Circus Time with Reactive Designs

Kun Wei, Jim Woodcock, and Ana Cavalcanti

Department of Computer Science, University of York, York, YO10 5GH, UK
{kun.wei,jim.woodcock,ana.cavalcanti}@york.ac.uk

Abstract. The UTP theories for CSP and *Circus* can be built by the combination of the theories of designs and reactive processes. Defining the CSP operators using reactive design provides a more concise, readable and uniform UTP semantics, and, more importantly, exposes the pre-postcondition semantics of the operators. For *Circus Time*, a few operators have been defined as reactive designs, but some important operators are still be considered. In this paper, we develop the reactive design semantics of sequential composition, hiding and recursion within *Circus Time*, and show how to prove some subtle laws using the new semantics.

Keywords: Circus Time; UTP; reactive designs

1 Introduction

Circus [3, 17, 18] is a comprehensive combination of Z [15], CSP [5, 10] and Morgan's refinement calculus [8], 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. *Circus Time* is an extension of a subset of *Circus* with some time operators added to the notion of actions in *Circus*. The semantics of *Circus Time* is defined using the UTP by introducing time observation variables. The *Circus Time* UTP theory is a discrete time model, and time operators are very similar to that in Timed CSP [11]. Compared to the original *Circus Time* [12] and Timed CSP, we have recently developed a new version of *Circus Time* [13] that provides more time operators such as deadlines. Using various languages in the *Circus* family, we have proposed an approach in [1] for stepwise development of safety-critical Java programs [7]. In the new *Circus Time* theory, besides some new time operators, each action is expressed as a reactive design for a more concise, readable and uniform UTP semantics. In the UTP, Hoare and He provide many sub-theories by adopting different healthiness conditions. For example, the theory of designs can describe some sequential programming languages, and the theory of reactive processes allows communications between processes. The theory of CSP is traditionally built upon the theory of reactive processes by imposing extra healthiness conditions. However, there is no uniform pattern of the semantics for CSP primitive processes and various operators. Hoare and He have, however, proposed an approach to generate the theory of CSP by embedding the theory of designs in the theory of reactive processes.

This means that each process in CSP can be expressed as a reactive design. The importance of this semantics is that it exposes the pre-postcondition semantics.

The work in [2, 9] have provided the reactive design semantics to some operators in CSP. On the other hand, sequential composition, recursion, hiding and so on are still to be considered. In this paper, based on our new *Circus Time* model, we develop the reactive design semantics of these operators in a timed environment, and also demonstrate how we can easily to prove some very tricky laws using the new semantics. This paper has the following structure. Section 2 gives a brief introduction to *Circus Time* and related UTP theories. We show how to deduce the reactive design semantics of those operators from their original UTP definitions in Section 3. We then give a demonstration to show how to prove some algebraic laws using the new semantics in Section 4. Finally in Section 5 we present some conclusions and summarize future work. We assume knowledge of CSP.

2 UTP and *Circus Time*

In the UTP, a relation P is a predicate with an alphabet αP, composed of *undashed* variables $(a, b, ...)$ and *dashed* variables $(a', x', ...)$. The former, written as $in\alpha P$, stands for initial observations, and the latter, $out\alpha P$, for intermediate or final observations. The relation is then called *homogeneous* if $out\alpha P = in\alpha P'$, where $in\alpha P'$ is simply obtained by putting a dash on 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 $(x := e)$, non-determinism $(P \sqcap Q)$ 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 follows:

$$S \sqsubseteq P \;\; \textit{iff} \;\; [P \Rightarrow S]$$

where 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 behaves arbitrarily ($[\mathbf{true} \sqsubseteq P]$), and the top element is the strongest relation **false**, which behaves miraculously and satisfies any specification ($[P \sqsubseteq \mathbf{false}]$). The bottom and top elements in this complete lattice are usually called **CHAOS** and **Miracle** respectively.

2.1 Designs

A design in the UTP is a relation that can be expressed as a pre-postcondition pair in combination with a boolean variable, called ok. In designs, ok records that

the program has started, and ok' records that it has terminated. If a precondition P and a postcondition Q are predicates, a design with P and Q, written as $P \vdash Q$, is defined as follows:

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

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. The healthy relations are the fixed points of these functions. There are four healthiness conditions identified by Hoare and He in the theory of designs and here we introduce only two of them.

$$\textbf{H1} \quad P = ok \Rightarrow P \qquad \textbf{H2} \quad [P[\mathit{false}/ok'] \Rightarrow P[\mathit{true}/ok']]$$

The first healthiness 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 predicate is **H1** and **H2** if, and only if, it is a design; the proof is in [6]. A useful law about designs, which is used later, is given as below.

Law 1 *Suppose P and Q are predicates and b is a condition,*

$$((P_1 \vdash P_2) \lhd b \rhd (Q_1 \vdash Q_2)) = ((P_1 \lhd b \rhd Q_1) \vdash (P_2 \lhd b \rhd Q_2))$$

This states that conditionals distribute through designs. A proof of this law can be found in [6].

The purpose of the theory of designs is to exclude relations that do not satisfy the zero laws, $\textbf{true}\,;\, P = \textbf{true} = P\,;\, \textbf{true}$. For example, the relations that satisfy the equation $\textbf{true}\,;\, P = P$ should not be included in the theory of designs. The program **true** behaves arbitrarily. For instance, the least fixed-point semantics of a non-terminating loop in the theory of relations is **true**, and it, when followed by an assignment like $x := c$, behaves like the assignment. In practice, it means that a program can recover from the non-terminating loop. For a tutorial introduction to designs, the reader is referred to [6, 16].

2.2 Reactive Processes

A reactive process in the UTP 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. For example, if $wait'$ is true, then the process is in an intermediate state. If *wait* is true, it denotes an intermediate observation of its predecessor. Thus, we are able to represent any states of a process by combining the values of ok and *wait*. If ok' is false, the process diverges. If ok' is true, the state of the process

depends on the value of $wait'$. If $wait'$ is true, the process is in an intermediate state; otherwise it has successfully terminated. Similarly, the values of undashed variables represent the states of a process's predecessor.

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 until the next 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 in the next observation. The reactive identity, II_{rea}, is defined as follows:

$$\mathit{II}_{rea} \mathrel{\widehat{=}} (\neg\, ok \wedge tr \leq tr') \vee (ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref)$$

As a result, a reactive process must satisfy the following healthiness conditions:

R1 $P = P \wedge tr \leq tr'$

R2 $P(tr, tr') = P(\langle\rangle, tr' - tr)$

R3 $P = \mathit{II}_{rea} \lhd wait \rhd P$

If a relation P describes a reactive process behaviour, **R1** states that it never changes history, or the trace is always increasing. 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 (II_{rea}) if it is in a waiting state ($wait = true$) of its predecessor. 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 $\mathbf{R}$. Since each of $\mathbf{R}i$ is idempotent and any two of them commuted, $\mathbf{R}$ is also idempotent. For a more detailed introduction to the theory of reactive designs, the reader is referred to the tutorial [2].

2.3 CSP Processes

In the UTP, the theory of CSP is built by applying extra 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:

CSP1 $P = P \vee (\neg\, ok \wedge tr \leq tr')$ **CSP2** $P = P\,;\, J$

where $J = (ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref$. The first healthiness condition requires that, in case of divergence of the predecessor, the extension of the trace and should be the only guaranteed property. The second one means that P cannot require non-termination, so that it is always possible to terminate. The CSP theory introduced in the UTP book is different from any standard models of CSP [5, 10] which have more restrictions or satisfy more healthiness conditions. There are more healthiness conditions, **CSP3-CSP5**, given in UTP to further restrict behaviours of reactive processes.

