Reactive Designs of Interrupts in *Circus Time*

Kun Wei

Department of Computer Science, University of York, York, YO10 5GH, UK
kun.wei@york.ac.uk

Abstract. The concept of interrupts is important in system specifications across both software and hardware. However, behaviours of interrupts are difficult to capture particularly in a timed environment because of its complexity. In this paper, the catastrophic interrupt adopted by the standard CSP models, the generic interrupt presented by Hoare in his original CSP book and the timed interrupt (time-driven) given in Timed CSP are considered in *Circus Time*. The contribution of the paper is a development of the reactive design semantics of these three interrupt operators in the UTP, a collection of verified laws, and a comprehensive discussion on the subtle difference between catastrophic and generic interrupts in applications.

Keywords: Interrupt, UTP, reactive designs.

1 Introduction

Interrupt behaviours are important in system modelling of both hardware and software. For instance, pressing the reset button can restart a system to its original state, or a piece of hardware may have a special input line to output a value even if it has not been ready. In developing software, interrupts can naturally describe a variety of behaviours such as exception handlers in object-oriented programs which may stop the current task to indicate an error situation, and a scheduling algorithm that can always execute a task with a highest priority by suspending the current tasks. However, formally specifying the behaviour of interrupts [7,24] is notoriously difficult particularly under a timed environment because of its complexity.

Over three decades, CSP presented by Hoare in [5] has become a successful formal language for specifying and reasoning about concurrency and communication in a system, with important technical results as those presented in [13,14], and many powerful tools, e.g., FDR [1] and ProB [9]. The interrupt operators and its applications in CSP have been discussed by Hoare in his original work [5], in $P \triangle Q$ [1] the execution of P is interrupted on occurrence of the first (external)

[1] Hoare's original work [5] uses $P^\frown Q$ to express the generic interrupt operator, and $P \widehat{\lightning} Q$ to present the catastrophic interrupt operator where $\lightning$ is a unique event. In later work, however, Hoare uses different symbols to express the interrupt operators. Here, we adopt $\triangle$ from [14] and $\triangle_c$ from [13] to present the generic and catastrophic interrupt operators, respectively.

Z. Liu, J. Woodcock, and H. Zhu (Eds.): ICTAC 2013, LNCS 8049, pp. 373–390, 2013.

event of Q. Here, Q is called the *interrupting process* and P is the *interruptible process*. In the standard (untimed) models of CSP [13], the interruptible process is always interrupted by a catastrophic interrupt event, and thereafter the process behaves like Q. This kind of interruptions is called a catastrophe, $P \triangle_c Q$, in which c is a unique event that does not appear in P. This simpler interrupt can avoid the complication that arises from allowing Q to be an arbitrary process. Alternatively, Timed CSP [14] uses the original interrupt operator in Hoare's work [5], namely the generic interrupt operator, since the catastrophic interrupt operator is insufficient for specifying certain timing behaviours in a timed system. For example, the behaviour of $P \triangle (Wait\ d; c \rightarrow Skip)$ cannot be captured by using a catastrophic interrupt operator. Moreover, Timed CSP [14] provides a timed interrupt operator, $P \triangle_d Q$, to allow P to execute for d time units at most before Q takes the control. The characteristic of this timed interrupt operator, compared with the generic interrupt operator, is that the occurrence of an interruption is time-driven; that is, the interruption is out of control of the environment, but only depends on the time.

Circus [3,21,22] is a comprehensive combination of Z [20], CSP [5,13] and Morgan's refinement calculus [11], so that it can define both data and behavioural aspects of a system. *Circus Time* [15,19] 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 [6] by introducing timed observation variables. The *Circus Time* model is a discrete time model, and time operators are very similar to that in Timed CSP. In the *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. The importance of this semantics is that it exposes the pre-postcondition semantics. In this paper, we develop the reactive design semantics of the catastrophic interrupt, the generic interrupt and the timed interrupt operators in *Circus Time*.

The work in [10] firstly explores the UTP semantics of the catastrophic interrupt operators and related laws in (untimed) CSP. This approach considers an interruption as a kind of sequential composition, because that the interrupting process, in fact, has not happened until the occurrence of the catastrophic event. Therefore, it imitates the idea of the interrupt step law to make the interrupt event able to happen at any intermediate waiting state of P. The UTP semantics of the catastrophic interrupt operator in a timed environment can be found in a hybrid CSP system [8]. Unfortunately, these approaches that consider interruptions as sequential composition cannot contribute to the UTP semantics of the generic interrupt operator, since interrupt events might be a part of an interrupting process as both P and Q evolve together. From this point of view, we ponder the generic interrupt operator as a parallel composition.

The main contribution of this paper is to define the reactive designs of the catastrophic interrupt operator that follows the same idea in the work [10], of the generic interrupt operator that is inspired by Timed CSP [14] to treat an interruption as a parallel composition, and of the timed interrupt operator that is intuitively defined by considering its precondition and postcondition respectively.

Remarkably, the reactive design semantics indicates that the catastrophic interrupt operator is not equal to the generic interrupt operator with an explicitly interrupting event, because their reactive designs use different approaches so that the generic interrupt operator is able to capture more behaviours. In addition, these reactive designs are underpinned by showing that they still enjoy a number of existing algebraic laws.

