any number between 0 and 9 can be passed through channel $\mathsf{ch}$. We restrict the transitions of the action $ch?x \rightarrow P(x)$ in *Circus Time* to one symbolic transition as follows, rather than enumerate it by ten possible transitions.

$$ch?x \rightarrow P(x) \xrightarrow{ch.w_0} P(w_0) \;\text{ for loose constant } w_0 \in \{n \mid n \geq 0 \wedge \leq 9\}$$

To represent the operational semantics in *Circus Time*, a configuration (or a state) is identified as a triple $(\mathsf{c},\mathsf{s},\mathsf{A})$, where

- $\mathsf{c}$ is a text, describing constraints on loose constants.
- $\mathsf{s}$ is a text, describing an assignment for all program variables including state components, input and local variables and parameters, but not for any observational variable.
- $\mathsf{A}$ is a text, describing the rest of a *Circus Time* action.

Because simply considering $\mathsf{s}$ the command of storing the values only for the program variables, we need to add the observational variables and their healthiness conditions, when composing the semantics of $\mathsf{s}$ with its sequential actions. We lift the syntactic assignment $\mathsf{s}$ (or relational assignment) to a reactive design as follows:

$$\mathit{lift}(s)_R \mathrel{\widehat=} \mathbf{R_{ct}}(\mathbf{true} \vdash s \wedge tr' = tr \wedge \neg\, wait') \qquad (5)$$

where $s$ is added into the postcondition with $tr' = tr \land \neg\, wait'$ that also means that this assignment does not take any time and terminates immediately. The precondition **true** states that this reactive design never diverges and the healthiness conditions $\mathbf{R_{ct}}$ describes how it interacts with its predecessors.

In *Circus Time*, we adopt the idea of transitions for Timed CSP to separate time and events. That is, events are considered instantaneous, whereas *evolution* transitions are used to describe the passage of time. Thus, timed transition systems are allowed combinations of event and evolution transitions, and the constraints on them. In addition, the separation of time and events allows most transitions rules for untimed CSP to be valid in the timed system.

## 3.1 Event transitions

The execution of events are understood as not taking any time, no matter these events are visible or invisible. The transition corresponding to the communication of channels are defined as follows.

**Definition 1.**

$$(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{ch.w_0}} (\mathsf{c}_2, \mathsf{s}_2, \mathsf{A}_2) \mathrel{\widehat{=}} \forall\, w \bullet c_1 \land c_2 \Rightarrow \mathit{lift}(s_1); A_1 \sqsubseteq (\mathit{lift}(s_2); ch.w_0 \rightarrow A_2) \;\Box\; (\mathit{lift}(s_1); A_1)$$

Here, $w$ is a vector of loose constants including $w_0$, $c_1$ is the constraint on $w$ before the transition, and $c_2$ is the constraint after the transition. Note that loose constants are not subject to refinement because $c_1$ and $c_2$ should always be satisfied. If we choose values just to fit ourself, the proved transitions can lose important behaviours.

To ensure the occurrence of $ch.w_0$ is instant, we need to further explore Definition 1. The action transition in CSP is defined as $P \sqsubseteq (a \rightarrow Q) \;\Box\; P$, which in turn can be rewritten to $P = (a \rightarrow Q \;\Box\; e \rightarrow P) \setminus \{e\})$ with $e \notin \alpha P$. In CSP, the hidden event that is first enabled must be chosen when a process is resolving a choice of different events. That is, the occurrence of $a$ must be immediately, otherwise it will be prevented from happening by the hidden event. Similarly, for Definition 1 in *Circus Time*, we have the same transformation for the refinement relation, $\mathit{lift}(s_1); A_1 = ((\mathit{lift}(s_2); ch.w_0 \rightarrow A_2) \;\Box\; (\mathit{lift}(s_1); e \rightarrow A_1)) \setminus \{e\}$. Since $\mathit{lift}(s_1)$ and $\mathit{lift}(s_2)$ do not take any time as their definitions, $ch.w_0$ must happen immediately, which is consistent with the assumption of instant events.

The labelled transition of an invisible action in *Circus Time* is further simplified by comparison with Definition 2.

**Definition 2.**

$$(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\tau} (\mathsf{c}_2, \mathsf{s}_2, \mathsf{A}_2) \mathrel{\widehat{=}} \forall\, w \bullet c_1 \land c_2 \Rightarrow (\mathit{lift}(s_1); A_1) \sqsubseteq (\mathit{lift}(s_2); A_2)$$

This definition states that, if the vector of the loose constants $w$ is satisfied by the constraints $c_1$ and $c_2$, $\mathit{lift}_R(s_2); A_2$ is a refinement of $\mathit{lift}_R(s_1); A_1$. In *Circus Time*, invisible actions may involve the execution of internal events, any assignment on program variables, and some operators like internal choice, timeout and so on.

## 3.2 Evolution transitions

The passage of time can be described in the similar symbolic approach.

**Definition 3.**

$$(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{d}} (\mathsf{c}_2, \mathsf{s}_2, \mathsf{A}_2) \mathrel{\widehat{=}} \forall\, w \bullet c_1 \land c_2 \Rightarrow \mathit{lift}(s_1); A_1 \sqsubseteq (\mathit{Wait}\;\; d; \mathit{lift}(s_2); A_2)$$

This definition states that, if $c_1$ and $c_2$ are satisfied for $w$, where $d$ is a loose constant, $\mathit{Wait}\;\; d; \mathit{lift}(s_2); A_2$ is one of the subsequent behaviours of $\mathit{lift}(s_1); A_1$.

In the semantics of *Circus Time*, we request that $d$ is simply an integer, because the definition of the delay action can make meaningless delays like negative integers behave like *Miracle*. However, the operational semantics of *Circus Time* re strictly requests that $d$ can only be any positive integer.

## 4 Feasibility

In the theory of relations, **false** is the top element in the refinement ordering since it can refine any other element. Usually, the top element in a theory is called *Miracle*. Obviously, **false** is infeasible or never be implemented, because it can not give rise to any observation. Infeasible programs should never be executed by a machine, even though they may be so useful as a mathematical abstraction in reasoning about programs.

In the theory of designs, which is sufficient for describing sequential programming languages, the healthiness condition **H4**, $P$; **true** = **true**, is a feasibility condition that no design can be implemented if it fails to satisfy **H4**. In fact, **H4** can also be expressed as $[\exists\, ok', v' \bullet P]$ where $v$ is a vector of the program variables. That is, **H4** states, if the precondition of a design is satisfied, the design is required to terminate and deliver final values for the program variables, and these values must satisfy the postcondition. The *Miracle* of the theory of designs is **true** ⊢ **false**, which, admittedly, is not **H4** healthy.

$$
\begin{aligned}
&[\exists\, ok', v' \bullet \mathbf{true} \vdash \mathbf{false}] && \text{[design]}\\
=&[\exists\, ok', v' \bullet ok \wedge \mathbf{true} \Rightarrow ok' \wedge \mathbf{false}] && \text{[propositional calculus]}\\
=&[\exists\, ok', v' \bullet \neg\, ok] && \text{[predicate calculus]}\\
=&[\neg\, ok] && \text{[case split]}\\
=&\neg\, \mathbf{true} \wedge \neg\, \mathbf{false} && \text{[propositional calculus]}\\
=&\ \mathbf{false}
\end{aligned}
$$

In the theory of *Circus Time* (reactive designs), we use the same healthiness condition **H4** to identify infeasible actions, and thereby ensure right transitions rules for all operators. Before we give an intuitive perception of how **H4** identifies infeasible actions in *Circus Time*, we prove some auxiliary lemmas. For example, Lemma 1 states that a divergence will lose $\mathbf{R1_{ct}}$ through a composition with **true**, and Lemma 2 shows that the reactive design $I\!I_{ct}$ satisfies the zero law.

**Lemma 1.** $\mathbf{R1_{ct}}(\neg\, ok)$; **true** $= \neg\, ok$

*Proof.*

$$
\begin{aligned}
&\mathbf{R1_{ct}}(\neg\, ok);\ \mathbf{true} && [\mathbf{R1_{ct}}]\\
=&\ (\neg\, ok \wedge RT);\ \mathbf{true} && [RT(1)]\\
=&\ (\neg\, ok \wedge tr \preccurlyeq tr' \wedge front(ref) \leq ref' \wedge \#diff(tr', tr) = \#(ref' - front(ref)));\ \mathbf{true}\\
& && \text{[relational composition]}\\
=&\ \exists\, ok_0, wait_0, tr_0, ref_0, state_0 \bullet \left(\begin{array}{c} \neg\, ok \wedge tr \preccurlyeq tr_0 \wedge front(ref) \leq ref_0 \wedge \\ \#diff(tr_0, tr) = \#(ref_0 - front(ref)) \end{array}\right)\\
& && [\text{Let } tr_0 = tr,\ wait_0 = wait, tr_0 = tr, ref_0 = ref \text{ and } state_0 = state]\\
=&\ \neg\, ok \wedge tr \preccurlyeq tr \wedge front(ref) \leq ref \wedge \#diff(tr, tr) = \#(ref - front(ref))\\
& && \text{[properties of sequences and propositional calculus]}\\
=&\ \neg\, ok
\end{aligned}
$$

□

**Lemma 2.** $I\!I_{ct}$; **true** = **true**

*Proof.*

$$\begin{aligned}
&\mathit{II}_{ct}; \mathbf{true} && [\mathit{II}_{ct}(4)]\\
&= ((\neg\, ok \wedge RT) \vee (ok' \wedge \mathit{II})); \mathbf{true} && [\text{relational calculus}]\\
&= ((\neg\, ok \wedge RT); \mathbf{true}) \vee ((ok' \wedge \mathit{II}); \mathbf{true}) && [\mathit{II}(3)]\\
&= ((\neg\, ok \wedge RT); \mathbf{true}) \vee && [\text{relational composition}]\\
&\quad ((ok' \wedge ok' = ok \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge state' = state); \mathbf{true})\\
&= ((\neg\, ok \wedge RT); \mathbf{true}) \vee\\
&\quad \exists\, ok_0, wait_0, tr_0, ref_0, state_0 \bullet \begin{pmatrix} ok_0 \wedge ok_0 = ok \wedge wait_0 = wait \wedge tr_0 = tr \\ \wedge\, ref_0 = ref \wedge state_0 = state \end{pmatrix}\\
& && [\text{Let } ok_0 = ok, wait_0 = wait, tr_0 = tr, ref_0 = ref \text{ and } state_0 = state]\\
&= ((\neg\, ok \wedge RT); \mathbf{true}) \vee ok && [\text{Lemma 1}]\\
&= \neg\, ok \vee ok && [\text{propositional calculus}]\\
&= \mathbf{true}
\end{aligned}$$

□

A reactive design is a $\mathbf{R_{ct}}$-healthy design, expressed as $\mathbf{R_{ct}}(\neg\, P_f^f \vdash P_f^t)$, in which the design describes the program's behaviour when its predecessor terminates, $\mathbf{R3_{ct}}$ describes how the program behaves when its predecessor has not finished, and $\mathbf{R1_{ct}}$ enables the property of $tr, tr', ref$ and $ref'$. Intuitively, the feasibility of the reactive design is irrelevant to $\mathbf{R3_{ct}}$, and results from $\mathbf{R1_{ct}}(\neg\, P_f^f \vdash P_f^t)$. Therefore, we need to deal with $\mathbf{R1_{ct}}$-healthy designs only when identifying feasibility.

**Definition 4.** ($\mathbf{R1_{ct}}$-designs) $P \models Q \mathrel{\widehat=} \mathbf{R1_{ct}}(P \vdash Q)$

The closure property is still valid for the $\mathbf{R1_{ct}}$-designs, and, here, we only give the closure law for sequential composition.

**Law 1.** (closure-;-$\models$)

$$(P_1 \models Q_1); (P_2 \models Q_2) = \begin{pmatrix} \neg(\mathbf{R1_{ct}}(\neg P_1); \mathbf{R1_{ct}}(\mathbf{true})) \wedge \neg(\mathbf{R1_{ct}}(Q_1); \mathbf{R1_{ct}}(\neg P_2)) \\ \models \mathbf{R1_{ct}}(Q_1); \mathbf{R1_{ct}}(Q_2) \end{pmatrix}$$

The proof of Law 1 is straightforward and can be found in [11].

As a result, the feasibility of an action in *Circus Time* can be justified by the following theorem.

**Theorem 2.** *(*Feasibility*)*
*A Circus Time action satisfies* **H4***, if and only if,* $(\neg\, P_f^f \models P_f^t); \mathbf{true} = \mathbf{true}$.