A CSP process can also be obtained by applying the healthiness condition $\mathbf{R}$ to a design. This follows from the theorem in [6], that, for every CSP process P, $P = \mathbf{R}(\neg\, P_f^f \vdash P_f^t)$. 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 $\mathbf{R}$. Note that P_b^a is an abbreviation of $P[a, b/ok', wait]$, and it is often used in this paper. Motivated by the above theorem, the work in [2, 9] provide reactive design definitions for some constructs of CSP such as $STOP$, $SKIP$, $CHAOS$, external choice, and so on. The reactive design definitions of more operators, such as sequential composition, hiding and recursion, will be developed in this paper.

2.4 *Circus Time*

We give a brief introduction to *Circus Time* because the reactive design semantics is developed within this timed model presented in this paper. In *Circus Time*, an action is described as an alphabetised predicate whose observational variables include ok, $wait$, tr, ref, $state$ and their dashed counterparts.

- $ok, ok' : Boolean$
- $wait, wait' : Boolean$
- $state, state' : N \pfun Value$
- $tr, tr' : seq^+(seq\ Event)$
- $ref, ref' : seq^+(\mathbb{P}\ Event)$

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 (seq^+), 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 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 $state'$ records a set of local variables and their values. N is a set of names of these variables.

Both the original *Circus Time* and Timed CSP use the concept of failures, each of which consists of a trace and a refusal. This structure, however, is hard to manipulate: a trace is no longer a sequence of events, but a sequence of pairs containing a sequence of events and a refusal set. In the new *Circus Time* model, we split a failure as shown above to record sequences of traces and a refusals, and use their indices (which start at 1) to match related pairs. However, the decomposition of the failures results in a little bit inconvenience, since we have to ensure the equality of the lengths of tr and ref, or tr' and ref'. This is achieved by imposing an extra constraint on the healthiness conditions.

Although a trace in the new model is a sequence of sequences, the standard operations on sequences defined in Z can still be used here such as $head$, $tail$, $front$, $last$, $\#$(length), $\frown$(concatenation), $\frown/$(flattening), $-$(difference) and $\leq$(prefix). Additionally, it is unnecessary that $last(tr) = tr'(\#tr)$. An expanding relation between traces is defined as follows, requiring that $front(tr)$ and $last(tr)$ are the prefixes of tr' and $tr'(\#tr)$ respectively.

$$tr \preccurlyeq tr' \mathrel{\widehat{=}} front(tr) \leq tr' \wedge last(tr) \leq tr'(\#tr) \tag{1}$$

An action in *Circus Time* must satisfy the healthiness conditions, $\mathbf{R1_t}$-$\mathbf{R3_t}$ and $\mathbf{CSP1_t}$-$\mathbf{CSP5_t}$. These healthiness conditions have similar meanings as 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, $\mathbf{R1_t}$ and $\mathbf{R3_t}$, as follows (the properties including other healthiness conditions are usually straightforward to be proven). The detailed introduction to other healthiness conditions can be found in [13].

$$\mathbf{R1_t}(X) \mathrel{\widehat{=}} X \wedge RT \qquad \mathbf{R3_t}(X) \mathrel{\widehat{=}} \mathit{II}_t \lhd wait \rhd X$$

where the predicate RT, II^{-ok} (the identity without ok) and the timed reactive identity II_t are given as

$$RT \mathrel{\widehat{=}} tr \preccurlyeq tr' \wedge front(ref) \leq ref' \wedge \#diff(tr', tr) = \#(ref' - front(ref))$$
$$\mathit{II}^{-ok} \mathrel{\widehat{=}} \big(tr' = tr \wedge front(ref') = front(ref) \wedge wait' = wait \wedge state' = state \big)$$
$$\mathit{II}_t \mathrel{\widehat{=}} (\neg\, ok \wedge RT) \vee (ok' \wedge \mathit{II}^{-ok})$$

Note that we impose a restriction, $\#ref' = \#tr'$ and $\#ref = \#tr$, to ensure that the lengths of ref and ref' are always the same as those of tr and tr' respectively. This is a consequence of splitting traces and refusals as already explained. Rather than recording the refusals only at the end of traces in CSP, *Circus Time* records the refusals at the end of each time unit. In other words, we need to keep the history of refusals. However, we are usually not interested in the refusals of the last time unit after an action terminates. Therefore, we use $front(ref) \leq ref'$ and $front(ref') = front(ref)$ in these healthiness conditions, instead of $ref \leq ref'$ and $ref' = ref$ because we have to maintain the consistency among $\mathbf{R_t} \mathrel{\widehat{=}} \mathbf{R1_t} \circ \mathbf{R2_t} \circ \mathbf{R3_t}$ and $\mathbf{CSP1_t}$-$\mathbf{CSP5_t}$. In addition, by means of a result in [13], each action in *Circus Time* can also be described as a reactive design.

Theorem 1. *For every action P in Circus Time,*

$$P = \mathbf{R_t}(\neg\, P_f^f \vdash P_f^t)$$

Another useful law about $\mathbf{R1_t}$, which is used later, is given and its detailed proof can be found in [13].

Law 2

$$\mathbf{R1_t}(P \vee Q) = \mathbf{R1_t}(P) \vee \mathbf{R1_t}(Q)$$

The full syntax, definitions and detailed explanations of *Circus Time* can be found in [13]. Here, we briefly introduce some operators that are used in the following sections. The action *Skip* terminates immediately without changing

anything, *Chaos* is the worst action (the bottom element in the refinement ordering) whose behaviour is arbitrary, but satisfies $\mathbf{R_t}$. The action *Miracle* is the top element that expresses an unstarted process. This action is not included in the standard failures-divergences model of CSP. The definition and properties of *Miracle* are discussed in Section 4. The delay action *Wait* d does nothing except that it requires d time units to elapse before it terminates. The sequential composition $P\,;\,Q$ behaves like P until P terminates, and then behaves as Q. The prefix action $c.e \rightarrow P$ is usually constructed by a composition of a simple prefix and P itself, written as $(c.e \rightarrow \mathit{Skip})\,;\,P$. The hiding action $P \setminus CS$ will behave like P, but the events within the set CS become invisible. The recursive action $\mu\, X \bullet P$ behaves like P with every occurrence of the variable X in P representing a recursive invocation. The recursive call takes no time.

3 Reactive Designs for *Circus Time*

In this section, we calculate the reactive design semantics of sequential composition, hiding and recursion from their original UTP semantics (which have been slightly changed to contain time) in terms of Theorem 1.

3.1 Sequential Composition

The definition of sequential composition in the UTP is given as follows:

$$P\,;\,Q \mathrel{\widehat{=}} \exists\, obs_0 \bullet P[obs_0/obs'] \wedge Q[obs_0/obs]$$

To deduce the reactive design of sequential composition, we first give some auxiliary laws that have been proven in [13].

Law 3 *Suppose P is $\mathbf{R1_t}$ and $\mathbf{CSP1_t}$,*

$$\mathbf{R1_t}(\neg\, ok)\,;\,P = \mathbf{R1_t}(\neg\, ok)$$

Law 4 *Suppose P is a predicate and Q is $\mathbf{R1_t}$ and $\mathbf{R3_t}$,*

$$\mathbf{R3_t}(P)\,;\,Q = \mathbf{R3_t}(P\,;\,Q)$$

Below, we characterise the behaviour of the sequential composition of two $\mathbf{R1_t}$-healthy predicates.