This paper is structured as follows. Section 2 presents an overview of related UTP theories and *Circus Time*. The detailed reactive designs of the catastrophic, generic and timed interrupt operators are given in Section 3 with a number of relevant laws. Finally, we conclude and discuss future work in Section 4.

2 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 αP, 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.

A theory in the UTP is a collection of relations (or alphabetised predicates), which contains three essential parts: an alphabet, a signature, and healthiness conditions: the alphabet is a set of variable names for observation, the signature gives a set of operators and atomic components of the programming theory, and the healthiness conditions identify properties that characterise the theory.

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 ($P \sqsubseteq \mathbf{false}$).

2.1 Designs

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. There are four healthiness conditions identified by Hoare and He [6] in the theory of designs and here we introduce only two of them.

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

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 predicate is **H1** and **H2** if, and only if, it is a design; the proof is in [6]. The theory of designs (**H1** and **H2**-healthy relations) has an important role in models for process algebras for refinement like CSP and *Circus*. For a tutorial introduction to designs, the reader is referred to [6,23].

2.2 Reactive Processes

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. We are able to represent some 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 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, II_{rea}, is defined as $\mathit{II}_{rea} \mathrel{\widehat{=}} (\neg\ ok \wedge tr \leq tr') \vee (ok' \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref)$ which states that if its predecessor diverges ($\neg\ ok$), the extension of traces is the only guaranteed observation; otherwise (ok'), other variables keep unchanged. A reactive process must satisfy the following healthiness conditions:

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

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 (II_{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 $\mathbf{R}$. 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:

$$\mathbf{CSP1} \quad P = P \vee (\neg\, ok \wedge tr \leq tr') \qquad \mathbf{CSP2} \quad P = P\,;\, J$$

where $J = (ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref$. 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. The CSP theory introduced in the UTP book is different from any standard models of CSP [5,13] which have more restrictions or satisfy more healthiness conditions.

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, $P = \mathbf{R}(\neg\, P_f^f \vdash P_f^t)$, where P_b^a 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 $\mathbf{R}$. For example, P_f^t describes the behaviour when P is stable, and P_f^f captures the behaviour before a divergent state. The importance of this reactive design semantics is that it exposes the pre-postcondition semantics so as to not only support contract-based reasoning about models, but also simplify proof of *Circus Time* laws.

2.4 *Circus Time*

We give a brief introduction to *Circus Time* because the reactive design semantics of interrupts is developed within this timed model. In *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. *Circus Time* presents traces and refusals individually rather than using the concept of failures. However, for their consistency we have to ensure the equality of the lengths of tr and ref, and tr' and ref'. This is achieved by imposing an extra constraint on the healthiness conditions.

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

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

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 [19].

$$\mathbf{R1_{ct}}(X) \mathrel{\widehat=} X \wedge RT \qquad \mathbf{R3_{ct}}(X) \mathrel{\widehat=} \mathit{II}_{ct} \lhd wait \rhd X$$

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

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

Note that we impose a restriction, $\#tr = \#ref \wedge \#tr' = \#ref'$, 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 order to retain enough information for refinement. 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 $front(ref) \leq ref'$ in these healthiness conditions, instead of $ref \leq ref'$. In addition, we have proved in [19] that for every action P in *Circus Time*, it can also be expressed as $\mathbf{R_{ct}}(\neg\, P_f^f \vdash P_f^t)$.

The full syntax, definitions and detailed explanations of *Circus Time* can be found in [19]. Here, we briefly introduce some operators that are used in the following sections. The action *Skip* terminates immediately without changing anything. *Stop* represents a deadlock, but allows time to elapse. *Chaos* is the worst action (the bottom element in the refinement ordering) whose behaviour is arbitrary, but satisfies $\mathbf{R_t}$. *Miracle* is the top element that expresses an unstarted process. This primitive operator is not included in the standard failures-divergences model of CSP. *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 external choice, $P \,\Box\, Q$, may behave either like the conjunction of P and Q if no external event has been observed yet, or like their disjunction if the decision has been made. The hiding action $P \setminus CS$ will behave like P, but the events within the set CS become invisible.

Here, we use the definitions of simple prefix and normal prefix, which are used in Section 3.1 and 3.2, to demonstrate how the reactive design semantics captures behaviours of processes.