*Proof.*

$$\begin{aligned}
&P; \mathbf{true} && [\text{Theorem 1}]\\
&= \mathbf{R3_{ct}} \circ \mathbf{R1_{ct}}(\neg\, P_f^f \vdash P_f^t); \mathbf{true} && [\mathbf{R3_{ct}}]\\
&= (\mathit{II}_{ct} \lhd wait \rhd \mathbf{R1_{ct}}(\neg\, P_f^f \vdash P_f^t)); \mathbf{true} && [\text{relational calculus}]\\
&= ((\mathit{II}_{ct} \wedge wait); \mathbf{true}) \vee ((\neg\, wait \wedge \mathbf{R1_{ct}}(\neg\, P_f^f \vdash P_f^t)); \mathbf{true}) && [\text{rel. cal.}]\\
&= (wait \wedge (\mathit{II}_{ct}; \mathbf{true})) \vee (\neg\, wait \wedge (\mathbf{R1_{ct}}(\neg\, P_f^f \vdash P_f^t); \mathbf{true})) && [\text{Lemma 2}]\\
&= (wait \wedge \mathbf{true}) \vee (\neg\, wait \wedge (\mathbf{R1_{ct}}(\neg\, P_f^f \vdash P_f^t); \mathbf{true})) && [\text{propositional calculus}]\\
&= wait \vee (\neg\, wait \wedge (\mathbf{R1_{ct}}(\neg\, P_f^f \vdash P_f^t); \mathbf{true})) && [\text{assumption}]\\
&= wait \vee (\neg\, wait \wedge \mathbf{true}) && [\text{propositional calculus}]\\
&= \mathbf{true}
\end{aligned}$$

□

As a matter of fact, we can prove *Miracle*, $\mathbf{R_{ct}}(\mathbf{true} \vdash \mathbf{false})$, in *Circus Time* to be infeasible with ease by means of Theorem 2.

$$
\begin{aligned}
& [(\neg\ Miracle^f_f \models Miracle^t_f); \mathbf{true}] && [\models \text{ and } Miracle] \\
=& [\mathbf{R1_{ct}}(\mathbf{true} \vdash \mathbf{false}); \mathbf{true}] && [\text{design}] \\
=& [\mathbf{R1_{ct}}(\neg\ ok); \mathbf{true}] && [\text{Lemma 1}] \\
=& [\neg\ ok] && [\text{predicate calculus}] \\
=& \neg\ \mathbf{true} \wedge \neg\ \mathbf{false} && [\text{propositional calculus}] \\
=& \mathbf{false}
\end{aligned}
$$

We present a Plotkin-style structured operational semantics for *Circus Time*. Each transition rule is written as follows, in which $P$ is a collection of antecedents.

$$\frac{P}{(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_2, \mathsf{s}_2, \mathsf{A}_2)}$$

In order to reach a conclusion of each rule, we require that all rules must be feasible, providing that $\forall\, w \bullet P \wedge c_1 \wedge c_2$. To verify the feasibility, we have to lift $\mathsf{s}$ to be a $\mathbf{R1_{ct}}$-healthy design.

$$lift(s)_D \mathrel{\widehat=} \mathbf{true} \models s \wedge tr' = tr \tag{6}$$

The *Circus Time* model adopts Theorem 2 to check for feasibility, but with a subtle change. The alphabet of *Circus Time* contains not only $ok, wait, tr, ref$ and their dashed counterparts, which are the same as the alphabet of CSP, but also $state$ and $state'$ to store the values of program variables. In other words, Theorem 2 is used to identify the infeasible behaviours ($tr'$ and $ref'$), which are not influenced by its predecessor in CSP. However, in *Circus Time*, considering a configuration $(\mathsf{c}, \mathsf{s}, \mathsf{A})$, checking the feasibility of $\mathsf{A}$ only is not sufficient, because the inconsistency between the state ($\mathsf{s}$) and the program ($\mathsf{A}$) can also result in the infeasible program. For comprehensively checking feasibility a *Circus Time* program, we use the following theorem.

**Theorem 3.** *A Circus Time program is feasible, if and only if,* $lift_D(s); (\neg\ A^f_f \models A^t_f); \mathbf{true} = \mathbf{true}$.

The right side of Theorem 3 may be easier to understand when it is split into two parts, each of which can imply it alternatively.

**Theorem 4.**

$lift_D(s); (\neg\ A^f_f \models A^t_f); \mathbf{true} = \mathbf{true}$ provided that

(1).$(s; \mathbf{R1_{ct}}(A^f_f); \mathbf{true}) = \mathbf{true}$, or

(2).$\neg\ (s; \mathbf{R1_{ct}}(A^f_f); \mathbf{true}) = \mathbf{true}$ and $(s; \mathbf{R1_{ct}}(A^t_f); \mathbf{true}) = \mathbf{true}$

*Proof.*

$$\begin{array}{ll}
& \mathit{lift}_D(s); (\neg\ A^f_f \models A^t_f); \mathbf{true} & [\mathit{lift}_D(6)] \\
= & (\mathbf{true} \models s \wedge tr' = tr); (\neg\ A^f_f \models A^t_f); \mathbf{true} & [\text{Law 1}] \\
= & \left(\begin{array}{l} \neg(\mathbf{R1_{ct}}(\neg\mathbf{true}); \mathbf{R1_{ct}}(\mathbf{true})) \wedge \neg(\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(\neg\neg\ A^f_f)) \\ \models \mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^t_f) \end{array}\right); \mathbf{true} & \\
& & [\text{relational calculus}] \\
= & (\neg(\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^f_f)) \models \mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^t_f)); \mathbf{true} & [\models] \\
= & \mathbf{R1_{ct}}\left(\begin{array}{l} \neg\ ok \vee (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^f_f)) \vee \\ ok' \wedge (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^t_f)) \end{array}\right); \mathbf{true} & \\
& & [\mathbf{R1_{ct}}\text{-idempotent and relational calculus}] \\
= & \left(\begin{array}{l} (\mathbf{R1_{ct}}(\neg\ ok); \mathbf{true}) \vee (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^f_f); \mathbf{true}) \vee \\ (ok' \wedge (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^t_f))); \mathbf{true} \end{array}\right) & [\text{Lemma 1}] \\
= & \left(\begin{array}{l} \neg\ ok \vee (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^f_f); \mathbf{true}) \vee \\ (ok' \wedge (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A^t_f))); \mathbf{true} \end{array}\right) & \\
& & [s \text{ and } tr' = tr \text{ are } \mathbf{R1_{ct}} \text{ and relational calculus}] \\
= & \neg\ ok \vee (s; \mathbf{R1_{ct}}(A^f_f); \mathbf{true}) \vee (s; \mathbf{R1_{ct}}(A^t_f); \mathbf{true}) & [\text{assumption}] \\
= & \mathbf{true} &
\end{array}$$

□

The first proviso in Theorem 4 states that, although A is feasible, its precondition cannot be satisfied after sequentially following s. For this case, the postcondition of A is arbitrary. The second proviso of the above theorem states that the precondition of A is satisfied and then the postcondition can provide a final observation. The two provisos highlight the origin of infeasibility that the postcondition cannot provide the final observation even if the precondition has been satisfied.

In *Circus Time*, we treat every action as a reactive design. For example, we transform a specification statement, $x : [p, Q]$, into a reactive design.

**Definition 5.** $x : [p, Q] \mathrel{\widehat=} \mathbf{R_{ct}}(p \vdash Q \wedge tr' = tr \wedge \neg\ wait')$

where $p$ is the precondition for describing the initial state of the program variables ($p$ is a condition), and $Q$ is a postcondition for describing the final state, and $x$ is a frame for listing the variables whose values may change.

We, here, use Lemma 3 and 4 to calculate the antecedents for a feasible specification statement.

**Lemma 3.** $s;\ \mathbf{R1_{ct}}((x : [p, Q])^f_f);\ \mathbf{true} = \neg\ (s;\ p)$

*Proof.*

$$\begin{array}{ll}
& s;\ \mathbf{R1_{ct}}((x : [p, Q])^f_f);\ \mathbf{true} & [\text{Definition 5}] \\
= & s;\ \mathbf{R1_{ct}}(\neg\ p);\ \mathbf{true} & [p \text{ is for program variables only}] \\
= & s;\ \neg\ p;\ \mathbf{true} & [p \text{ is a condition}] \\
= & s;\ \neg\ p & [s \text{ is such as } v := e \text{ and property of assignment}] \\
= & (\neg\ p)[e/v] & [\text{property of substitution}] \\
= & \neg\ (p[e/v]) & [\text{property of assignment}] \\
= & \neg\ (s;\ p) &
\end{array}$$

□

**Lemma 4.** $s; \mathbf{R1_{ct}}((x : [p, Q])^t_f); \mathbf{true} = \exists\, v' \bullet s;\ Q$

*Proof.*

$$\begin{array}{ll}
& s; \mathbf{R1_{ct}}((x : [p, Q])^t_f); \mathbf{true} & [\text{Definition 5}] \\
= & s; \mathbf{R1_{ct}}(Q \wedge tr' = tr \wedge \neg\ wait'); \mathbf{true} & [Q \text{ and } tr' = tr \text{ are } \mathbf{R1_{ct}}] \\
= & s;\ (Q \wedge tr' = tr \wedge \neg\ wait'); \mathbf{true} & [\text{relational calculus}] \\
= & s;\ Q; \mathbf{true} & [\text{relational calculus}] \\
= & \exists\, v' \bullet s;\ Q &
\end{array}$$

□

As a result, the operational semantics for a specification is given as follows.

$$\frac{\neg\,(s;\ p)}{(\mathsf{c},\mathsf{s},\mathsf{x}\!:[\mathsf{p},\mathsf{Q}]) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathsf{Chaos})} \qquad \frac{s;\ p \quad \exists\, v' \bullet s;\ Q \quad v = out\alpha(s)}{(\mathsf{c},\mathsf{s},\mathsf{x}\!:[\mathsf{p},\mathsf{Q}]) \xrightarrow{\tau} (\mathsf{c}\ \mathbf{and}\ (\mathsf{s};\mathsf{Q}[\mathsf{w_0}/\mathsf{v'}]), \mathsf{s};\mathsf{v}:=\mathsf{w_0}, \mathsf{Skip})}$$

The left transition rule states that, if $p$ cannot be satisfied by the current state, or $s;\,p$ is **false**, it transits to a simple divergence. The right rule states that, if the precondition is satisfied and the commitment on $s$ is feasible, this action is completed and becomes *Skip*. Because $Q$ may be nondeterministic, a loose constant ($w_0$) is used as the target of the later assignment. Of course, $w_0$ is constrained by $s;\,Q[w_0/v']$.

There is no transition rule for any infeasible action. For example, an infeasible specification arises if $p$ is satisfied but $Q$ is not, which cannot make any antecedent of the rules for the specification satisfied. If a machine encounters an infeasible program, it is stuck immediately even if time is not allowed to pass. This state is called *timestop*. Apart from infeasible specifications, timestops can also be caused by the actions involving *Miracle* and deadline operators in *Circus Time*. In other words, in the operational semantics of *Circus Time*, the infeasible variables are excluded by Theorem **??**, and the infeasible actions (CSP) are considered timestops.

## 5 Transition rules for *Circus Time*

### 5.1 Syntax devices

We introduce some syntax devices to extend BNF of *Circus* because the definition of the operational semantics requires some extra constructs. First, the syntax device, **and**, is introduced in the constraints $\mathsf{c}$ for the loose constants $w$ in a configuration, simply denoting a new addition of the constraint in $\mathsf{c}$. Second, we define $\mathbf{let}\,\mathsf{x} \bullet \mathsf{A}$ in the program text to mark the scope of the variable. This device is often used in the transition rules for the declaration of the program variables. Third, we use $\mathbf{loc}\,(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}$ to localise the constraints on the loose constants and the state condition in the action $\mathsf{A}$ that is one of operands in a compounded action. For some cases, we localise one of $\mathsf{c}$ and $\mathsf{s}$ only, which means that the absent one is the one of the configuration. The device **loc** is used for the operators like external choice, parallel composition, timeout and so on.

In a valid configuration, the state condition $\mathsf{s}$ and the action $\mathsf{A}$ have the same alphabet. Therefore the alphabet of **let** and **loc** are defined as

$$\alpha(\mathbf{let}\,\mathsf{x} \bullet \mathsf{A}) = \alpha\mathsf{A}, \text{with } \{x, x'\} \subseteq \alpha\mathsf{A}$$
$$\alpha(\mathbf{loc}\,(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}) = \alpha\mathsf{A}$$

In addition, we use $\Sigma$ to denote the alphabet of all communications, and $\Sigma^{+\tau} = \Sigma \cup \{\tau\}$. We also define a function $FV(e)$ that returns a set of the free variables in $e$, and a function $Label(A)$ to capture the initial events of $A$.

### 5.2 Transition rules for *Circus Time* actions

In the section, we give the operational semantics for all operators with CSP actions. In a *Circus Time* process, the state variables are invisible to other processes. Therefore, we consider that the transition system for *Circus Time* processes is similar to that of pure CSP processes. Therefore, we expect that the novelty of this work comes from the symbolic transition system for the computation, communication and concurrency between the *Circus Time* actions.

#### 5.2.1 Primitive actions

The action *Stop* represents deadlock but allows time to elapse. Since the program variables becomes arbitrary at any deadlock, we make $\mathsf{s}$ unchanged for the after state, so that no one can access the arbitrary values.

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{Stop}) \xrightarrow{\mathsf{w_0}} (\mathsf{c}\ \mathbf{and}\ \mathsf{w_0} \in \mathbb{Z}^+, \mathsf{s}, \mathsf{Stop})} \tag{7}$$

Here, $w_0$ is a loose constant that can be any positive integer.

The action *Chaos* [1] represents a chaotic action that can only silently transit to another chaotic action. Similar to the rule for *Stop*, we also allow $\mathsf{s}$ to be unchanged during the transition.

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{Chaos}) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathsf{Chaos})} \tag{8}$$

For the actions *Skip* and *Miracle*, we simple give no transition rule.