Law 5 *Suppose P, Q, R and S are predicates (ok,ok' are not in αP, αQ, αR and αS),*

$$\mathbf{R1_t}(P \vdash Q)\,;\,\mathbf{R1_t}(R \vdash S) = \mathbf{R1_t}\left(\begin{array}{c}\neg\,(\mathbf{R1_t}(\neg\, P)\,;\,\mathbf{R1_t}(\mathit{true})) \wedge \neg\,(\mathbf{R1_t}(Q)\,;\,\mathbf{R1_t}(\neg\, R)) \\ \vdash \\ \mathbf{R1_t}(Q)\,;\,\mathbf{R1_t}(S)\end{array}\right)$$

Proof.

$$
\begin{aligned}
& \mathbf{R1_t}(P \vdash Q) \,;\, \mathbf{R1_t}(R \vdash S) && \text{[def of design]} \\
=& \mathbf{R1_t}(\neg\, ok \vee \neg\, P \vee (ok' \wedge Q)) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \\
& && \text{[Law 2 and rel. cal.]} \\
=& \mathbf{R1_t}(\neg\, ok) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \vee && \text{[Law 3]} \\
& \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \vee \\
& \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \\
=& \mathbf{R1_t}(\neg\, ok) \vee \mathbf{R1_t}(\neg\, P) \,;\, R1_t(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \vee \\
& \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \\
& && \text{[def of ; and case split on } ok\text{]} \\
=& \mathbf{R1_t}(\neg\, ok) \vee && \text{[propositional calculus]} \\
& \mathbf{R1_t}(\neg\, P)[\mathit{false}/ok'] \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S))[\mathit{false}/ok] \vee \\
& \mathbf{R1_t}(\neg\, P)[\mathit{true}/ok'] \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S))[\mathit{true}/ok] \vee \\
& \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \\
=& \mathbf{R1_t}(\neg\, ok) \vee \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee && \text{[rel. calculus and Law 2]} \\
& \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\neg\, R \vee (ok' \wedge S)) \vee \\
& \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \\
=& \mathbf{R1_t}(\neg\, ok) \vee \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee && \text{[propositional calculus]} \\
& \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok \vee \neg\, R \vee (ok' \wedge S)) \\
=& \mathbf{R1_t}(\neg\, ok) \vee \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee && \text{[rel. calculus and Law 2]} \\
& \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok \vee (ok \wedge \neg\, R) \vee (ok \wedge ok' \wedge S)) \\
=& \mathbf{R1_t}(\neg\, ok) \vee && \text{[def of ;]} \\
& \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok) \vee \\
& \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(ok \wedge \neg\, R) \vee \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(ok \wedge ok' \wedge S) \\
=& \mathbf{R1_t}(\neg\, ok) \vee && \text{[Law 2 and prop. calculus]} \\
& \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok) \vee \\
& \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(ok \wedge \neg\, R) \vee \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(ok \wedge ok' \wedge S) \\
=& \mathbf{R1_t}(\neg\, ok) \vee \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee \mathit{false} \vee && \text{[def of ;]} \\
& \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee \mathbf{R1_t}(ok' \wedge Q) \,;\, \mathbf{R1_t}(\neg\, ok) \vee \\
& \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(\neg\, R) \vee \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(ok' \wedge S) \\
=& \mathbf{R1_t}(\neg\, ok) \vee \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee && \text{[rel. calculus]} \\
& \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(\neg\, R) \vee \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(ok' \wedge S) \\
=& \mathbf{R1_t}(\neg\, ok) \vee \mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true}) \vee && \text{[def of design]} \\
& \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(\neg\, R) \vee (ok' \wedge (\mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(S))) \\
=& \mathbf{R1_t} \left(\begin{array}{c} \neg\, (\mathbf{R1_t}(\neg\, P) \,;\, \mathbf{R1_t}(\mathit{true})) \wedge \neg\, (\mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(\neg\, R)) \vdash \\ \mathbf{R1_t}(Q) \,;\, \mathbf{R1_t}(S) \end{array} \right)
\end{aligned}
$$

However, the timed reactive identity II_t is not a design, and hence $\mathbf{R3_t}(P)$ is not, even if P is a design. Therefore, Woodcock in [14] introduces a new healthiness condition to replace **R3**, in order to make a design behave like the identity design when waiting. And here we have a similar healthiness condition, $\mathbf{R3j_t}(P) \mathrel{\widehat{=}} \mathit{II}_D \lhd \mathit{wait} \rhd P$ where $\mathit{II}_D = \mathit{true} \vdash \mathit{II}$. We also have a useful law about the new healthiness condition whose proof can be found in [13].

Law 6

$$\mathbf{R1_t} \circ \mathbf{R3j_t} = \mathbf{R1_t} \circ \mathbf{R3_t}$$

Finally, we are ready to deduce the reactive design of sequential composition from its original definition.

Theorem 2. *Suppose P and Q are two Circus Time actions,*

$$P \,;\, Q = \mathbf{R_t} \left(\begin{array}{c} \neg\, (\mathbf{R1_t}(P_f^f) \,;\, \mathbf{R1_t}(\mathit{true})) \wedge \neg\, (\mathbf{R1_t}(P_f^t) \,;\, \mathbf{R1_t}(\neg\, \mathit{wait} \wedge Q_f^f)) \\ \vdash \\ \mathbf{R1_t}(P_f^t) \,;\, \mathbf{R1_t}(\mathit{II} \lhd \mathit{wait} \rhd Q_f^t) \end{array} \right)$$

Proof.

$$\begin{array}{ll}
& P \,;\, Q \qquad [\text{Theorem 1}] \\
= & \mathbf{R_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R_t}(\neg\, Q_f^f \vdash Q_f^t) \qquad [\text{def of } \mathbf{R_t}] \\
= & \mathbf{R3_t} \circ \mathbf{R1_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R1_t} \circ \mathbf{R3_t}(\neg\, Q_f^f \vdash Q_f^t) \qquad [\text{Law 4}] \\
= & \mathbf{R3_t} \circ (\mathbf{R1_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R1_t} \circ \mathbf{R3_t}(\neg\, Q_f^f \vdash Q_f^t)) \qquad [\text{Law 6}] \\
= & \mathbf{R3_t} \circ (\mathbf{R1_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R1_t} \circ \mathbf{R3j_t}(\neg\, Q_f^f \vdash Q_f^t)) \qquad [\text{def of } \mathbf{R3j_t}] \\
= & \mathbf{R3_t} \circ (\mathbf{R1_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R1_t}(\mathit{II}_D \lhd \mathit{wait} \rhd \neg\, Q_f^f \vdash Q_f^t)) \qquad [\text{def of } \mathit{II}_D] \\
= & \mathbf{R3_t} \circ (\mathbf{R1_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R1_t}(\mathit{true} \vdash \mathit{II} \lhd \mathit{wait} \rhd \neg\, Q_f^f \vdash Q_f^t)) \qquad [\text{Law 1}] \\
= & \mathbf{R3_t} \circ (\mathbf{R1_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R1_t}((\mathit{true} \lhd \mathit{wait} \rhd \neg\, Q_f^f) \vdash (\mathit{II} \lhd \mathit{wait} \rhd Q_f^t))) \\
& \qquad [\text{rel. cal.}] \\
= & \mathbf{R3_t} \circ (\mathbf{R1_t}(\neg\, P_f^f \vdash P_f^t) \,;\, \mathbf{R1_t}((\mathit{wait} \vee \neg\, Q_f^f) \vdash (\mathit{II} \lhd \mathit{wait} \rhd Q_f^t))) \\
& \qquad [\text{Law 5}] \\
= & \mathbf{R3_t} \circ \mathbf{R1_t} \left(\begin{array}{c} \neg\, (\mathbf{R1_t}(P_f^f) \,;\, \mathbf{R1_t}(\mathit{true})) \wedge \neg\, (\mathbf{R1_t}(P_f^t) \,;\, \mathbf{R1_t}(\neg\, \mathit{wait} \wedge Q_f^f)) \\ \vdash \\ \mathbf{R1_t}(P_f^t) \,;\, \mathbf{R1_t}(\mathit{II} \lhd \mathit{wait} \rhd Q_f^t) \end{array} \right) \\
& \qquad [\text{def of } \mathbf{R_t}] \\
= & \mathbf{R_t} \left(\begin{array}{c} \neg\, (\mathbf{R1_t}(P_f^f) \,;\, \mathbf{R1_t}(\mathit{true})) \wedge \neg\, (\mathbf{R1_t}(P_f^t) \,;\, \mathbf{R1_t}(\neg\, \mathit{wait} \wedge Q_f^f)) \\ \vdash \\ \mathbf{R1_t}(P_f^t) \,;\, \mathbf{R1_t}(\mathit{II} \lhd \mathit{wait} \rhd Q_f^t) \end{array} \right)
\end{array}$$

This theorem shows that, if P does not diverge and Q does not diverge after P terminates, $P \,;\, Q$ behaves like the sequential composition of the terminations of P and Q.

3.2 Hiding

Similar to the CSP hiding operator in the UTP [6], the hiding operator in *Circus Time* is defined as follows:

$$
\begin{aligned}
&P \setminus CS \mathrel{\widehat=} \mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \land L_t)\,;\, Skip \\
&L_t \mathrel{\widehat=} diff(tr', tr) = diff(s, tr) \downarrow_t (\Sigma - CS) \land \\
&\qquad\qquad r - front(ref) = ((ref' - front(ref)) \cup_t CS) \\
&diff(tr', tr) \mathrel{\widehat=} \langle tr'(\#tr) - last(tr)\rangle \frown tail(tr' - front(tr)) \qquad (2)
\end{aligned}
$$

where *diff* is the difference of two traces, and two special operators, $\downarrow_t$ and $\cup_t$, are defined to restrict timed traces and complement refusals respectively.

$$
\begin{aligned}
&tr_1 = (tr_2 \downarrow_t CS) \Leftrightarrow \forall\, i : 1..\#tr_1 \bullet tr_1(i) = (tr_2(i) \downarrow CS) \land \#tr_1 = \#tr_2 \\
&ref_1 = (ref_2 \cup_t CS) \Leftrightarrow \forall\, i : 1..\#ref_1 \bullet ref_1(i) = (ref_2(i) \cup CS) \land \#ref_1 = \#ref_2
\end{aligned}
$$

Clearly, this definition is not a reactive design. As usual, three useful laws are given and their proof can be found in [13].

Law 7

$$Skip^f = \mathbf{R1_t}(\neg\, ok)$$

Law 8

$$Skip^t = \mathbf{R1_t}(\neg\, ok) \lor (ok \land \mathit{II})$$

Law 9

$$\mathbf{R1_t}(\exists\, s, r \bullet L_t) = \exists\, s, r \bullet L_t$$

Law 10 *Suppose P is a Circus Time action,*

$$(P \setminus CS)^f_f = (\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \land L_t)\,;\, \mathbf{R1_t}(true)$$

Proof.

$$
\begin{aligned}
&(P \setminus CS)^f_f && \text{[def of } \setminus\text{]} \\
=&(\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \land L_t)\,;\, Skip)^f_f && \text{[relational calculus]} \\
=&(\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \land L_t))_f\,;\, Skip^f && \text{[Law 7]} \\
=&(\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \land L_t))_f\,;\, \mathbf{R1_t}(\neg\, ok) && \text{[case split on } ok'\text{]} \\
=&((\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \land L_t))^f_f\,;\, \mathbf{R1_t}(\neg\, ok)) \lor && \text{[relational calculus]} \\
&((\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \land L_t))^t_f\,;\, \mathbf{R1_t}(\neg\, ok)) \\
=&((\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \land L_t))^f_f\,;\, \mathbf{R1_t}(\neg\, ok)) \lor false \\
& && [\mathbf{R3_t} \text{ and } wait \text{ is } false]
\end{aligned}
$$

$$= (\mathbf{R1_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t))^f_f \,;\, \mathbf{R1_t}(\neg\, ok)$$
[P and L_t, Law 9 and predicate calculus]

$$= (\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t)^f_f \,;\, \mathbf{R1_t}(\neg\, ok)$$ [relational calculus]

$$= (\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t)^f_f \,;\, \mathbf{R1_t}(true)$$ [predicate calculus]

$$= (\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true)$$

Law 11 *Suppose P is a Circus Time action,*

$$(P \setminus E)^t_f = \left(\begin{array}{c}(\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true) \\ \vee\, (\exists\, s, r \bullet P^t_f[s, r/tr', ref'] \wedge L_t)\end{array}\right)$$

Proof.

$(P \setminus E)^t_f$ [def of $\setminus$]

$= (\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t) \,;\, Skip)^t_f$ [relational calculus]

$= (\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t))_f \,;\, Skip^t$ [Law 8]

$= (\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t))_f \,;\, (\mathbf{R1_t}(\neg\, ok) \vee (ok \wedge \mathit{II}))$ [relational cal.]

$= (\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t))_f \,;\, \mathbf{R1_t}(\neg\, ok) \vee$ [Step 4 in Law 10]
$(\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t))_f \,;\, (ok \wedge \mathit{II})$

$= (\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true) \vee$ [P is $\mathbf{R1_t}$ and *wait* is *false*]
$(\mathbf{R_t}(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t))_f \,;\, (ok \wedge \mathit{II})$

$= (\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true) \vee$ [relational calculus]
$(\exists\, s, r \bullet P[s, r/tr', ref'] \wedge L_t)_f \,;\, (ok \wedge \mathit{II})$

$= (\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true) \vee$ [unit law]
$(\exists\, s, r \bullet P^t_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathit{II}$

$= (\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true) \vee (\exists\, s, r \bullet P^t_f[s, r/tr', ref'] \wedge L_t)$

Now, the reactive design of hiding can be deduced in terms of the above laws.

Theorem 3. *Suppose P is a Circus Time action,*

$$P \setminus CS = \mathbf{R_t}\left(\begin{array}{c}\neg\, ((\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true)) \\ \vdash\, (\exists\, s, r \bullet P^t_f[s, r/tr', ref'] \wedge L_t)\end{array}\right)$$

Proof.