### 5.2.2 Variable declaration, assignment and specification

Commands in *Circus Time* are defined as variable declaration and variable assignment. To describe a variable declaration, we use a syntactic device, $\mathbf{let}\,\mathsf{x} \bullet \mathsf{A}$, in the program text to mark the scope of the variable. The first transition rule for the variable declaration is given as

$$\frac{T \neq \emptyset \quad x \notin \alpha s}{(\mathsf{c},\mathsf{s},\mathbf{var}\,\mathsf{x}\!:\!\mathsf{T} \bullet \mathsf{A}) \xrightarrow{\tau} (\mathsf{c}\ \mathbf{and}\ \mathsf{w}_0 \in \mathsf{T}, \mathsf{s};\mathbf{var}\,\mathsf{x} := \mathsf{w}_0, \mathbf{let}\,\mathsf{x} \bullet \mathsf{A})} \tag{9}$$

where the antecedents require that the elements contained by the type $T$ cannot be empty, and that the variable cannot be redeclared within its own scope. When this declaration transits to a new state, we select a fresh loose constant $w_0$ for the initialisation of the variable in the new state ($\mathsf{s};\mathbf{var}\,\mathsf{x} := \mathsf{w}_0$). We require that $w_0$ can be any element of $T$. In addition, we give two transition rules to deal with the program text with **let**.

$$\frac{(\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_2,\mathsf{s}_s,\mathsf{A}_2)}{(\mathsf{c}_1,\mathsf{s}_1,\mathbf{let}\,\mathsf{x}\bullet\mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_2,\mathsf{s}_2,\mathbf{let}\,\mathsf{x}\bullet\mathsf{A}_2)} \tag{10}$$

$$\frac{}{(\mathsf{c},\mathsf{s},\mathbf{let}\,\mathsf{x}\bullet\mathsf{Skip}) \xrightarrow{\tau} (\mathsf{c},\mathsf{s};\mathbf{end}\,\mathsf{x},\mathsf{Skip})} \tag{11}$$

There is only one rule for an assignment $\mathsf{v} := \mathsf{e}$ where the loose constant $w_0$ is used as a new target for the new assignment in $(\mathsf{s};(\mathsf{v} := \mathsf{w}_0))$ since the assignment may be nondeterministic.

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{v} := \mathsf{e}) \xrightarrow{\tau} (\mathsf{c}\ \mathbf{and}\ \mathsf{s};(\mathsf{w}_0 = \mathsf{e}),\mathsf{s};(\mathsf{v} := \mathsf{w}_0),\mathsf{Skip})} \tag{12}$$

Here, $s;(w_0 = e)$ in the constraints enables that the program variables in the expression $e$ are included in the alphabet of $s$.

For a specification, $\mathsf{x}$:[$\mathsf{p}$,$\mathsf{Q}$], if the precondition cannot be satisfied by the current state, or $s;p$ is **false**, it transits to a divergence.

$$\frac{\neg\,(s;\,p)}{(\mathsf{c},\mathsf{s},\mathsf{x}\!:[\mathsf{p},\mathsf{Q}]) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathsf{Div})} \tag{13}$$

If the precondition is satisfied and the commitment on $s$ is feasible, this action is completed and becomes *Skip*. Because the postcondition $Q$ may be nondeterministic, a loose constant ($w_0$) is used as the target of the later assignment.

$$\frac{s;\,p \quad \exists\, v' \bullet s;\,Q \quad v = out\alpha(s)}{(\mathsf{c},\mathsf{s},\mathsf{x}\!:[\mathsf{p},\mathsf{Q}]) \xrightarrow{\tau} (\mathsf{c}\ \mathbf{and}\ (\mathsf{s};\,\mathsf{Q}[\mathsf{w}_0/\mathsf{v}']),\mathsf{s};\,\mathsf{v} := \mathsf{w}_0,\mathsf{Skip})} \tag{14}$$

There is no rule for the case that $p$ is satisfied but $Q$ is not. If this case does happen, the action will go nowhere, even not allow time to pass. This state is called *timestop*. The timestop state can also be caused by the actions involving *Miracle* and deadline operators in *Circus Time*.

[1] In Hoare's book, *CHAOS*, the bottom element in the refinement ordering, is considered to be able to diverge as well as everything else it can do. There is no special representation for a simple divergent process like an internal loop. In Roscoe's book, such a special process is introduced as *Div*. Obviously, *Div* refines *CHAOS*. In practice such as in FDR, we identify a divergent process by means of checking whether it has an internal loop, and other behaviours of *CHAOS* like arbitrarily performing visible events can only confuse our decision. Since we will not use the operational semantics of *Circus Time* for the refinement, we, here, adopt *Div*, represented by *Chaos*, rather than *CHAOS* to represent a divergence.

### 5.2.3 Prefix, input and output

In *Circus Time*, we have a simple prefix operator for synchronisation only, and also have the input and output operators to pass values along a channel.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{ch} \to \mathsf{A}) \xrightarrow{\mathsf{ch}} (\mathsf{c}, \mathsf{s}, \mathsf{A})} \tag{15}$$

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{ch} \to \mathsf{A}) \xrightarrow{\mathsf{w}_0} (\mathsf{c}\,\mathbf{and}\,\mathsf{w}_0 \in \mathbb{Z}^+, \mathsf{s}, \mathsf{ch} \to \mathsf{A})} \tag{16}$$

$$\frac{T \neq \emptyset \quad x \notin \alpha s}{(\mathsf{c}, \mathsf{s}, \mathsf{ch?x}\!:\!\mathsf{T} \to \mathsf{A}) \xrightarrow{\mathsf{ch.w}_0} (\mathsf{c}\ \mathbf{and}\ \mathsf{w}_0 \in \mathsf{T}, \mathsf{s}; \mathbf{var}\,\mathsf{x} := \mathsf{w}_0, \mathbf{let}\,\mathsf{x} \bullet \mathsf{A})} \tag{17}$$

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{ch!e} \to \mathsf{A}) \xrightarrow{\mathsf{ch.w}_0} (\mathsf{c}\ \mathbf{and}\ \mathsf{s}; (\mathsf{w}_0 = \mathsf{e}), \mathsf{s}, \mathsf{A})} \tag{18}$$

We give two rules for a simple prefix that either $ch$ occurs immediately or arbitrary time units are allowed to pass. The simple prefix does not change the program variables. For the input operator, we declare a new variable in the scope of $\mathsf{A}$ to record a chosen value. In addition, the fresh loose constant $w_0$ is selected to symbolically represent the possible values through the channels. The output operator requires that $w_0$ does not occur in $c, s, e$ and $A$.

### 5.2.4 Timed event prefix

The timed event prefix is another variant of the prefix operator, in which a variable is used to record the time when the event occurs since the beginning of the action. Therefore, in the following rule, we declare a new variable $u$ within $s$ by **var**, while initialising the value of $u$ as zero, and also give the scope of $u$ by **let**.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{a@u} \to \mathsf{A}) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}; \mathbf{var}\,\mathsf{u} := 0, \mathbf{let}\,\mathsf{u} \bullet \mathsf{a} \to \mathsf{A})} \tag{19}$$

If this action evolves without performing $a$, $u$ is updated in the second rule. Note that $a$ may be just a channel, or a channel with parameters.

$$\frac{(\mathsf{c}, \mathsf{s}, \mathsf{a} \to \mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1, \mathsf{s}, \mathsf{a} \to \mathsf{A}) \quad l \in \mathbb{Z}^+}{(\mathsf{c}, \mathsf{s}, \mathbf{let}\,\mathsf{u} \bullet \mathsf{a} \to \mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1, \mathsf{s}; (\mathsf{u} := \mathsf{u} + \mathsf{l}), \mathbf{let}\,\mathsf{u} \bullet \mathsf{a} \to \mathsf{A})} \tag{20}$$

If $a$ occurs, $s$ is unchanged. After that, $u$ can be accessed by $\mathsf{A}$.

$$\frac{(\mathsf{c}, \mathsf{s}, \mathsf{a} \to \mathsf{A}) \xrightarrow{\mathsf{a}} (\mathsf{c}_1, \mathsf{s}_1, \mathsf{A})}{(\mathsf{c}, \mathsf{s}, \mathbf{let}\,\mathsf{u} \bullet \mathsf{a} \to \mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1, \mathsf{s}_1, \mathbf{let}\,\mathsf{u} \bullet \mathsf{A})} \tag{21}$$

The transition rules of a timed event prefix depend on the rules of other prefix operators.

### 5.2.5 Substitution

Invocation of unnamed parametrised actions is defined simply as a simple substitution of argument for the formal parameter as follows in *Circus Time*.

**Definition 6.** $(x : T \bullet A)(e) = A[e/x]$

**Definition 7.** $A[e/x] = \mathbf{var}\, x : T \bullet (x := e; A)$

Here, the requirement for $A[e/x]$ is $x \notin FV(e)$. That is, $e$ contains no free $x$. This is also the requirement on the unnamed parametrised .

Usually, a *Circus Time* program explicitly gives the types, by value, result, or by value-result, of the substitution for parameters. There is no transition rule for these substitutions. Instead, we syntactically transform them into the compounds of declarations, assignments and simple substitutions.

**Definition 8.** $(\mathbf{val}\, x : T \bullet A)(e) = \mathbf{var}\, y : T \bullet (y := e; A[y/x])$

**Definition 9.** $(\mathbf{res}\, x : T \bullet A)(e) = \mathbf{var}\, y : T \bullet (A[y/x];\ e := y)$

**Definition 10.** $(\textbf{val-res}\, x : T \bullet A)(e) = \mathbf{var}\, y : T \bullet (y := e;\ A[y/x];\ e := y)$

### 5.2.6 Guarded actions

In a guarded action, $g$ is a condition and is included in the alphabet of $A$. So, if $s; g$ is **true**, it will silently transit to $A$. Otherwise, it simply allows time to elapse.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{g} \& \mathsf{A}) \xrightarrow{\tau} (\mathsf{c} \ \mathbf{and} \ (\mathsf{s}; \mathsf{g}), \mathsf{s}, \mathsf{A})} \tag{22}$$

$$\frac{\neg\ (s; g)}{(\mathsf{c}, \mathsf{s}, \mathsf{g} \& \mathsf{A}) \xrightarrow{\mathsf{w}_0} (\mathsf{c} \ \mathbf{and} \ \mathsf{w}_0 \in \mathbb{Z}^+, \mathsf{s}, \mathsf{g} \& \mathsf{A})} \tag{23}$$

### 5.2.7 Sequential composition

We have two rules for sequential composition. One is the front action has not finished yet, the other is it has terminated.

$$\frac{(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_2, \mathsf{s}_\mathsf{s}, \mathsf{A}_2)}{(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1; \mathsf{B}) \xrightarrow{\mathsf{l}} (\mathsf{c}_2, \mathsf{s}_2, \mathsf{A}_2; \mathsf{B})} \tag{24}$$

$$\frac{}{(\mathsf{c}_1, \mathsf{s}_1, \mathsf{Skip}; \mathsf{B}) \xrightarrow{\tau} (\mathsf{c}_1, \mathsf{s}_1, \mathsf{B})} \tag{25}$$

### 5.2.8 Internal choice

For a internal choice, the rule is very simple, in which either $\mathsf{A}_1$ or $\mathsf{A}_2$ can silently decide the choice.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{A}_1 \sqcap \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}, \mathsf{A}_1), (\mathsf{c}, \mathsf{s}, \mathsf{A}_1 \sqcap \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}, \mathsf{A}_2)} \tag{26}$$

### 5.2.9 External choice

The transitions for the external choice operator are considered that both operands take a same copy of the current state, and run in parallel if the choice has not been resolved, or choose one of them if it can run an observable event or terminate. Therefore, we use a syntactic device $\mathbf{loc}\,\mathsf{s} \bullet \mathsf{A}$ to duplicate the current state ($\mathsf{s}$) to the action $\mathsf{A}$. There are five rules for external choice, and the first one duplicates the state and separates $\mathsf{A}_1$ and $\mathsf{A}_2$ in parallel.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{A}_1 \,\Box\, \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,\mathsf{s} \bullet \mathsf{A}_1 \,\Box\, \mathbf{loc}\,\mathsf{s} \bullet \mathsf{A}_2))} \tag{27}$$

$$\frac{(\mathsf{c}, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\tau} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3)}{(\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,\mathsf{s}_1 \bullet \mathsf{A}_1 \,\Box\, \mathbf{loc}\,\mathsf{s}_2 \bullet \mathsf{A}_2)) \xrightarrow{\tau} (\mathsf{c}_3, \mathsf{s}, (\mathbf{loc}\,\mathsf{s}_3 \bullet \mathsf{A}_3 \,\Box\, \mathbf{loc}\,\mathsf{s}_2 \bullet \mathsf{A}_2))} \tag{28}$$

$$\frac{(\mathsf{c}, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}_1} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad (\mathsf{c}, \mathsf{s}_2, \mathsf{A}_2) \xrightarrow{\mathsf{l}_2} (\mathsf{c}_4, \mathsf{s}_4, \mathsf{A}_4) \quad \mathsf{l}_1, \mathsf{l}_2 \in \mathbb{Z}^+}{(\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,\mathsf{s}_1 \bullet \mathsf{A}_1 \,\Box\, \mathbf{loc}\,\mathsf{s}_2 \bullet \mathsf{A}_2)) \xrightarrow{\mathsf{l}_1} (\mathsf{c}_3 \ \mathbf{and} \ \mathsf{c}_4 \ \mathbf{and} \ \mathsf{l}_1 = \mathsf{l}_2, \mathsf{s}, (\mathbf{loc}\,\mathsf{s}_3 \bullet \mathsf{A}_3 \,\Box\, \mathbf{loc}\,\mathsf{s}_4 \bullet \mathsf{A}_4))} \tag{29}$$

$$\frac{}{(\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,\mathsf{s}_1 \bullet \mathsf{Skip} \,\Box\, \mathbf{loc}\,\mathsf{s}_2 \bullet \mathsf{A})) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}_1, \mathsf{Skip})} \tag{30}$$

$$\frac{(\mathsf{c}, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad l \in \Sigma}{(\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,\mathsf{s}_1 \bullet \mathsf{A}_1 \,\Box\, \mathbf{loc}\,\mathsf{s}_2 \bullet \mathsf{A}_2)) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3)} \tag{31}$$

The second one allows either of $\mathsf{A}_1$ and $\mathsf{A}_2$ to transit to a new state through an invisible event without resolving the external choice. The third one states that the external choice can evolve only if both actions evolve at the same rate. Because $l_1$ and $l_2$ are the loose constants, we require them to be equal in the new constraints. The fourth rule takes place when one of the operands terminates. The external choice can be resolve by a visible event as the fifth rule.