$P \setminus CS$ [Theorem 1]

$= \mathbf{R_t}(\neg\, (P \setminus CS)^f_f \vdash (P \setminus CS)^t_f)$ [def of design]

$= \mathbf{R_t}(ok \wedge \neg\, (P \setminus CS)^f_f \Rightarrow ok' \wedge (P \setminus CS)^t_f)$ [Law 10 and Law 11]

$$= \mathbf{R_t}\left(\begin{array}{l}ok \wedge \neg\, ((\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true)) \Rightarrow \\ ok' \wedge \left(\begin{array}{c}(\exists\, s, r \bullet P^f_f[s, r/tr', ref'] \wedge L_t) \,;\, \mathbf{R1_t}(true) \\ \vee\, (\exists\, s, r \bullet P^t_f[s, r/tr', ref'] \wedge L_t)\end{array}\right)\end{array}\right)$$ [prop. cal.]

$$= \mathbf{R_t}\begin{pmatrix} ok \wedge \neg\,((\exists\, s, r \bullet P_f^f[s, r/tr', ref'] \wedge L_t)\,;\, \mathbf{R1_t}(true)) \Rightarrow \\ ok' \wedge (\exists\, s, r \bullet P_f^t[s, r/tr', ref'] \wedge L_t) \end{pmatrix} \quad \text{[def-design]}$$

$$= \mathbf{R_t}\begin{pmatrix} \neg\,((\exists\, s, r \bullet P_f^f[s, r/tr', ref'] \wedge L_t)\,;\, \mathbf{R1_t}(true)) \\ \vdash (\exists\, s, r \bullet P_f^t[s, r/tr', ref'] \wedge L_t) \end{pmatrix}$$

Note that $\mathbf{R1_t}(true)$ in the precondition captures the observation that leads to a divergence.

3.3 Recursion

The semantics of recursion is the same as that in the UTP [6]: weakest fixed point. Given a monotonic function F, the semantics of recursion is the weakest fixed point of F.

$$\mu\, X \bullet F(X) \mathrel{\widehat{=}} \bigsqcap\{X \mid F(X) \sqsubseteq X\} \tag{3}$$

The strongest fixed point of $F(X)$ is defined as the dual of the weakest.

$$\nu F \mathrel{\widehat{=}} \neg\ \mu\, X \bullet \neg\, F(\neg\, X) \tag{4}$$

To express a recursion as a reactive design, we have to calculate the precondition and postcondition of a recursively defined design. For that, we can use the definition of a recursive design and some theorems on linking theories in [6]. In the theory of designs, any monotonic function of designs can be expressed in terms of a pair of function that apply separately to the precondition and the postcondition, for example

$$F(P, Q) \vdash G(P, Q)$$

Here, P and Q are predicates representing the precondition and postcondition of a design, F is monotonic in P and antimonotonic in Q, whereas G is monotonic in Q and antimonotonic in P. Thus, as described in the theory of designs, the weakest fixed point is given by a mutually recursive formula, that we reproduce below.

Law 12

$$\begin{aligned} & \mu(X, Y) \bullet (F(X, Y) \vdash G(X, Y)) = P(Q) \vdash Q \\ \text{where}\quad & P(Y) = \nu X \bullet F(X, Y) \\ \text{and}\quad & Q = \mu\, Y \bullet (P(Y) \Rightarrow G(P(Y), Y)) \end{aligned}$$

As shown in Theorem 1, if X is a reactive design, $X = \neg\, X_f^f \vdash X_f^t$. Hence, based on Law 12, we have the following theorem for recursively reactive designs.

Theorem 4.

$$\mu(X, Y) \bullet (\mathbf{R_t}(F(X, Y) \vdash G(X, Y))) = \mathbf{R_t}(\mu(X, Y) \bullet (F(X, Y) \vdash G(X, Y))$$

To prove Theorem 4, we directly adopt an important theorem from the linking theories of the UTP book, which can be described here.

Theorem 5. *Let D and E be monotonic functions. If there exists a function R such that $R \circ D = E \circ R$, then $R(\mu\, D) = \mu\, E$.*

As a result, the proof of Theorem 4 can be established as follows.

Proof.

$$\begin{aligned} \text{Let}\quad & D(X \vdash Y) = F(X, Y) \vdash G(X, Y) \quad \textit{and} \\ & E(\mathbf{R_t}(X \vdash Y)) = \mathbf{R_t}(F(X, Y) \vdash G(X, Y)) \end{aligned}$$

$$\begin{aligned} \text{then}\quad & E \circ \mathbf{R_t}(X \vdash Y) && [\text{def of } E] \\ =& \mathbf{R_t}(F(X, Y) \vdash G(X, Y)) && [\text{def of } D] \\ =& \mathbf{R_t}(D(X \vdash Y) && [\text{def of composition}] \\ =& \mathbf{R_t} \circ D(X \vdash Y) \end{aligned}$$

$$\text{therefore}\quad \mu(X, Y) \bullet E(\mathbf{R_t}(X \vdash Y)) = \mathbf{R_t}(\mu(X, Y) \bullet D(X \vdash Y))$$

4 Applications of Reactive Designs

The reactive design semantics can help us understand the exact behaviours of some complex processes. For example, the reactive design of a simple prefix, which is based on the semantics in [2], has been worked out in [13].

$$c.e \rightarrow \mathit{Skip} \mathrel{\widehat{=}} \mathbf{R_t}(\mathit{true} \vdash \mathit{wait\_com}(c) \lor \mathit{terminating\_com}(c.e)) \tag{5}$$

$$\mathit{wait\_com}(c) \mathrel{\widehat{=}} \mathit{wait}' \land \mathit{possible}(\mathit{ref}, \mathit{ref}', c) \land {}^\frown\!/tr' = {}^\frown\!/tr \tag{6}$$

$$\mathit{possible}(\mathit{ref}, \mathit{ref}', c) \mathrel{\widehat{=}} \forall\, i : \#\mathit{ref} .. \#\mathit{ref}' \bullet c \notin \mathit{ref}'(i) \tag{7}$$

$$\mathit{term\_com}(c.e) \mathrel{\widehat{=}} \begin{pmatrix} \neg\ \mathit{wait}' \land \mathit{diff}(tr', tr) = \langle\langle c.e \rangle\rangle \\ \land\ \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref}) \end{pmatrix} \tag{8}$$

$$\mathit{terminating\_com}(c.e) \mathrel{\widehat{=}} \begin{pmatrix} \mathit{wait\_com}(c)\,;\, \mathit{term\_com}(c.e) \\ \lor\ \mathit{term\_com}(c.e) \end{pmatrix} \tag{9}$$

Such a process never diverges since its precondition is true, and, as described by its postcondition, behaves in three different ways: it waits for interaction from its environment, or it waits for a while and then terminates with a fired event, or it simply executes the event immediately. The action *Miracle*, expressed as $\mathbf{R_t}(\mathit{true} \vdash \mathit{false})$, has miraculous behaviour that simply denotes an unstarted action. Therefore, it should never appear during an execution of a process. The exact behaviour of the combination of the two actions can be easily figured out using our newly established reactive design of sequential composition.

Theorem 6.

$$c.e \rightarrow \mathit{Miracle} = \mathbf{R_t}(\mathit{true} \vdash \mathit{wait}' \wedge \frown/tr' = \frown/tr \wedge \mathit{possible}(tr, tr', c))$$

Proof.

$$
\begin{array}{ll}
& c.e \rightarrow \mathit{Miracle} \qquad [\text{def of prefix}] \\
= & (c.e \rightarrow \mathit{Skip})\,;\,\mathit{Miracle} \qquad [\text{def 5 and } \mathit{Miracle}] \\
= & \mathbf{R_t}(\mathit{true} \vdash \mathit{wait\_com}(c) \vee \mathit{terminating\_com}(c.e))\,;\,\mathbf{R_t}(\mathit{true} \vdash \mathit{false}) \\
& \qquad [\text{Theorem 2}] \\
= & \mathbf{R_t}\left(\begin{array}{c} \neg\,(\mathbf{R1_t}(\mathit{true})\,;\,\mathbf{R1_t}(\mathit{true})) \wedge \\ \neg\,(\mathbf{R1_t}((\mathit{wait\_com}(c) \vee \mathit{terminating\_com}(c.e))^t_f)\,;\,\mathbf{R1_t}(\neg\,\mathit{wait} \wedge \mathit{false})) \\ \vdash \\ \mathbf{R1_t}(\mathit{wait\_com}(c) \vee \mathit{terminating\_com}(c.e))\,;\,\mathbf{R1_t}(\mathit{I\!I} \lhd \mathit{wait} \rhd \mathit{false}) \end{array}\right) \\
& \qquad [\text{rel. cal.}] \\
= & \mathbf{R_t}(\mathit{true} \vdash \mathbf{R1_t}(\mathit{wait\_com}(c) \vee \mathit{terminating\_com}(c.e))\,;\,\mathbf{R1_t}(\mathit{I\!I} \wedge \mathit{wait})) \\
& \qquad [\text{rel. cal.}] \\
= & \mathbf{R_t}\left(\mathit{true} \vdash \left(\begin{array}{c} \mathbf{R1_t}(\mathit{wait\_com}(c))\,;\,\mathbf{R1_t}(\mathit{I\!I} \wedge \mathit{wait}) \vee \\ \mathbf{R1_t}(\mathit{terminating\_com}(c.e))\,;\,\mathbf{R1_t}(\mathit{I\!I} \wedge \mathit{wait}) \end{array}\right)\right) \qquad [\text{def 9}] \\
= & \mathbf{R_t}\left(\mathit{true} \vdash \left(\begin{array}{c} \mathbf{R1_t}(\mathit{wait\_com}(c))\,;\,\mathbf{R1_t}(\mathit{I\!I} \wedge \mathit{wait}) \vee \\ \left(\begin{array}{c} \mathbf{R1_t}(\mathit{wait\_com}(c)\,;\,\mathit{term\_com}(c.e) \\ \vee\, \mathit{term\_com}(c.e) \end{array}\right)\,;\,\mathbf{R1_t}(\mathit{I\!I} \wedge \mathit{wait}) \end{array}\right)\right) \\
& \qquad [\mathit{wait}' \text{ in } \mathit{term\_com} \text{ is false}] \\
= & \mathbf{R_t}(\mathit{true} \vdash (\mathbf{R1_t}(\mathit{wait\_com}(c))\,;\,\mathbf{R1_t}(\mathit{I\!I} \wedge \mathit{wait}) \vee \mathit{false})) \qquad [\text{def 6}] \\
= & \mathbf{R_t}(\mathit{true} \vdash \mathbf{R1_t}(\mathit{wait}' \wedge \frown/tr' = \frown/tr \wedge \mathit{possible}(tr, tr', c))\,;\,\mathbf{R1_t}(\mathit{I\!I} \wedge \mathit{wait}) \\
& \qquad [\text{rel. cal.}] \\
= & \mathbf{R_t}(\mathit{true} \vdash \mathbf{R1_t}(\mathit{wait}' \wedge \frown/tr' = \frown/tr \wedge \mathit{possible}(tr, tr', c)) \\
& \qquad [\mathbf{R1_t} \text{ is idempotent}] \\
= & \mathbf{R_t}(\mathit{true} \vdash \mathit{wait}' \wedge \frown/tr' = \frown/tr \wedge \mathit{possible}(tr, tr', c))
\end{array}
$$

This theorem states that, if this action starts, it waits for interaction with its environment, but never actually perform any event even if the event $c.e$ has been offered. This process is different from that of the standard CSP failures-divergences model in which one of the assumptions requires that, if an event is not in the refusal set, the process is always willing to execute the event.

There is a very subtle law in the CSP theory about hiding and recursion as $(\mu\, X \bullet c \rightarrow X) \setminus \{c\} = \mathit{Chaos}$, which is difficult to be proved using their original UTP definitions. However, the reactive designs of the two operators allow us to prove this law straightforwardly. To prove this law, we use the *Kleene* theorem rather than the traditional definition of the weakest fixed point to calculate the recursive design in the recursively reactive design.

Theorem 7 (Kleene fixed point theorem).
If F is continuous [1]*, then $\mu\, X \bullet F(X) = \bigsqcup_{n=0}^{\infty} F^n(true)$ where $F^0(X) \mathrel{\widehat{=}} true$, and $F^{n+1} \mathrel{\widehat{=}} F(F^n(X))$.*

This theorem states a normal form for programs that contain recursion. First of all, the behaviour of a recursive program is expressed as an *infinite* sequence of predicates $\{F^i \mid i \in \mathbb{N}\}$ and each F^i is a finite normal form. Since each F^{i+1} is defined by its previous expression, F^{i+1} is potentially stronger if $F^i \sqsubseteq F^{i+1}$. If i is large enough, the exact behaviour of the program can be captured by the *least upper bound* of the infinite sequence, written $\bigsqcup_{n=0}^{\infty} F^n(true)$.

In addition, we are able to prove a similar theorem for the strongest fixed point of F.

Theorem 8. *If F is continuous, $\nu X \bullet F(X) = \bigsqcap_{n=0}^{\infty} F^n(false)$*

Proof.

$$
\begin{aligned}
& \nu X \bullet F(X) && \text{[def of } \nu\text{]} \\
=& \neg\ \mu\, X \bullet \neg\, F(\neg\, X) && \text{[Theorem 7]} \\
=& \neg \bigsqcup_{n=0}^{\infty} (\lambda\, X \bullet \neg\, F(\neg\, X))^n(true) && \text{[relational calculus]} \\
=& \bigsqcap_{n=0}^{\infty} \neg\, (\lambda\, X \bullet \neg\, F(\neg\, X))^n(true) && \text{[predicate calculus]} \\
=& \bigsqcap_{n=0}^{\infty} F^n(false)
\end{aligned}
$$

Now, firstly, we calculate the reactive design of a single call of $\mu\, X \bullet c \to X$. The procedure is similar to Theorem 6, and the proof can be found in [13].

Law 13

$$
c \to X = \mathbf{R_t} \begin{pmatrix} \neg\, (terminating\_com(c)\,;\, \mathbf{R1_t}(\neg\, wait \wedge X_f^f)) \\ \vdash \\ wait\_com(c) \vee (terminating\_com(c)\,;\, \mathbf{R1_t}(X_f^t)) \end{pmatrix}
$$

Secondly, in terms of Theorem 4 and Law 13, we let

$$X = \neg\, X_f^f \quad \text{and} \quad Y = X_f^t, \quad \text{then} \tag{10}$$

$$F(X, Y) = \neg\, (terminating\_com(c)\,;\, \mathbf{R1_t}(\neg\, wait \wedge \neg\, X)) \tag{11}$$

$$G(X, Y) = wait\_com(c) \vee (terminating\_com(c)\,;\, \mathbf{R1_t}(Y)) \tag{12}$$

As a result, the weakest fixed point of $(\mu\, X \bullet c \to X) \setminus \{c\}$ can be calculated by the following law.

[1] A function is *continuous* only if its value at a limit point can be determined from its values on a sequence converging to that point. Also, a continuous function is monotonic.

Law 14

$$\mu(X, Y) \bullet E(\mathbf{R_t}(X \vdash Y)) = \mathbf{R_t}((\nu X \bullet F(X, Y)) \vdash Q)$$

Proof.

$$\begin{aligned}
&\mu(X, Y) \bullet E(\mathbf{R_t}(X \vdash Y)) && \text{[Theorem 4]}\\
=&\mathbf{R_t}(\mu(X, Y) \bullet D(X \vdash Y)) && \text{[Law 12]}\\
=&\mathbf{R_t}((\nu X \bullet F(X, Y)) \vdash Q)
\end{aligned}$$

Note that, since the postcondition of D has no influence on the final result, we here simply use Q to denote the postcondition and never unfold it in the later proof.