### 5.2.10 Parallel composition

We use the same syntactic device in the transition rules for parallel composition, but, to duplicate both the constraints on loose constants and the current state. The first transition rule simply executes the duplication. However, before that, we require that $\mathsf{A}_1$ and $\mathsf{A}_2$ cannot update a same program variable ($x_1 \cap x_2 = \emptyset$), and the program variables that will be updated are included in the alphabet of $s$ ($(x_1 \cup x_2)' \subseteq out(\alpha s)$). The second one allows any non-synchronised event transition to happen, in which $l \in \Sigma^{+\tau}$ in the antecedents ensures that this is an event transition, and $l \notin cs$ in the constraints after the transition guarantees that $l$ is a non-synchronised event. Similarly, the third rule states that if $\mathsf{A}_1$ and $\mathsf{A}_2$ can perform the same event, they jointly transit to next state. The equality of $l_1$ and $l_2$ is identified in the after-transition constraints. The time evolution is described by the fourth rule and and the termination is by the fifth rule.

$$\frac{x_1 \cap x_2 = \emptyset \quad (x_1 \cup x_2)' \subseteq out(\alpha s)}{(\mathsf{c}, \mathsf{s}, \mathsf{A}_1[|\ \mathsf{x}_1 \,|\, \mathsf{cs} \,|\, \mathsf{x}_2\ |]\mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,(\mathsf{c}, \mathsf{s}) \bullet \mathsf{A}_1[|\ \mathsf{x}_1 \,|\, \mathsf{cs} \,|\, \mathsf{x}_2\ |]\mathbf{loc}\,(\mathsf{c}, \mathsf{s}) \bullet \mathsf{A}_2))} \tag{32}$$

$$\frac{(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad l \in \Sigma^{+\tau}}{\begin{array}{l}(\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,(\mathsf{c}_1, \mathsf{s}_1) \bullet \mathsf{A}_1[|\ \mathsf{x}_1 \,|\, \mathsf{cs} \,|\, \mathsf{x}_2\ |]\ \mathbf{loc}\,(\mathsf{c}_2, \mathsf{s}_2) \bullet \mathsf{A}_2)) \xrightarrow{\mathsf{l}} \\ \qquad (\mathsf{c}_3\ \mathbf{and}\ \mathsf{l} \notin \mathsf{cs}, \mathsf{s}, (\mathbf{loc}\,(\mathsf{c}_3, \mathsf{s}_3) \bullet \mathsf{A}_3[|\ \mathsf{x}_1 \,|\, \mathsf{cs} \,|\, \mathsf{x}_2\ |]\mathbf{loc}\,(\mathsf{c}_2, \mathsf{s}_2) \bullet \mathsf{A}_2))\end{array}} \tag{33}$$

$$\frac{(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}_1} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad (\mathsf{c}_2, \mathsf{s}_2, \mathsf{A}_2) \xrightarrow{\mathsf{l}_2} (\mathsf{c}_4, \mathsf{s}_4, \mathsf{A}_4) \quad l_1, l_2 \in \Sigma}{\begin{array}{l}(\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,(\mathsf{c}_1, \mathsf{s}_1) \bullet \mathsf{A}_1[|\ \mathsf{x}_1 \mid \mathsf{cs} \mid \mathsf{x}_2\ |]\ \mathbf{loc}\,(\mathsf{c}_2, \mathsf{s}_2) \bullet \mathsf{A}_2)) \xrightarrow{\mathsf{l}_1} \\ \quad (\mathsf{c}_3\ \mathbf{and}\ \mathsf{c}_4\ \mathbf{and}\ \mathsf{l}_1 = \mathsf{l}_2\ \mathbf{and}\ \mathsf{l}_1 \in \mathsf{cs}, \mathsf{s}, (\mathbf{loc}\,(\mathsf{c}_3, \mathsf{s}_3) \bullet \mathsf{A}_3[|\ \mathsf{x}_1 \mid \mathsf{cs} \mid \mathsf{x}_2\ |]\mathbf{loc}\,(\mathsf{c}_4, \mathsf{s}_4) \bullet \mathsf{A}_4))\end{array}} \tag{34}$$

$$\frac{(\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}_1} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad (\mathsf{c}_2, \mathsf{s}_2, \mathsf{A}_2) \xrightarrow{\mathsf{l}_2} (\mathsf{c}_4, \mathsf{s}_4, \mathsf{A}_4) \quad l_1, l_2 \in \mathbb{Z}^+}{\begin{array}{l}(\mathsf{c}, \mathsf{s}, (\mathbf{loc}\,(\mathsf{c}_1, \mathsf{s}_1) \bullet \mathsf{A}_1[|\ \mathsf{x}_1 \mid \mathsf{cs} \mid \mathsf{x}_2\ |]\ \mathbf{loc}\,(\mathsf{c}_2, \mathsf{s}_2) \bullet \mathsf{A}_2)) \xrightarrow{\mathsf{l}_1} \\ \quad (\mathsf{c}_3\ \mathbf{and}\ \mathsf{c}_4\ \mathbf{and}\ \mathsf{l}_1 = \mathsf{l}_2, \mathsf{s}, (\mathbf{loc}\,(\mathsf{c}_3, \mathsf{s}_3) \bullet \mathsf{A}_3[|\ \mathsf{x}_1 \mid \mathsf{cs} \mid \mathsf{x}_2\ |]\mathbf{loc}\,(\mathsf{c}_4, \mathsf{s}_4) \bullet \mathsf{A}_4))\end{array}} \tag{35}$$

$$\frac{}{(\mathsf{c},\mathsf{s},(\mathbf{loc}(\mathsf{c}_1,\mathsf{s}_1) \bullet \mathsf{Skip}[|\mathsf{x}_1 \,|\, \mathsf{cs} \,|\, \mathsf{x}_2\,|]\ \mathbf{loc}(\mathsf{c}_2,\mathsf{s}_2) \bullet \mathsf{Skip})) \xrightarrow{\tau} (\mathsf{c},(\mathsf{s}_1; \mathbf{end}\, \mathsf{x}_2) \oplus (\mathsf{s}_2; \mathbf{end}\, \mathsf{x}_1), \mathsf{Skip})} \tag{36}$$

In the final transition rule, the states of both the operands are merged. We terminate the scope of the variables from $\mathsf{x}_2$ and $\mathsf{x}_1$ in $\mathsf{s}_1$ and $\mathsf{s}_2$ respectively, and then update $\mathsf{s}_1$ by $\mathsf{s}_2$. Here, the operator $\oplus$ can extend the alphabet of $\mathsf{s}_1$ in terms of the variables that are included in $\mathsf{s}_2$ but not in $\mathsf{s}_1$.

### 5.2.11 Recursion

A recursive action can be expressed either by an explicit notation, e.g., $A = \mu X \bullet c \to X$, or by an implicit notation, $A = c \to A$. The transformation between the two different notations is purely syntactic. Here, one transition rules for a recursive action is given to the implicit notation, in which the copy rule is used to copy the body of the action as the remained program text. This can also be considered a recursive call that takes no time to happen.

$$\frac{A = B}{(\mathsf{c}, \mathsf{s}, \mathsf{A}) \xrightarrow{\mu} (\mathsf{c}, \mathsf{s}, \mathsf{B})} \tag{37}$$

### 5.2.12 Hiding

The hiding operator is the only one to be able to result in a divergence other than *Chaos* in *Circus Time* constructs. The hiding can also make program to happen urgently by one of the important assumptions in Timed CSP and *Circus Time*, which requires that internal events must occur as soon as they are enabled. There features of the hiding operator will be reflected in its operational semantics. There are four transition rules, and the first two of them are for the event transitions.

$$\frac{(\mathsf{c}, \mathsf{s}, \mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \quad l \in \Sigma^{+\tau}}{(\mathsf{c}, \mathsf{s}, \mathsf{A} \setminus \mathsf{cs}) \xrightarrow{\tau} (\mathsf{c}_1\ \mathbf{and}\ \mathsf{l} \in \mathsf{cs}, \mathsf{s}_1, \mathsf{A}_1 \setminus \mathsf{cs})} \tag{38}$$

$$\frac{(\mathsf{c}, \mathsf{s}, \mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1, \mathsf{s}_1, \mathsf{A}_1) \quad l \in \Sigma^{+\tau}}{(\mathsf{c}, \mathsf{s}, \mathsf{A} \setminus \mathsf{cs}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1\ \mathbf{and}\ \mathsf{l} \notin \mathsf{cs}, \mathsf{s}_1, \mathsf{A}_1 \setminus \mathsf{cs})} \tag{39}$$

Urgency of internal events is captured in the transition rule for time evolution. That is, a time evolution is allowed only if there are no outstanding internal events. Therefore, we have a special function, $Label(A)$, which will return all initial events, including visible and invisible events, since $A$.

$$\frac{(\mathsf{c},\mathsf{s},\mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \quad l \in \mathbb{Z}^+}{(\mathsf{c},\mathsf{s},\mathsf{A}\setminus\mathsf{cs}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1 \ \mathbf{and}\ \forall\, \mathsf{a}:\mathsf{Label}(\mathsf{A}) \bullet \mathsf{a} \notin \mathsf{cs}, \mathsf{s}_1, \mathsf{A}_1\setminus\mathsf{cs})} \tag{40}$$

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{Skip}\setminus\mathsf{cs}) \xrightarrow{\tau} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{Skip})} \tag{41}$$

Thus, the third rule states that the time evolution in $A$ can be adopted by $A \setminus cs$ only if no initial event from $A$ will be hidden. The final rule is for the termination with the hiding operator.

### 5.2.13 Delay

The delay action, $Wait\ d$, simply allows it to wait for $d$ time units. Therefore, we use a loose constant $w_0$ to denote any positive integer less than or equal to $d$, because this action may not terminate when engaging in an external choice.

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{Wait}\ \mathsf{d}) \xrightarrow{\mathsf{w}_0} (\mathsf{c}\ \mathbf{and}\ \mathsf{w}_0 \in \mathbb{Z}^+ \wedge \mathsf{w}_0 \leq \mathsf{d}, \mathsf{s}, \mathsf{Wait}\ (\mathsf{d}-\mathsf{w}_0))} \tag{42}$$

The following transition rule represents the termination of the delay action.

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{Wait}\ 0) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathsf{Skip})} \tag{43}$$

### 5.2.14 Timeout

In Timed CSP and *Circus Time*, the timeout operator can also be defined in terms of the prefix, external choice and hiding operators.

**Definition 11.** $P \triangleright \{d\}\ Q \mathrel{\widehat=} (P \ \Box\ Wait\ d; e \to Q) \setminus \{e\}$

However, this simple transformation of the timeout operator makes its operational semantics convolved. Here, we decide to give the transition rules directly. From Definition 11, the timeout operator is like the external choice to some extent. Therefore, the first rule is to duplicate $\mathsf{c}$ and $\mathsf{s}$ for $\mathsf{A}_2$, but not for $\mathsf{A}_1$. This is consistent with the denotational semantics of the timeout operator, in which $\mathsf{A}_2$ will adopt the original $\mathsf{c}$ and $\mathsf{s}$ from the beginning, rather than from the possible update by $\mathsf{A}_1$, if the timeout does occur.

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{A}_1 \triangleright\{\mathsf{d}\}\ \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},(\mathsf{A}_1 \triangleright\{\mathsf{d}\}\ \mathbf{loc}\,(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}_2))} \tag{44}$$

$$\frac{(\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \xrightarrow{\tau} (\mathsf{c}_3,\mathsf{s}_3,\mathsf{A}_3) \quad d \geq 0}{(\mathsf{c}_1,\mathsf{s}_1,(\mathsf{A}_1 \triangleright\{\mathsf{d}\}\ \mathbf{loc}(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}_2)) \xrightarrow{\tau} (\mathsf{c}_3,\mathsf{s}_3,(\mathsf{A}_3 \triangleright\{\mathsf{d}\}\ \mathbf{loc}(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}_2))} \tag{45}$$

$$\frac{(\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_3,\mathsf{s}_3,\mathsf{A}_3) \quad l \in \Sigma \quad d \geq 0}{(\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1 \triangleright\{\mathsf{d}\}\ \mathbf{loc}(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}_2) \xrightarrow{\mathsf{l}} (\mathsf{c}_3,\mathsf{s}_3,\mathsf{A}_3)} \tag{46}$$

$$\frac{(\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_3,\mathsf{s}_3,\mathsf{A}_3) \quad l \in \mathbb{Z}^+}{(\mathsf{c}_1,\mathsf{s}_1,(\mathsf{A}_1 \triangleright\{\mathsf{d}\}\ \mathbf{loc}(\mathsf{c},\mathsf{s})\bullet\mathsf{A}_2)) \xrightarrow{\mathsf{l}} (\mathsf{c}_3\ \mathbf{and}\ \mathsf{l}\leq\mathsf{d},\mathsf{s}_3,(\mathsf{A}_3 \triangleright\{\mathsf{d}-\mathsf{l}\}\ \mathbf{loc}(\mathsf{c},\mathsf{s})\bullet\mathsf{A}_2))} \tag{47}$$

$$\frac{d \geq 0}{(\mathsf{c}_1,\mathsf{s}_1,(\mathsf{Skip} \triangleright\{\mathsf{d}\}\ \mathbf{loc}(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}_2)) \xrightarrow{\tau} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{Skip})} \tag{48}$$

$$\frac{}{(\mathsf{c}_1,\mathsf{s}_1,(\mathsf{A}_1 \triangleright\{0\}\ \mathbf{loc}(\mathsf{c},\mathsf{s}) \bullet \mathsf{A}_2)) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathsf{A}_2)} \tag{49}$$

Then, the second rule is for the silent transition and the third one is for the visible transition. The difference of them is the latter can resolve the timeout. The fourth rule is for the time evolution only if $d \geq 0$. The fifth one is for the termination of $\mathsf{A}_1$, which can also resolve the timeout. The final one allows it to silently transit to $\mathsf{A}_2$ if the timeout is due.