Next, we calculate the strongest fixed point of F by means of the *Kleene* theorem. Before starting to prove the law before, we give some useful properties. Some proofs can be found in [13], and some leave to the reader.

Property 1.

- *L*1. $\mathbf{R1_t}(terminating\_com(c)) = terminating\_com(c)$
- *L*2. $terminating\_com(c);\ \mathbf{R1_t}(true) \wedge terminating\_com(c)^2;\ \mathbf{R1_t}(true)$
  $= terminating\_com(c)^2;\ \mathbf{R1_t}(true)$
- *L*3. $terminating\_com(c)^n \sqsubseteq term\_com(c)^n$
- *L*4. $diff(tr', tr) = \langle\langle c\rangle\rangle \Leftrightarrow front(tr') = front(tr) \wedge last(tr') - last(tr) = \langle c\rangle$
- *L*5. $((front(ref') = front(ref));\ (front(ref') = front(ref)))$
  $\Leftrightarrow front(ref') = front(ref)$

Law 15

$$\nu(X) \bullet F(X, Y) = \neg \left(\mathbf{R1_t} \begin{pmatrix} front(tr) \frown \langle last(tr) \frown \langle c\rangle^n\rangle \preccurlyeq tr' \\ \wedge\ front(ref) \leq ref' \\ \vee\ (terminating\_com(c)^n ; \mathbf{R1_t}(true)) \end{pmatrix} \right)$$

Proof.

$$\begin{aligned}
&\nu(X) \bullet F(X, Y) && \text{[Theorem 8]}\\
=&\prod_{n=0}^{\infty} F^n(false) && \text{[unfold } \textstyle\prod\text{]}\\
=&\ F^0(false) \sqcap F^1(false) \sqcap F^2(false) ... \sqcap F^n(false) && \text{[unfold } F\text{(def 11)]}\\
=&\ false \sqcap \neg\,(terminating_com(c)\,;\,\mathbf{R1_t}(\neg\ wait \wedge \neg\ false)) ...\\
&\sqcap \neg \left(\begin{matrix} terminating_com(c) \\ ; \\ \mathbf{R1_t}(\neg\ wait \wedge (terminating_com(c)\,;\,\mathbf{R1_t}(\neg\ wait \wedge \neg\ false))) \end{matrix} \right) ...\\
&\sqcap F^n(false) && \text{[relational calculus } (\neg\ wait \text{ is absorbed by } terminating_com)\text{]}
\end{aligned}$$

$$
\begin{aligned}
&= \mathit{false} \sqcap \neg\,(\mathit{terminating\_com}(c)\,;\mathbf{R1_t}(\mathit{true})) \\
&\quad \sqcap \neg\,(\mathit{terminating\_com}(c)\,;\mathbf{R1_t}((\mathit{terminating\_com}(c)\,;\mathbf{R1_t}(\mathit{true})))... \\
&\quad \sqcap F^n(\mathit{false}) \qquad \text{[Property 1-L1 and } \mathbf{R1_t} \text{ is idempotent and rel. cal.]} \\
&= \mathit{false} \sqcap \neg\,(\mathit{terminating\_com}(c)\,;\mathbf{R1_t}(\mathit{true})) \qquad \text{[property of } \textstyle\bigsqcap\text{]} \\
&\quad \sqcap \neg\,(\mathit{terminating\_com}(c)^2\,;\mathbf{R1_t}(\mathit{true}))... \\
&\quad \sqcap \neg\,(\mathit{terminating\_com}(c)^n\,;\mathbf{R1_t}(\mathit{true})) \\
&= \prod_{n=1}^{\infty}(\neg\,(\mathit{terminating\_com}(c)^n\,;\mathbf{R1_t}(\mathit{true}))) \qquad \text{[property of } \textstyle\bigsqcup\text{]} \\
&= \neg \bigsqcup_{n=1}^{\infty}(\mathit{terminating\_com}(c)^n\,;\mathbf{R1_t}(\mathit{true})) \qquad \text{[unfold } \textstyle\bigsqcup\text{]} \\
&= \neg \begin{pmatrix} (\mathit{terminating\_com}(c)\,;\mathbf{R1_t}(\mathit{true})) \wedge \\ (\mathit{terminating\_com}(c)^2\,;\mathbf{R1_t}(\mathit{true})) \wedge \\ \mathbf{....} \wedge \\ (\mathit{terminating\_com}(c)^n\,;\mathbf{R1_t}(\mathit{true})) \end{pmatrix} \qquad \text{[Property 1-L2 and Induction]} \\
&= \neg\,(\mathit{terminating\_com}(c)^n\,;\mathbf{R1_t}(\mathit{true})) \qquad \text{[def 9]} \\
&= \neg\,((\mathit{wait\_com}(c)\,;\mathit{term\_com}(c) \vee \mathit{term\_com}(c))^n\,;\mathbf{R1_t}(\mathit{true})) \\
&\qquad \text{[Property 1-L3]} \\
&= \neg \left(\begin{pmatrix} (\mathit{wait\_com}(c)\,;\mathit{term\_com}(c) \vee \mathit{term\_com}(c))^n \\ \vee\ \mathit{term\_com}(c)^n \end{pmatrix} ;\mathbf{R1_t}(\mathit{true}) \right) \qquad \text{[def 9]} \\
&= \neg\,((\mathit{term\_com}(c)^n \vee \mathit{terminating\_com}(c)^n)\,;\mathbf{R1_t}(\mathit{true})) \qquad \text{[def 8]} \\
&= \neg \left(\left(\begin{pmatrix} \neg\ \mathit{wait}' \wedge \mathit{diff}(tr', tr) = \langle\langle c\rangle\rangle \\ \wedge\ \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref}) \end{pmatrix}^n \right) ;\mathbf{R1_t}(\mathit{true}) \right) \qquad \text{[Property 1-L4]} \\
&= \neg \left(\left(\begin{pmatrix} \neg\ \mathit{wait}' \wedge \mathit{last}(tr') - \mathit{last}(tr) = \langle c\rangle \\ \wedge\ \mathit{front}(tr') = \mathit{front}(tr) \wedge \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref}) \end{pmatrix}^n \right) ;\mathbf{R1_t}(\mathit{true}) \right) \\
&\qquad \text{[Property 1-L5 and Induction]} \\
&= \neg \left(\left(\begin{pmatrix} \neg\ \mathit{wait}' \wedge (\mathit{last}(tr') - \mathit{last}(tr) = \langle c\rangle)^n \wedge \\ \mathit{front}(tr') = \mathit{front}(tr) \wedge \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref}) \end{pmatrix} \right) ;\mathbf{R1_t}(\mathit{true}) \right) \\
&\qquad \text{[rel. cal.]} \\
&= \neg \left(\begin{pmatrix} \neg\ \mathit{wait}' \wedge \mathit{diff}(tr' - tr) = \langle\langle c\rangle^n\rangle \\ \wedge\ \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref}) \end{pmatrix} ;\mathbf{R1_t}(\mathit{true}) \right) \\
&\qquad \vee\ \mathit{terminating\_com}(c)^n\,;\mathbf{R1_t}(\mathit{true}) \\
&\qquad \text{[def of 2 and relational calculus]} \\
&= \neg \left(\begin{pmatrix} \neg\ \mathit{wait}' \wedge \\ \langle tr'(\#tr) - \mathit{last}(tr)\rangle \frown \mathit{tail}(tr' - \mathit{front}(tr)) = \langle\langle c\rangle^n\rangle \\ \wedge\ \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref}) \end{pmatrix} ;\mathbf{R1_t}(\mathit{true}) \right) \\
&\qquad \vee\ \mathit{terminating\_com}(c)^n\,;\mathbf{R1_t}(\mathit{true}) \\
&\qquad \text{[relational calculus]}
\end{aligned}
$$

$$= \neg \begin{pmatrix} \mathbf{R1_t}(\mathit{front}(tr) \frown \langle \mathit{last}(tr) \frown \langle c \rangle^n \rangle \preccurlyeq tr' \wedge \mathit{front}(ref) \leq ref') \\ \vee\ \mathit{terminating\_com}(c)^n \,;\, \mathbf{R1_t}(\mathit{true}) \end{pmatrix}$$

Finally, we are ready to prove the law, $(\mu\, X \bullet c \to X) \setminus \{c\} = \mathit{Chaos}$, and the proof is simply the combination of the laws and theorems above.

Theorem 9.

$$(\mu\, X \bullet c \to X) \setminus \{c\} = \mathit{Chaos}$$

Proof.

$$(\mu\, X \bullet c \to X) \setminus \{c\} \qquad \text{[Theorem 3]}$$

$$= \mathbf{R_t}(\neg\, ((\exists\, s, r \bullet (\mu\, X \bullet c \to X)^f_f[s, r/tr', ref'] \wedge L_t);\ \mathbf{R1_t}(\mathit{true})) \vdash EE^2)$$
[def of $\vdash$ and merge unused proof]

$$= \mathbf{R_t}(((\exists\, s, r \bullet (\mu\, X \bullet c \to X)^f_f[s, r/tr', ref'] \wedge L_t);\ \mathbf{R1_t}(\mathit{true})) \vee EE)$$
[Theorem 4]

$$= \mathbf{R_t}\begin{pmatrix} ((\exists\, s, r \bullet (\mu(X, Y) \bullet E(\mathbf{R_t}(X \vdash Y)))^f_f[s, r/tr', ref'] \wedge L_t); \mathbf{R1_t}(\mathit{true})) \\ \vee\ EE \end{pmatrix}$$
[Law 14]

$$= \mathbf{R_t}\begin{pmatrix} ((\exists\, s, r \bullet (\neg\, \nu X \bullet F(X, Y))[s, r/tr', ref'] \wedge L_t); \mathbf{R1_t}(\mathit{true})) \\ \vee\ EE \end{pmatrix} \qquad \text{[Law 15]}$$

$$= \mathbf{R_t}\left(\begin{pmatrix} \exists\, s, r \bullet \begin{pmatrix} \mathbf{R1_t}\begin{pmatrix} \mathit{front}(tr) \frown \langle \mathit{last}(tr) \frown \langle c \rangle^n \rangle \preccurlyeq tr' \\ \wedge\ \mathit{front}(ref) \leq ref' \end{pmatrix} \\ \vee\ \mathit{terminating\_com}(c)^n \,;\, \mathbf{R1_t}(\mathit{true}) \end{pmatrix}[s, r/tr', ref'] \wedge L_t \\ ;\ \mathbf{R1_t}(\mathit{true}) \\ \vee\ EE \end{pmatrix}\right)$$
[merge $(\mathit{terminating\_com}(c)^n \,;\, \mathbf{R1_t}(\mathit{true}))$ to EE]

$$= \mathbf{R_t}\left(\begin{pmatrix} \exists\, s, r \bullet \mathbf{R1_t}\begin{pmatrix} \mathit{front}(tr) \frown \langle \mathit{last}(tr) \frown \langle c \rangle^n \rangle \preccurlyeq tr' \\ \wedge\ \mathit{front}(ref) \leq ref' \end{pmatrix}[s, r/tr', ref'] \wedge L_t \\ ;\ \mathbf{R1_t}(\mathit{true}) \\ \vee\ EE \end{pmatrix}\right)$$
[only $tr' = tr \wedge \mathit{front}(ref) \leq ref'$ can satisfy L_t]

$$= \mathbf{R_t}((\mathbf{R1_t}(tr' = tr \wedge \mathit{front}(ref) \leq ref') \,;\, \mathbf{R1_t}(\mathit{true})) \vee EE) \qquad \text{[rel. calculus]}$$

$$= \mathbf{R_t}(\mathbf{R1_t}(\mathit{true}) \vee EE) \qquad \text{[prop. calculus]}$$

$$= \mathbf{R_t}(\mathit{true}) \qquad \text{[def of design]}$$

$$= \mathbf{R_t}(\mathit{false} \vdash \mathit{true}) \qquad \text{[def of } \mathit{Chaos}\text{]}$$

$$= \mathit{Chaos}$$

[2] We simply use EE to denote the unused part of the proof. This abbreviation continuously collects unused parts during the proof and it is changing at each step of this proof.

The proof of the above law shows one of the cases to result in *Chaos*. If c happens immediately at each call, the hiding operator is able to make this recursion become divergent at once when it starts.

5 Conclusion

In this paper we develop the reactive design semantics of three important CSP operators, sequential composition, hiding and recursion; this complements the early work in [2, 9]. Compared to the original CSP semantics in UTP, the reactive designs provides us with a more concise, readable and uniform semantics, which can help us to exactly understand the behaviours of some subtle processes. In addition, this reactive design semantics is developed in a timed context, *Circus Time*, and the full version can be found in [13]. So far, this semantics and related laws have been proved by hand. In our short-term goal, we will mechanise them in a new *Circus* tool, Isabelle/Circus [4], to underpin their correctness.

Acknowledgments. This work was fully supported by hiJaC project funded by EPSRC(EP/H017461/1).

References

1. A. Cavalcanti, A. Wellings, J. Woodcock, K. Wei, and F. Zeyda. Safety-critical Java in Circus. In *Proceedings of the 9th International Workshop on Java Technologies for Real-Time and Embedded Systems*, JTRES '11, pages 20–29, New York, NY, USA, 2011. ACM.
2. A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.
3. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
4. A. Feliachi, M.-C. Gaudel, and B. Wolff. Isabelle/Circus : a Process Specification and Verification Environment. Technical Report 1547, LRI, http://www.lri.fr/Rapports-internes, Université Paris-Sud XI, November 2011.
5. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
6. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
7. D. Locke and et al. Safety Critical Java Specification. *First Release 0.76, The Open Group, UK*, 2010.
8. C. Morgan. *Programming from specifications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1990.
9. M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1):3 – 32, 2007.
10. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1998.
11. S. A. Schneider. *Concurrent and real-time systems: the CSP approach*. John Wiley & Sons, 1999.

12. 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.
13. K. Wei, J. Woodcock, and A. Cavalcanti. New *Circus Time*. Technical report, Department of Computer Science, University of York, UK, March 2012. Avaliable at http://www.cs.york.ac.uk/circus/hijac/publication.html.
14. J. Woodcock. The miracle of reactive programming. In *Unifying Theories of Programming 2008: 2nd International Symposium*, Dublin, Ireland, 2008. Springer-Verlag.
15. J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.
16. J. C. P. Woodcock and A. L. C. Cavalcanti. A Tutorial Introduction to Designs in Unifying Theories of Programming. In *IFM 2004*, volume 2999 of *LNCS*, pages 40 – 66.
17. 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.
18. J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of *circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors, *ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of *Lecture Notes in Computer Science*, pages 184—203. Springer-Verlag, 2002.