### 5.2.15 Deadlines

There are two deadline operators in *Circus Time*. One ($A \blacktriangleright d$) is that $A$ must terminate before $d$, the other ($d \blacktriangleleft A$) is that a visible event in $A$ must occur before $d$.

The first transition rule for $A \blacktriangleright d$ is that the program simply transits to next execution no matter by silently or by visibly.

$$\frac{(\mathsf{c},\mathsf{s},\mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \quad l \in \Sigma^{+\tau} \quad d \geq 0}{(\mathsf{c},\mathsf{s},\mathsf{A} \blacktriangleright \mathsf{d}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1 \blacktriangleright \mathsf{d})} \tag{50}$$

If $\mathsf{A}$ may evolve, the deadline is decreased.

$$\frac{(\mathsf{c},\mathsf{s},\mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \quad l \in \mathbb{Z}^+ \wedge l \leq d}{(\mathsf{c},\mathsf{s},\mathsf{A} \blacktriangleright \mathsf{d}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1 \blacktriangleright (\mathsf{d}-\mathsf{l}))} \tag{51}$$

Of course, the termination of $\mathsf{A}$ can resolve the deadline operator.

$$\frac{d \geq 0}{(\mathsf{c},\mathsf{s},\mathsf{Skip} \blacktriangleright \mathsf{d}) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathsf{Skip})} \tag{52}$$

The transition rules for the deadline operator ($d \blacktriangleleft A$) are slightly different from those in $A \blacktriangleright d$, since the occurrence of any external event in $A$ can resolve the deadline operator. Therefore, if internal events occur, it just moves to the next state.

$$\frac{(\mathsf{c},\mathsf{s},\mathsf{A}) \xrightarrow{\tau} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \quad d \geq 0}{(\mathsf{c},\mathsf{s},\mathsf{d} \blacktriangleleft \mathsf{A}) \xrightarrow{\tau} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{d} \blacktriangleleft \mathsf{A}_1)} \tag{53}$$

Or, it may evolve by reducing the deadline.

$$\frac{(\mathsf{c},\mathsf{s},\mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \quad l \in \mathbb{Z}^+ \wedge l \leq d}{(\mathsf{c},\mathsf{s},\mathsf{d} \blacktriangleleft \mathsf{A}) \xrightarrow{\tau} (\mathsf{c}_1,\mathsf{s}_1,(\mathsf{d}-\mathsf{l}) \blacktriangleleft \mathsf{A}_1)} \tag{54}$$

The next two rules describe that either the observable event or the termination in $A$ can resolve the deadline.

$$\frac{(\mathsf{c},\mathsf{s},\mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1) \quad l \in \Sigma \wedge d \geq 0}{(\mathsf{c},\mathsf{s},\mathsf{d} \blacktriangleleft \mathsf{A}) \xrightarrow{\mathsf{l}} (\mathsf{c}_1,\mathsf{s}_1,\mathsf{A}_1)} \tag{55}$$

$$\frac{d \geq 0}{(\mathsf{c},\mathsf{s},\mathsf{d} \blacktriangleleft \mathsf{Skip}) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathsf{Skip})} \tag{56}$$

### 5.2.16 Interrupt

*Circus Time* has three kinds of the interrupt operators, catastrophic, generic and timed interrupts. For catastrophe, the interrupting event is explicit. So, it can be easily expressed by the generic interrupt operator.

**Definition 12.**

$$P \triangle_c Q \mathrel{\widehat{=}} P \triangle (c \rightarrow Q)$$

Therefore, we directly give the transition rules to the generic interrupt. There are totally six rules for describing the transitions of both operands. The first rule is for duplicating the current state to two operands respectively.

$$\frac{}{(\mathsf{c},\mathsf{s},\mathsf{A}_1 \triangle \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c},\mathsf{s},\mathbf{loc}\,\mathsf{s} \bullet \mathsf{A}_1 \triangle \mathbf{loc}\,\mathsf{s} \bullet \mathsf{A}_2)} \tag{57}$$

If $A_1$ can execute either an internal event or an external event, $A_1$ can move to the next step. Note that the state $s$ will not updated until the interrupt operator is resolved.

$$\frac{(\mathsf{c}, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad l \in \Sigma^{+\tau}}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_3 \bullet \mathsf{A}_3 \bigtriangleup \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2)} \tag{58}$$

Similarly, the interrupt action can evolve only if both operands can.

$$\frac{(\mathsf{c}, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}_1} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad (\mathsf{c}, \mathsf{s}_2, \mathsf{A}_2) \xrightarrow{\mathsf{l}_2} (\mathsf{c}_4, \mathsf{s}_4, \mathsf{A}_4) \quad l_1, l_2 \in \mathbb{Z}^+}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\mathsf{l}_1} (\mathsf{c}_3\, \mathbf{and}\, \mathsf{c}_4\, \mathbf{and}\, \mathsf{l}_1 = \mathsf{l}_2, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_3 \bullet \mathsf{A}_3 \bigtriangleup \mathbf{loc}\, \mathsf{s}_4 \bullet \mathsf{A}_4)} \tag{59}$$

If $A_1$ terminates, $A_2$ may be discarded.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{Skip} \bigtriangleup \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}_1, \mathsf{Skip})} \tag{60}$$

Obviously, the internal events from $A_2$ cannot interrupt $A_1$.

$$\frac{(\mathsf{c}, \mathsf{s}_2, \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}_4, \mathsf{s}_4, \mathsf{A}_4)}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\mathsf{l}} (\mathsf{c}_4, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup \mathbf{loc}\, \mathsf{s}_4 \bullet \mathsf{A}_4)} \tag{61}$$

And the interruption does occur by the external events from $A_2$.

$$\frac{(\mathsf{c}, \mathsf{s}_2, \mathsf{A}_2) \xrightarrow{\mathsf{l}} (\mathsf{c}_4, \mathsf{s}_4, \mathsf{A}_4) \quad l \in \Sigma}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\mathsf{l}} (\mathsf{c}_4, \mathsf{s}_4, \mathsf{A}_4)} \tag{62}$$

For the timed interrupt operator, there are five rules. Rule 64 and 65 state that $A_1$ either executes events or evolve, as long as $d \geq 0$, $A_1$ transits into the next step individually.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathsf{A}_1 \bigtriangleup_{\mathsf{d}} \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s} \bullet \mathsf{A}_1 \bigtriangleup_{\mathsf{d}} \mathbf{loc}\, \mathsf{s} \bullet \mathsf{A}_2)} \tag{63}$$

$$\frac{(\mathsf{c}, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad l \in \Sigma^{+\tau} \wedge d \geq 0}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup_{\mathsf{d}} \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_3 \bullet \mathsf{A}_3 \bigtriangleup_{\mathsf{d}} \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2)} \tag{64}$$

$$\frac{(\mathsf{c}, \mathsf{s}_1, \mathsf{A}_1) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}_3, \mathsf{A}_3) \quad l \in \mathbb{Z}^+ \wedge l \leq d}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup_{\mathsf{d}} \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\mathsf{l}} (\mathsf{c}_3, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_3 \bullet \mathsf{A}_3 \bigtriangleup_{\mathsf{d-l}} \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2)} \tag{65}$$

Or, $A_1$ terminates before $d$, the interrupt operator can be resolved.

$$\frac{d \geq 0}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{Skip} \bigtriangleup_{\mathsf{d}} \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}_1, \mathsf{Skip})} \tag{66}$$

Finally, when the time is up ($d = 0$), the program control is passed to $A_2$ inevitably.

$$\frac{}{(\mathsf{c}, \mathsf{s}, \mathbf{loc}\, \mathsf{s}_1 \bullet \mathsf{A}_1 \bigtriangleup_0 \mathbf{loc}\, \mathsf{s}_2 \bullet \mathsf{A}_2) \xrightarrow{\tau} (\mathsf{c}, \mathsf{s}_2, \mathsf{A}_2)} \tag{67}$$

## 6 Soundness

The transition rules are sound with respect to the denotational semantics [11] of *Circus Time* and the definitions for the transition relations given in Section **??**. As space is limited, we present the soundness of one of the most elaborate rules, external choice.

As we discussed in Section 5.2.9, the transition rules for external choice use the syntactic device **loc** to duplicate the values of program variables to two actions. Now, we give it a denotational semantics that we can use it in the proofs of the soundness of external choice.

**Definition 13.** $\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2 \mathrel{\widehat{=}} (\mathit{lift}(s_1); A_1) \Box (\mathit{lift}(s_2); A_2)$

In addition, we give two distributive laws for assignment and delay over external choice. The trivial proofs of the two laws can be found in Appendix.

**Law 2.** $\mathit{lift}(s); (A_1 \Box A_2) = (\mathit{lift}(s); A_1) \Box (\mathit{lift}(s); A_2)$

**Law 3.** $\mathit{Wait}\ d; (A_1 \Box A_2) = (\mathit{Wait}\ d; A_1) \Box (\mathit{Wait}\ d; A_2)$

We give five theorems that in turn correspond to the transition rules of external choice. Theorem 5, for Rule 27 in Section 5.2.9, simply makes the identical copies of the current state to two operands.

**Theorem 5.** $\forall\, w \bullet c \Rightarrow \mathit{lift}(s); (A_1 \Box A_2) \sqsubseteq \mathit{lift}(s); (\mathbf{loc}\, s \bullet A_1 \Box \mathbf{loc}\, s \bullet A_2))$

*Proof.*

$$\begin{aligned}
&\mathit{lift}(s); (\mathbf{loc}\, s \bullet A_1 \Box \mathbf{loc}\, s \bullet A_2)) && \text{[Definition 13]}\\
=&\mathit{lift}(s); ((\mathit{lift}(s); A_1) \Box (\mathit{lift}(s); A_2)) && \text{[Lemma 2]}\\
=&\mathit{lift}(s); \mathit{lift}(s); (A_1 \Box A_2) && \text{[idempotence of assignment]}\\
=&\mathit{lift}(s); (A_1 \Box A_2) && \text{[R.H.S.= L.H.S.]}
\end{aligned}$$

□

The next theorem (for Rule 28) states that the left operand makes an internal progress that of course cannot resolve the choice. Note that all rules for external choice are symmetric.

**Theorem 6.**

$$\begin{aligned}
&\forall\, w \bullet c \wedge c_3 \wedge (\mathit{lift}(s_1); A_1 \sqsubseteq \mathit{lift}(s_3); A_3) \Rightarrow\\
&\qquad \mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2) \sqsubseteq \mathit{lift}(s); (\mathbf{loc}\, s_3 \bullet A_3 \Box \mathbf{loc}\, s_2 \bullet A_2)
\end{aligned}$$

*Proof.*

$$\begin{aligned}
&\mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2) && \text{[Definition 13]}\\
=&\mathit{lift}(s); (\mathit{lift}(s_1); A_1 \Box \mathit{lift}(s_2); A_2) && \text{[assumption, monotonicity]}\\
\sqsubseteq&\mathit{lift}(s); (\mathit{lift}(s_3); A_3 \Box \mathit{lift}(s_2); A_2) && \text{[Definition 13]}\\
=&\mathit{lift}(s); (\mathbf{loc}\, s_3 \bullet A_3 \Box \mathbf{loc}\, s_2 \bullet A_2) && \text{[L.H.S.=R.H.S.]}
\end{aligned}$$

□

The above proof replies on the monotonicity of external choice in the refinement relation, $(A_1 \sqsubseteq A_2) \Rightarrow (A_1 \Box B) \sqsubseteq (A_2 \Box B)$. Theorem 7 (for Rule 29) proves that the external choice action can evolve if both operands can evolve at a same rate.

**Theorem 7.**

$$\begin{aligned}
&\left(\begin{array}{c} \forall\, w \bullet c \wedge c_3 \wedge c_4 \wedge (\mathit{lift}(s_1); A_1 \sqsubseteq \mathit{Wait}\ l_1; \mathit{lift}(s_3); A_3) \wedge \\ l_1 = l_2 \wedge (\mathit{lift}(s_2); A_2 \sqsubseteq \mathit{Wait}\ l_2; \mathit{lift}(s_4); A_4) \end{array}\right) \Rightarrow\\
&\quad \mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2) \sqsubseteq \mathit{lift}(s); \mathit{Wait}\ l_1; (\mathbf{loc}\, s_3 \bullet A_3 \Box \mathbf{loc}\, s_4 \bullet A_4)
\end{aligned}$$

*Proof.*

$$\begin{aligned}
&\mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2) && \text{[Definition 13]}\\
=&\mathit{lift}(s); (\mathit{lift}(s_1); A_1 \Box \mathit{lift}(s); A_2) && \text{[assumption, monotonicity]}\\
\sqsubseteq&\mathit{lift}(s); ((\mathit{Wait}\ l_1; \mathit{lift}(s_3); A_3) \Box (\mathit{Wait}\ l_2; \mathit{lift}(s_4); A_4)) && \text{[assump. } (l_1 = l_2)\text{]}\\
=&\mathit{lift}(s); ((\mathit{Wait}\ l_1; \mathit{lift}(s_3); A_3) \Box (\mathit{Wait}\ l_1; \mathit{lift}(s_4); A_4)) && \text{[Lemma 3]}\\
=&\mathit{lift}(s); \mathit{Wait}\ l_1; (\mathit{lift}(s_3); A_3 \Box \mathit{lift}(s_4); A_4) && \text{[Definition 13]}\\
=&\mathit{lift}(s); \mathit{Wait}\ l_1; (\mathbf{loc}\, s_3 \bullet A_3 \Box \mathbf{loc}\, s_4 \bullet A_4) && \text{[L.H.S.=R.H.S.]}
\end{aligned}$$

□

This proof replies on the monotonicity of external choice for both operands, $(A_1 \sqsubseteq A_2 \land B_1 \sqsubseteq B_2) \Rightarrow (A_1 \Box B_1) \sqsubseteq (A_2 \Box B_2)$. Theorem 8 (for Rule 30) states that the termination of one operands may resolve the external choice operator.

**Theorem 8.** $\forall\, w \bullet c \Rightarrow \mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet \mathit{Skip} \Box \mathbf{loc}\, s_2 \bullet A_2) \sqsubseteq \mathit{lift}(s_1); \mathit{Skip}$

*Proof.*

$$\begin{array}{ll}
\quad \mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet \mathit{Skip} \Box \mathbf{loc}\, s_2 \bullet A_2) & [\text{Definition 13}]\\
= \mathit{lift}(s); ((\mathit{lift}(s_1); \mathit{Skip}) \Box (\mathit{lift}(s_2); A_2)) & [\text{idempotence of assignment}]\\
= \mathit{lift}(s); ((\mathit{lift}(s_1); \mathit{Skip}) \Box (\mathit{lift}(s_1); \mathit{lift}(s_2); A_2)) & [\text{Lemma 2}]\\
= \mathit{lift}(s); \mathit{lift}(s_1); (\mathit{Skip} \Box \mathit{lift}(s_2); A_2) & [\text{property of } \mathit{Skip} \text{ in } \Box]\\
\sqsubseteq \mathit{lift}(s); \mathit{lift}(s_1); \mathit{Skip} & [\text{idempotence of assignment}]\\
= \mathit{lift}(s_1); \mathit{Skip} & [\text{L.H.S.=R.H.S.}]
\end{array}$$

□

This proof replies on the well-known refinement relation, $\mathit{Skip} \Box A \sqsubseteq \mathit{Skip}$, in CSP, which is admittedly valid in *Circus*. The final theorem (for Rule 31) proves that the visible event can resolve the choice.

**Theorem 9.**

$$\begin{array}{l}
\forall\, w \bullet c \land c_3 \land l \in \Sigma \land (\mathit{lift}(s_1); A_1 \sqsubseteq \mathit{lift}(s_3); l \to A_3 \Box \mathit{lift}(s_1); A_1)\\
\qquad\qquad \Rightarrow \left( \begin{array}{l} \mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2) \sqsubseteq \\ (\mathit{lift}(s_3); l \to A_3) \Box (\mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2)) \end{array} \right)
\end{array}$$

*Proof.*

$$\begin{array}{ll}
\quad \mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2) & [\text{Definition 13}]\\
= \mathit{lift}(s); (\mathit{lift}(s_1); A_1 \Box \mathit{lift}(s_2); A_2) & [\text{assumption, monotonicity}]\\
\sqsubseteq \mathit{lift}(s); ((\mathit{lift}(s_3); l \to A_3) \Box (\mathit{lift}(s_1); A_1) \Box (\mathit{lift}(s_2); A_2)) & [\text{Lemma 2}]\\
= (\mathit{lift}(s); \mathit{lift}(s_3); l \to A_3) \Box (\mathit{lift}(s); (\mathit{lift}(s_1); A_1 \Box \mathit{lift}(s_2); A_2)) & [\text{Def. 13}]\\
= (\mathit{lift}(s); \mathit{lift}(s_3); l \to A_3) \Box (\mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2)) & [\text{idemp of :=}]\\
= (\mathit{lift}(s_3); l \to A_3) \Box (\mathit{lift}(s); (\mathbf{loc}\, s_1 \bullet A_1 \Box \mathbf{loc}\, s_2 \bullet A_2)) & [\text{L.H.S.=R.H.S.}]
\end{array}$$

□

As a result, the five theorems (Theorem 5-9 conclude the soundness of the operational semantics for external choice. The soundness for other operators can be proved in a similar way.

## 7 Conclusion

## Appendix

To prove Law 2 and 3, we need to know some denotational semantics of delay, sequential composition and external choice.

**Definition 14.**

$$\mathit{Wait}\ d \mathrel{\widehat{=}} \mathbf{R_{ct}}(\mathbf{true} \vdash {}^\frown/tr' = {}^\frown/tr \land (\#tr' - \#tr < d \lhd \mathit{wait}' \rhd (\#tr' - \#tr = d \land \mathit{state}' = \mathit{state})))$$

**Definition 15.**

$$P\,;\,Q = \mathbf{R_{ct}} \left( \begin{array}{c} \neg\,(\mathbf{R1_{ct}}(P^f_f)\,;\,\mathbf{R1_{ct}}(\mathbf{true})) \land \neg\,(\mathbf{R1_{ct}}(P^t_f)\,;\,\mathbf{R1_{ct}}(\neg\, \mathit{wait} \land Q^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(P^t_f)\,;\,\mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd Q^t_f) \end{array} \right)$$

**Definition 16.**

$$P \;\square\; Q \mathrel{\widehat=} \mathbf{R_{ct}} \left( \begin{array}{c} \neg\, (((P_f^f \vee Q_f^f) \wedge tr' = tr)\,;\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\, ((P_f^f \wedge ((Q_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait');\; tr' - tr = \langle\langle\rangle\rangle));\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\, ((Q_f^f \wedge ((P_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait');\; tr' - tr = \langle\langle\rangle\rangle));\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\, (P_f^f \wedge ((Q_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait')\,;\, head(tr' - tr) \neq \langle\rangle)) \wedge \\ \neg\, (Q_f^f \wedge ((P_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait')\,;\, head(tr' - tr) \neq \langle\rangle)) \wedge \\ \neg\, ((P_f^f \vee Q_f^f) \wedge head(diff(tr', tr)) \neq \langle\rangle) \\ \vdash \\ (P_f^t \wedge Q_f^t \wedge wait' \wedge {}^\frown/tr' = {}^\frown/tr) \vee (Diff(P_f^t, Q_f^t) \wedge (P_f^t \vee Q_f^t)) \end{array} \right)$$

$$Diff(P, Q) \mathrel{\widehat=} (((P \wedge Q \wedge wait' \wedge {}^\frown/tr' = {}^\frown/tr)\,;\, term\_next) \vee term\_now) \tag{68}$$

$$term\_now \mathrel{\widehat=} ((\neg\, wait' \wedge tr' = tr) \vee head(diff(tr', tr)) \neq \langle\rangle) \tag{69}$$

$$term\_next \mathrel{\widehat=} ((\neg\, wait' \wedge tr' - tr = \langle\langle\rangle\rangle) \vee head(tr' - tr) \neq \langle\rangle) \tag{70}$$

**Lemma 5.** $\mathbf{R1_{ct}}(\mathbf{false}) = \mathbf{false}$

**Lemma 6.** $\mathbf{R1_{ct}}(\mathbf{true}) = \mathbf{true}$

**Lemma 7.** $(\mathbf{R1_{ct}}(tr' = tr);\; \mathbf{R1_{ct}}(tr' = tr)) = \mathbf{R1_{ct}}(tr' = tr)$

**Lemma 8.** *(relational assignment)* $(x := e;\, x := e) = (x := e)$

**Lemma 9.** *(relational assignment)* $(x := e;\, x := e_1) = (x := e_1)$

**Law 4.** $lift(s);\, lift(s) = lift(s)$

*Proof.*

$$\begin{array}{ll} \quad lift(s);\, lift(s) & [lift(5)] \\ = \mathbf{R_{ct}}(\mathbf{true} \vdash s \wedge tr' = tr \wedge \neg\, wait');\, \mathbf{R_{ct}}(\mathbf{true} \vdash s \wedge tr' = tr \wedge \neg\, wait') & [\text{Definition 15}] \\ = \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(\mathbf{false});\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait');\, \mathbf{R1_{ct}}(\neg\, wait \wedge \mathbf{false})) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait');\, \mathbf{R1_{ct}}(I\!I \lhd wait \rhd (s \wedge tr' = tr \wedge \neg\, wait')) \end{array} \right) & \\ & [\text{Lemma 5,6}] \\ = \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{false};\, \mathbf{true}) \wedge \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait');\, (\neg\, wait \wedge \mathbf{false})) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait');\, \mathbf{R1_{ct}}(I\!I \lhd wait \rhd (s \wedge tr' = tr \wedge \neg\, wait')) \end{array} \right) & \\ & [\text{propositional calculus}] \\ = \mathbf{R_{ct}} \left( \mathbf{true} \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait');\, \mathbf{R1_{ct}}(I\!I \lhd wait \rhd (s \wedge tr' = tr \wedge \neg\, wait')) \right) & \\ & [\text{relational calculus}] \\ = \mathbf{R_{ct}} \left( \mathbf{true} \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait');\, \mathbf{R1_{ct}}(\neg\, wait \wedge s \wedge tr' = tr \wedge \neg\, wait') \right) & \\ & [\text{Lemma 7}] \\ = \mathbf{R_{ct}} \left( \mathbf{true} \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait');\, (\neg\, wait \wedge s \wedge tr' = tr \wedge \neg\, wait') \right) & \\ & [\text{relational calculus and Lemma 8}] \\ = \mathbf{R_{ct}}(\mathbf{true} \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, wait')) & [\text{property of } \mathbf{R1_{ct}}] \\ = \mathbf{R_{ct}}(\mathbf{true} \vdash s \wedge tr' = tr \wedge \neg\, wait') & [lift(5)] \\ = lift(s) & \end{array}$$

□

**Law 5.** $lift(s);\, lift(s_1) = lift(s_1)$

*Proof.* The proof of this law is similar to that of Law 4, with respect to Lemma 9. □

**Law 2** $\mathit{lift}(s);\, (A_1 \,\Box\, A_2) = (\mathit{lift}(s);\, A_1) \,\Box\, (\mathit{lift}(s);\, A_2)$

*Proof.*

$$
\begin{array}{ll}
& L.H.S \\
= & \mathit{lift}(s);\, (A_1 \,\Box\, A_2) \hfill [\mathit{lift}(5) \text{ and reactive design}] \\
= & \mathbf{R_{ct}}(\mathbf{true} \vdash s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R_{ct}}(\neg\, (A_1 \,\Box\, A_2)^f_f \vdash (A_1 \,\Box\, A_2)^t_f) \hfill [\text{Definition 15}] \\
= & \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(\mathbf{false});\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(\neg\, \mathit{wait} \wedge (A_1 \,\Box\, A_2)^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd (A_1 \,\Box\, A_2)^t_f) \end{array} \right) \\
& \hfill [\text{Lemma 6,5 and relational calculus}] \\
= & \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(\neg\, \mathit{wait} \wedge (A_1 \,\Box\, A_2)^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd (A_1 \,\Box\, A_2)^t_f) \end{array} \right) \\
& \hfill [\text{ only } tr' \text{ and } \mathit{ref}' \text{ are interested in a divergence }] \\
= & \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(tr' = tr);\, \mathbf{R1_{ct}}((A_1 \,\Box\, A_2)^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd (A_1 \,\Box\, A_2)^t_f) \end{array} \right) \hfill [\text{relational calculus}] \\
= & \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(tr' = tr);\, \mathbf{R1_{ct}}((A_1 \,\Box\, A_2)^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr);\, \mathbf{R1_{ct}}((A_1 \,\Box\, A_2)^t_f) \end{array} \right) \hfill [\text{property of } \mathbf{R1_{ct}}] \\
= & \mathbf{R_{ct}} \left( \neg\, (\mathbf{R1_{ct}}((A_1 \,\Box\, A_2)^f_f)) \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr);\, \mathbf{R1_{ct}}((A_1 \,\Box\, A_2)^t_f) \right) \hfill [\text{property of } \mathbf{R1_{ct}}] \\
= & \mathbf{R_{ct}} \left( \neg\, (A_1 \,\Box\, A_2)^f_f \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr);\, \mathbf{R1_{ct}}((A_1 \,\Box\, A_2)^t_f) \right)
\end{array}
$$

$$
\begin{array}{ll}
& R.H.S \\
= & (\mathit{lift}(s);\, A_1) \,\Box\, (\mathit{lift}(s);\, A_2) \\
= & (\mathbf{R_{ct}}(\mathbf{true} \vdash s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R_{ct}}(\neg\, {A_1}^f_f \vdash {A_1}^t_f)) \,\Box \hfill [\text{Definition 15}] \\
& (\mathbf{R_{ct}}(\mathbf{true} \vdash s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R_{ct}}(\neg\, {A_2}^f_f \vdash {A_2}^t_f)) \\
= & \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(\mathbf{false});\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(\neg\, \mathit{wait} \wedge {A_1}^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd {A_1}^t_f) \end{array} \right) \Box \\
& \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(\mathbf{false});\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(\neg\, \mathit{wait} \wedge {A_2}^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd {A_2}^t_f) \end{array} \right) \\
& \hfill [\text{Lemma 6,5 and relational calculus}] \\
= & \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(\neg\, \mathit{wait} \wedge {A_1}^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd {A_1}^t_f) \end{array} \right) \Box \hfill [\text{relational calculus}] \\
& \mathbf{R_{ct}} \left( \begin{array}{l} \neg\, (\mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(\neg\, \mathit{wait} \wedge {A_2}^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr \wedge \neg\, \mathit{wait}');\, \mathbf{R1_{ct}}(I\!I \lhd \mathit{wait} \rhd {A_2}^t_f) \end{array} \right) \\
= & \mathbf{R_{ct}}(\neg\, {A_1}^f_f \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr);\, \mathbf{R1_{ct}}({A_1}^t_f)) \,\Box\, \mathbf{R_{ct}}(\neg\, {A_2}^f_f \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr);\, \mathbf{R1_{ct}}({A_2}^t_f)) \\
& \hfill [\text{Definition 16}]
\end{array}
$$

$$= \mathbf{R_{ct}} \left( \begin{array}{l} \neg\,(A_1 \,\Box\, A_2)^f_f \vdash \\ \left( \begin{array}{l} (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)) \wedge \\ (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{2}{}^t_f)) \wedge \\ wait' \wedge {}^\frown/tr' = {}^\frown/tr \end{array} \right) \vee \\ \left( \begin{array}{l} \mathit{Diff}((\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)), (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f))) \wedge \\ ((\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)) \vee (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{2}{}^t_f))) \end{array} \right) \end{array} \right)$$

[relational calculus]

$$= \mathbf{R_{ct}} \left( \begin{array}{l} \neg\,(A_1 \,\Box\, A_2)^f_f \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr); (\mathbf{R1_{ct}}(A_{1}{}^t_f) \wedge \mathbf{R1_{ct}}(A_{2}{}^t_f) \wedge wait' \wedge {}^\frown/tr' = {}^\frown/tr) \vee \\ \left( \begin{array}{l} \mathit{Diff}((\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)), (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f))) \wedge \\ ((\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)) \vee (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{2}{}^t_f))) \end{array} \right) \end{array} \right)$$

[Definition 16]

$$= \mathbf{R_{ct}} \left( \begin{array}{l} \neg\,(A_1 \,\Box\, A_2)^f_f \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr); (\mathbf{R1_{ct}}(A_{1}{}^t_f) \wedge \mathbf{R1_{ct}}(A_{2}{}^t_f) \wedge wait' \wedge {}^\frown/tr' = {}^\frown/tr) \vee \\ \left( \begin{array}{l} \left( \left( \begin{array}{l} \mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)) \wedge \\ (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)) \wedge \\ wait' \wedge {}^\frown/tr' = {}^\frown/tr \end{array} \right); term\_next \vee term\_now \right) \wedge \\ ((\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{1}{}^t_f)) \vee (\mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}(A_{2}{}^t_f))) \end{array} \right) \end{array} \right)$$

[relational calculus]

$$= \mathbf{R_{ct}} \left( \begin{array}{l} \neg\,(A_1 \,\Box\, A_2)^f_f \vdash \\ \mathbf{R1_{ct}}(s \wedge tr' = tr); (\mathbf{R1_{ct}}(A_{1}{}^t_f) \wedge \mathbf{R1_{ct}}(A_{2}{}^t_f) \wedge wait' \wedge {}^\frown/tr' = {}^\frown/tr) \vee \\ \mathbf{R1_{ct}}(s \wedge tr' = tr); \left( \begin{array}{l} \left( \left( \begin{array}{l} \mathbf{R1_{ct}}(A_{1}{}^t_f)) \wedge \mathbf{R1_{ct}}(A_{1}{}^t_f)) \\ \wedge\, wait' \wedge {}^\frown/tr' = {}^\frown/tr \end{array} \right); term\_next \vee term\_now \right) \wedge \\ (\mathbf{R1_{ct}}(A_{1}{}^t_f) \vee \mathbf{R1_{ct}}(A_{2}{}^t_f)) \end{array} \right) \end{array} \right)$$

[relational calculus]

$$= \mathbf{R_{ct}} \left( \neg\,(A_1 \,\Box\, A_2)^f_f \vdash \mathbf{R1_{ct}}(s \wedge tr' = tr); \mathbf{R1_{ct}}((A_1 \,\Box\, A_2)^t_f) \right)$$

□

**Law 3** $Wait\ d; (A_1 \,\Box\, A_2) = (Wait\ d; A_1) \,\Box\, (Wait\ d; A_2)$

*Proof.*

$L.H.S.$

$= Wait\ d; (A_1 \,\Box\, A_2)$ [Definition 15]

$$= \mathbf{R_{ct}} \left( \begin{array}{l} \neg\,(\mathbf{R1_{ct}}((Wait\ d)^f_f); \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\,(\mathbf{R1_{ct}}((Wait\ d)^t_f); \mathbf{R1_{ct}}(\neg\, wait \wedge (A_1 \,\Box\, A_2)^f_f)) \\ \vdash \\ \mathbf{R1_{ct}}((Wait\ d)^t_f); \mathbf{R1_{ct}}(I\!I \lhd wait \rhd (A_1 \,\Box\, A_2)^t_f) \end{array} \right)$$

[Definition 14]

$$= \mathbf{R_{ct}} \left( \begin{array}{l} \neg\,(\mathbf{R1_{ct}}(\mathbf{false}; \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg \left( \mathbf{R1_{ct}} \left( \begin{array}{c} {}^\frown/tr' = {}^\frown/tr \wedge \#tr' - \#tr < d \\ \lhd\ wait'\ \rhd \\ {}^\frown/tr' = {}^\frown/tr \wedge \#tr' - \#tr = d \wedge state' = state \end{array} \right); \mathbf{R1_{ct}}(\neg\, wait \wedge (A_1 \,\Box\, A_2)^f_f) \right) \\ \vdash \\ \mathbf{R1_{ct}} \left( \begin{array}{c} {}^\frown/tr' = {}^\frown/tr \wedge \#tr' - \#tr < d \\ \lhd\ wait'\ \rhd \\ {}^\frown/tr' = {}^\frown/tr \wedge \#tr' - \#tr = d \wedge state' = state \end{array} \right); \mathbf{R1_{ct}}(I\!I \lhd wait \rhd (A_1 \,\Box\, A_2)^t_f) \end{array} \right)$$

[relatinal calculus]

$$=\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}(\neg\,wait' \wedge {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state);\, \mathbf{R1_{ct}}(\neg\,wait \wedge (A_1 \Box A_2)^f_f))\\ \vdash\\ \mathbf{R1_{ct}}({\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d \wedge wait');\, \mathbf{R1_{ct}}(\mathit{I\!I} \wedge wait) \vee\\ \mathbf{R1_{ct}}(\neg\,wait' \wedge {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state);\, \mathbf{R1_{ct}}(\neg\,wait \wedge (A_1 \Box A_2)^t_f)\end{array}\right)$$

[only $tr'$ and $ref'$ are interested in a divergence]

$$=\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}({\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d);\, \mathbf{R1_{ct}}((A_1 \Box A_2)^f_f))\\ \vdash\\ \mathbf{R1_{ct}}({\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d \wedge wait');\, \mathbf{R1_{ct}}(\mathit{I\!I} \wedge wait) \vee\\ \mathbf{R1_{ct}}(\neg\,wait' \wedge {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state);\, \mathbf{R1_{ct}}(\neg\,wait \wedge (A_1 \Box A_2)^t_f)\end{array}\right)$$

*R.H.S.*

$=(Wait\ d; A_1) \Box (Wait\ d; A_2)$ [Definition 14 and 15]

$$=\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}(\mathbf{false})\,;\mathbf{R1_{ct}}(\mathbf{true})) \wedge\\ \neg\left(\mathbf{R1_{ct}}\left(\begin{array}{c}{\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d\\ \lhd\ wait'\ \rhd\\ {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state\end{array}\right); \mathbf{R1_{ct}}(\neg\,wait \wedge {A_1}^f_f)\right)\\ \vdash\\ \mathbf{R1_{ct}}\left(\begin{array}{c}{\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d\\ \lhd\ wait'\ \rhd\\ {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state\end{array}\right); \mathbf{R1_{ct}}(\mathit{I\!I} \lhd wait \rhd {A_1}^t_f)\end{array}\right)$$

$$\Box$$

$$\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}(\mathbf{false})\,;\mathbf{R1_{ct}}(\mathbf{true})) \wedge\\ \neg\left(\mathbf{R1_{ct}}\left(\begin{array}{c}{\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d\\ \lhd\ wait'\ \rhd\\ {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state\end{array}\right); \mathbf{R1_{ct}}(\neg\,wait \wedge {A_2}^f_f)\right)\\ \vdash\\ \mathbf{R1_{ct}}\left(\begin{array}{c}{\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d\\ \lhd\ wait'\ \rhd\\ {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state\end{array}\right); \mathbf{R1_{ct}}(\mathit{I\!I} \lhd wait \rhd {A_2}^t_f)\end{array}\right)$$

[relational calculus]

$$=\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}({\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d);\, \mathbf{R1_{ct}}({A_1}^f_f))\\ \vdash\\ \mathbf{R1_{ct}}({\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d \wedge wait')\,;\mathbf{R1_{ct}}(\mathit{I\!I} \wedge wait) \vee\\ \mathbf{R1_{ct}}(\neg\,wait' \wedge {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state)\,;\mathbf{R1_{ct}}(\neg\,wait \wedge {A_1}^t_f)\end{array}\right) \Box$$

$$\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}({\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d);\, \mathbf{R1_{ct}}({A_2}^f_f))\\ \vdash\\ \mathbf{R1_{ct}}({\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr < d \wedge wait')\,;\mathbf{R1_{ct}}(\mathit{I\!I} \wedge wait) \vee\\ \mathbf{R1_{ct}}(\neg\,wait' \wedge {\frown}/tr' = {\frown}/tr \wedge \#tr' - \#tr = d \wedge state' = state)\,;\mathbf{R1_{ct}}(\neg\,wait \wedge {A_2}^t_f)\end{array}\right)$$

[Definition 16 and $P$ and $Q$ are used to denote the two folded operands]

$Case1 : d = 0$

$$=\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr=0);\,\mathbf{R1_{ct}}(A1^f_f))\\ \vdash\\ \mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr<0\wedge wait');\mathbf{R1_{ct}}(I\!I\wedge wait)\vee\\ \mathbf{R1_{ct}}(\neg\,wait'\wedge\frown/tr'=\frown/tr\wedge\#tr'-\#tr=0\wedge state'=state);\mathbf{R1_{ct}}(\neg\,wait\wedge A1^t_f)\end{array}\right)\Box$$

$$\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr=0);\,\mathbf{R1_{ct}}(A2^f_f))\\ \vdash\\ \mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr<0\wedge wait');\mathbf{R1_{ct}}(I\!I\wedge wait)\vee\\ \mathbf{R1_{ct}}(\neg\,wait'\wedge\frown/tr'=\frown/tr\wedge\#tr'-\#tr=0\wedge state'=state);\mathbf{R1_{ct}}(\neg\,wait\wedge A2^t_f)\end{array}\right)$$

[relational calculus]

$=\mathbf{R_{ct}}(\neg\,A1^f_f\vdash A1^t_f)\;\Box\;\mathbf{R_{ct}}(\neg\,A2^f_f\vdash A1^t_2)$

$=A_1\;\Box\;A_2$

$Case2 : d > 0$

$$=\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr=d);\,\mathbf{R1_{ct}}(A1^f_f))\\ \vdash\\ \mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr<d\wedge wait');\mathbf{R1_{ct}}(I\!I\wedge wait)\vee\\ \mathbf{R1_{ct}}(\neg\,wait'\wedge\frown/tr'=\frown/tr\wedge\#tr'-\#tr=d\wedge state'=state);\mathbf{R1_{ct}}(\neg\,wait\wedge A1^t_f)\end{array}\right)\Box$$

$$\mathbf{R_{ct}}\left(\begin{array}{l}\neg\,(\mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr=d);\,\mathbf{R1_{ct}}(A2^f_f))\\ \vdash\\ \mathbf{R1_{ct}}(\frown/tr'=\frown/tr\wedge\#tr'-\#tr<d\wedge wait');\mathbf{R1_{ct}}(I\!I\wedge wait)\vee\\ \mathbf{R1_{ct}}(\neg\,wait'\wedge\frown/tr'=\frown/tr\wedge\#tr'-\#tr=d\wedge state'=state);\mathbf{R1_{ct}}(\neg\,wait\wedge A2^t_f)\end{array}\right)$$

[Definition 16 and $P$ and $Q$ are used to denote the two folded operands]

$$=\mathbf{R_{ct}}\left(\begin{array}{c}\neg\,((P^f_f\vee Q^f_f)\wedge tr'=tr);\,\mathbf{R1_{ct}}(\mathbf{true}))\wedge\\ \neg\,((P^f_f\wedge((Q_f\wedge\frown/tr'=\frown/tr\wedge wait');\,tr'-tr=\langle\langle\rangle\rangle));\,\mathbf{R1_{ct}}(\mathbf{true}))\wedge\\ \neg\,((Q^f_f\wedge((P_f\wedge\frown/tr'=\frown/tr\wedge wait');\,tr'-tr=\langle\langle\rangle\rangle));\,\mathbf{R1_{ct}}(\mathbf{true}))\wedge\\ \neg\,(P^f_f\wedge((Q_f\wedge\frown/tr'=\frown/tr\wedge wait');\,head(tr'-tr)\neq\langle\rangle))\wedge\\ \neg\,(Q^f_f\wedge((P_f\wedge\frown/tr'=\frown/tr\wedge wait');\,head(tr'-tr)\neq\langle\rangle))\wedge\\ \neg\,((P^f_f\vee Q^f_f)\wedge head(diff(tr',tr))\neq\langle\rangle)\\ \vdash\\ (P^t_f\wedge Q^t_f\wedge wait'\wedge\frown/tr'=\frown/tr)\vee(Diff(P^t_f,Q^t_f)\wedge(P^t_f\vee Q^t_f))\end{array}\right)$$

[assumption (d > 0)]

$$=\mathbf{R_{ct}}\left(\begin{array}{c}\mathbf{true}\wedge\\ \neg\,((P^f_f\wedge((Q_f\wedge\frown/tr'=\frown/tr\wedge wait');\,tr'-tr=\langle\langle\rangle\rangle));\,\mathbf{R1_{ct}}(\mathbf{true}))\wedge\\ \neg\,((Q^f_f\wedge((P_f\wedge\frown/tr'=\frown/tr\wedge wait');\,tr'-tr=\langle\langle\rangle\rangle));\,\mathbf{R1_{ct}}(\mathbf{true}))\wedge\\ \neg\,(P^f_f\wedge((Q_f\wedge\frown/tr'=\frown/tr\wedge wait');\,head(tr'-tr)\neq\langle\rangle))\wedge\\ \neg\,(Q^f_f\wedge((P_f\wedge\frown/tr'=\frown/tr\wedge wait');\,head(tr'-tr)\neq\langle\rangle))\wedge\\ \mathbf{true}\\ \vdash\\ (P^t_f\wedge Q^t_f\wedge wait'\wedge\frown/tr'=\frown/tr)\vee(Diff(P^t_f,Q^t_f)\wedge(P^t_f\vee Q^t_f))\end{array}\right)$$

[propositional calculus]

$$=\mathbf{R_{ct}}\left(\begin{array}{c}
\neg\,((P_f^f \wedge ((Q_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait');\, tr' - tr = \langle\langle\rangle\rangle));\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\
\neg\,((Q_f^f \wedge ((P_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait');\, tr' - tr = \langle\langle\rangle\rangle));\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\
\neg\,(P_f^f \wedge ((Q_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait')\,;\, head(tr' - tr) \neq \langle\rangle)) \wedge \\
\neg\,(Q_f^f \wedge ((P_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait')\,;\, head(tr' - tr) \neq \langle\rangle)) \wedge \\
\vdash \\
(P_f^t \wedge Q_f^t \wedge wait' \wedge {}^\frown/tr' = {}^\frown/tr) \vee (\mathit{Diff}(P_f^t, Q_f^t) \wedge (P_f^t \vee Q_f^t))
\end{array}\right)$$

Here, we focus on the precondition, and work out the first clause.

(1)

$\neg\,((P_f^f \wedge ((Q_f \wedge {}^\frown/tr' = {}^\frown/tr \wedge wait');\, tr' - tr = \langle\langle\rangle\rangle));\, \mathbf{R1_{ct}}(\mathbf{true}))$

[$P_f^f = (wait\ \ d; A_1)_f^f$ ane $Q_f = (wait\ \ d; A_2)^f$]

$$=\neg\left(\begin{array}{l}
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\end{array}\right);\mathbf{R1_{ct}}({A_1}_f^f)\wedge\\
\left(\left(\left(\begin{array}{l}
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\end{array}\right);\mathbf{R1_{ct}}({A_2}_f^f)\vee\\
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr<d\wedge\\ wait'\end{array}\right);\mathbf{R1_{ct}}(\mathit{II}\wedge wait)\vee\\
\mathbf{R1_{ct}}\left(\begin{array}{l}\neg\ wait'\wedge\\ {}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\wedge\\ state'=state\end{array}\right);\mathbf{R1_{ct}}(\neg\ wait\wedge {A_2}_f^t)
\end{array}\right)\wedge{}^\frown/tr'={}^\frown/tr\wedge wait'\right);\ tr'-tr=\langle\langle\rangle\rangle\right)
\end{array}\right);\mathbf{R1_{ct}}(\mathbf{true})$$

(1.1) if $A_1$ diverges immediately, or ${A_1}_f^f = (({A_1}_f^f \wedge tr' = tr);\, \mathbf{R1_{ct}}(\mathbf{true}))$

$$=\neg\,(\mathbf{R1_{ct}}({}^\frown/tr'={}^\frown/tr\wedge\#tr'-\#tr=d);\ ((\mathbf{R1_{ct}}({A_1}_f^f)\wedge tr'=tr);\mathbf{R1_{ct}}(\mathbf{true}))\vee\mathbf{false}\vee\mathbf{false})$$

[Propositional calculus]

$$=\neg\,(\mathbf{R1_{ct}}({}^\frown/tr'={}^\frown/tr\wedge\#tr'-\#tr=d);\ ((\mathbf{R1_{ct}}({A_1}_f^f)\wedge tr'=tr);\mathbf{R1_{ct}}(\mathbf{true})))$$

(1.2) if $A_1$ does not diverge immediately, or ${A_1}_f^f \neq (({A_1}_f^f \wedge tr' = tr);\, \mathbf{R1_{ct}}(\mathbf{true}))$

$$=\neg\left(\begin{array}{l}
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\end{array}\right);\mathbf{R1_{ct}}({A_1}_f^f)\wedge\\
\left(\left(\left(\begin{array}{l}
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\end{array}\right);\mathbf{R1_{ct}}({A_2}_f^f)\vee\\
\mathbf{false}\vee\\
\mathbf{R1_{ct}}\left(\begin{array}{l}\neg\ wait'\wedge\\ {}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\wedge\\ state'=state\end{array}\right);\mathbf{R1_{ct}}(\neg\ wait\wedge {A_2}_f^t)
\end{array}\right)\wedge{}^\frown/tr'={}^\frown/tr\wedge wait'\right);\ tr'-tr=\langle\langle\rangle\rangle\right)
\end{array}\right);\mathbf{R1_{ct}}(\mathbf{true})$$

[$state'$ and $wait'$ are not interested in a divergence]

$$=\neg\left(\begin{array}{l}
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\end{array}\right);\mathbf{R1_{ct}}({A_1}_f^f)\wedge\\
\left(\left(\left(\begin{array}{l}
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\end{array}\right);\mathbf{R1_{ct}}({A_2}_f^f)\vee\\
\mathbf{R1_{ct}}\left(\begin{array}{l}{}^\frown/tr'={}^\frown/tr\wedge\\ \#tr'-\#tr=d\end{array}\right);\mathbf{R1_{ct}}({A_2}_f^t)
\end{array}\right)\wedge{}^\frown/tr'={}^\frown/tr\right);\ tr'-tr=\langle\langle\rangle\rangle\right)
\end{array}\right);\mathbf{R1_{ct}}(\mathbf{true})$$

[relational calculus]

$$=\neg\,(\mathbf{R1_{ct}}\begin{pmatrix} \frown/tr' = \frown/tr \wedge \\ \#tr' - \#tr = d \end{pmatrix}; (\mathbf{R1_{ct}}(A1_f^f) \wedge (((A2_f^f \vee A2_f^t) \wedge \frown/tr = \frown/tr); tr' - tr = \langle\langle\rangle\rangle)); \mathbf{R1_{ct}}(\mathbf{true}))$$

[predicate calculus]

$$=\neg\,(\mathbf{R1_{ct}}\begin{pmatrix} \frown/tr' = \frown/tr \wedge \\ \#tr' - \#tr = d \end{pmatrix}; (\mathbf{R1_{ct}}(A1_f^f) \wedge ((A2_f \wedge \frown/tr = \frown/tr); tr' - tr = \langle\langle\rangle\rangle)); \mathbf{R1_{ct}}(\mathbf{true}))$$

therefore, (1.1) and (1.2)

$$=\neg\left(\mathbf{R1_{ct}}\begin{pmatrix} \frown/tr' = \frown/tr \wedge \\ \#tr' - \#tr = d \end{pmatrix}; \begin{pmatrix} (A1_f^f \wedge tr' = tr); \mathbf{R1_{ct}}(\mathbf{true}) \vee \\ (\mathbf{R1_{ct}}(A1_f^f) \wedge ((A2_f \wedge \frown/tr = \frown/tr); tr' - tr = \langle\langle\rangle\rangle)); \mathbf{R1_{ct}}(\mathbf{true}) \end{pmatrix}\right)$$

(2)the second clause in the precondition has the similar result

$$\neg\,((Q_f^f \wedge ((P_f \wedge \frown/tr' = \frown/tr \wedge wait'); tr' - tr = \langle\langle\rangle\rangle)); \mathbf{R1_{ct}}(\mathbf{true}))$$

$$=\neg\left(\mathbf{R1_{ct}}\begin{pmatrix} \frown/tr' = \frown/tr \wedge \\ \#tr' - \#tr = d \end{pmatrix}; \begin{pmatrix} (A2_f^f \wedge tr' = tr); \mathbf{R1_{ct}}(\mathbf{true}) \vee \\ (\mathbf{R1_{ct}}(A2_f^f) \wedge ((A1_f \wedge \frown/tr = \frown/tr); tr' - tr = \langle\langle\rangle\rangle)); \mathbf{R1_{ct}}(\mathbf{true}) \end{pmatrix}\right)$$

(3) $and$ (4), the third and fourth clauses also have the similar result, $P$ (or $Q$) performs either a visivle event immediately or an event later.

$$\begin{pmatrix} \neg\,(P_f^f \wedge ((Q_f \wedge \frown/tr' = \frown/tr \wedge wait'); head(tr' - tr) \neq \langle\rangle)) \wedge \\ \neg\,(Q_f^f \wedge ((P_f \wedge \frown/tr' = \frown/tr \wedge wait'); head(tr' - tr) \neq \langle\rangle)) \end{pmatrix}$$

$$=\neg\left(\mathbf{R1_{ct}}\begin{pmatrix} \frown/tr' = \frown/tr \wedge \\ \#tr' - \#tr = d \end{pmatrix}; \begin{pmatrix} (A1_f^f \wedge ((A2_f \wedge \frown/tr' = \frown/tr \wedge wait'); head(tr' - tr) \neq \langle\rangle)) \vee \\ (A2_f^f \wedge ((A1_f \wedge \frown/tr' = \frown/tr \wedge wait'); head(tr' - tr) \neq \langle\rangle)) \vee \\ ((A1_f^f \vee A2_f^f) \wedge head(diff(tr', tr)) \neq \langle\rangle) \end{pmatrix}\right)$$

finally, combining (1),(2),(3) and (4), the precondition of $Wait\ d; A_1 \Box Wait\ d; A_2$ is

$$=\neg\,(\mathbf{R1_{ct}}(\frown/tr' = \frown/tr \wedge \#tr' - \#tr = d); (A_1 \Box A_2)_f^f)$$

For the postcondition of $Wait\ d; A_1 \Box Wait\ d; A_2$, we can easily have the following result.

$$(\mathbf{R1_{ct}}(\frown/tr' = \frown/tr \wedge \#tr' - \#tr = d); (A_1 \Box A_2)_f^t)$$

As a result, we finish the proof that $Wait\ d; (A_1 \Box A_2) = Wait\ d; A_1 \Box Wait\ d; A_2$.

□

## References

[1] A. Cavalcanti, A. Sampaio, and J. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2-3):146–181, 2003.

[2] J. S. Fitzgerald, P. G. Larsen, and M. Verhoef. Vienna development method. In B. W. Wah, editor, *Wiley Encyclopedia of Computer Science and Engineering*. John Wiley & Sons, Inc., 2008.

[3] L. Freitas. *Model-checking Circus*. PhD thesis, Department of Computer Science, The University of York, UK, 2005. YCST-2005/11.

[4] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.

[5] C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming.* Prentice-Hall International, 1998.

[6] C. Morgan. *Programming from specifications.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.

[7] A. W. Roscoe. *The Theory and Practice of Concurrency.* Prentice-Hall International, 1998.

[8] M. Saaltink. The Z/EVES system. In *ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 72–85, London, UK, 1997. Springer-Verlag.

[9] S. A. Schneider. *Concurrent and real-time systems: the CSP approach.* John Wiley & Sons, 1999.

[10] A. Sherif, A. L. C. Cavalcanti, H. Jifeng, and A. C. A. Sampaio. A process algebraic framework for specification and validation of real-time systems. *Formal Aspects of Computing*, 22(2):153 – 191, 2010.

[11] K. Wei, J. Woodcock, and A. Cavalcanti. New Circus Time. Technical report, Computer Science, University of York, UK, 2012. Avaliable at http://www.cs.york.ac.uk/circus/hijac/publication.html.

[12] J. Woodcock and A. Cavalcanti. *CML Definition 2.* COMPASS Deliverables D23.3 2013. Avaliable at http://www.compass-research.eu/deliverables.html.

[13] J. Woodcock, A. Cavalcanti, J. Fitzgeraldt, P. Larsen, A. Miyazawa, and S. Perry. Features of CML: a formal modelling language for Systems of Systems. In *7th International Conference on System of Systems Engineering (SoSE)*, 2012.

[14] J. Woodcock and J. Davies. *Using Z: Specification, Refinement and Proof.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[15] J. C. P. Woodcock and A. L. C. Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, *IWFM'01: 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.

[16] J. C. P. Woodcock, A. L. C. Cavalcanti, and L. Freitas. Operational Semantics for Model Checking Circus. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM 2005: Formal Methods, International Symposium of Formal Methods Europe*, volume 3582 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2005.