Theorem 1. *(***Simple prefix***)*

$$c.e \rightarrow \mathit{Skip} = \mathbf{R_{ct}}(\mathbf{true} \vdash \mathit{wait\_com}(c) \lor \mathit{terminating\_com}(c.e)) \tag{1}$$

$$\mathit{wait\_com}(c) \mathrel{\widehat{=}} (\mathit{wait}' \land \mathit{possible}(\mathit{ref}, \mathit{ref}', c) \land \frown/tr' = \frown/tr) \tag{2}$$

$$\mathit{possible}(\mathit{ref}, \mathit{ref}', c) \mathrel{\widehat{=}} \forall\, i : \#\mathit{ref} \,..\, \#\mathit{ref}' \bullet c \notin \mathit{ref}'(i) \tag{3}$$

$$\mathit{term\_com}(c.e) \mathrel{\widehat{=}} \begin{pmatrix} \neg\ \mathit{wait}' \land \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref}) \land \\ \mathit{diff}(tr', tr) = \langle\langle c.e\rangle\rangle \land \mathit{state}' = \mathit{state} \end{pmatrix} \tag{4}$$

$$\mathit{terminating\_com}(c.e) \mathrel{\widehat{=}} \begin{pmatrix} \mathit{term\_com}(c.e) \lor \\ ((\mathit{wait\_com}(c) \land \mathit{state}' = \mathit{state})\,;\, \mathit{term\_com}(c.e)) \end{pmatrix} \tag{5}$$

The precondition of the above definition is **true**, which means that a simple prefix never diverges. The postcondition states that, if it starts successfully, a simple prefix can behave in three different ways: first, the clause $\mathit{wait\_com}(c)$ expresses that it can wait for interaction from its environment and meanwhile communications over the channel c are not refused ($\mathit{possible}(\mathit{ref}, \mathit{ref}', c)$); second, the clause $\mathit{term\_com}(c.e)$ simply denotes that the event is executed immediately; third, the composition of $\mathit{wait\_com}(c)$ and $\mathit{term\_com}(c.e)$ means that it may wait for a while and then terminate with an event $c.e$. The difference of two traces is defined as $\mathit{diff}(tr', tr) \mathrel{\widehat{=}} \langle tr'(\#tr) - \mathit{last}(tr)\rangle \frown \mathit{tail}(tr' - \mathit{front}(tr))$. The reactive design of prefix is calculated by means of the simple prefix and sequential composition as follows.

Theorem 2. *(***Prefix***)*

$$c.e \rightarrow P = \mathbf{R_{ct}} \begin{pmatrix} \neg\ (\mathit{terminating\_com}(c.e)\,;\, \mathbf{R1_{ct}}(\neg\ \mathit{wait} \land \mathbf{R2_{ct}}(P_f^f))) \\ \vdash \\ (\mathit{wait\_com}(c) \lor \mathit{terminating\_com}(c.e))\,;\, \mathbf{R1_{ct}}(\mathbb{I} \lhd \mathit{wait} \rhd \mathbf{R2_{ct}}(P_f^t)) \end{pmatrix}$$

This theorem states that, from its precondition, $c.e \rightarrow P$ diverges if P does; otherwise, from its postcondition, it can wait for the interaction from its environment, execute $c.e$ right now or wait for a while to execute $c.e$, and then behave like P_f^t.

3 Reactive Designs of Interrupts

Hoare's CSP book [5] gives a generic interrupt operator, $P \triangle Q$, which allows P to execute, but it may be interrupted by the first external event from Q and the program control is simultaneously passed to Q. Thereafter, the standard models of CSP adopt a catastrophic interrupt, which is expressed as $P \triangle_c Q$. Here, the catastrophic event c is unique, and its occurrence can interrupt P. However, this simpler interrupt might not be convenient for specifying real-time systems. For example, a seminar room is booked for an hour. Therefore one hour later after the punctual start, the seminar has to be interrupted if the next session has been booked by someone else. But the speaker may continue his talk if no one turns up to use this room. This example may be described as follows in Timed CSP [14] that adopts the generic interrupt operator to satisfy time requirements.

$$SEMINAR \triangle (Wait\ 1\,;\, close \rightarrow Skip)$$

Note that the above process does not mean that the interrupt must occur after one hour because the occurrence of *close* depends on the environment. If we say that the seminar must finish after one hour no matter whether this room will be used then, the generic interrupt operator is not enough. Accordingly, a timed interrupt operator is introduced in Timed CSP to describe this scenario.

$$SEMINAR \triangle_1 Skip$$

which means the interruption must happen one hour later. That is to say, the timed interrupt operator is time-driven and out of control of its environment.

The UTP semantics of the catastrophe in CSP has been discussed in [10], and in Section 3.1 we will use the same idea to calculate its reactive design within the *Circus Time* model. The approach in [10] cannot be used for defining the generic interrupt operator. Therefore, in Section 3.2 we follow the idea in Timed CSP to deal with the generic interrupt operator to consider it a special kind of parallel composition. For the timed interrupt operator, it can still be treated as a sequential composition because Q happens exactly d time units later if P has not terminates.

3.1 Catastrophe

We use the same approach in [10] but with changes to accommodate time behaviours to generate a UTP definition for a catastrophic interrupt operator, which is then calculated to produce a reactive design. The general idea in [10] is to use a new healthiness condition **I3**, whose name simply reflects its relation to $\mathbf{R3_{ct}}$, to bring the catastrophic event forward to any waiting state of the interruptible process while this event is not refused by an alphabet extension. An **I3** healthy process can only execute while its predecessor is in an intermediate state, or can behave like a *Circus Time* identity if its predecessor terminates.

Definition 1. $\mathbf{I3}(P) = P \lhd wait \rhd \mathit{II}_{ct}$

Here we can clearly see **I3**'s relation to $\mathbf{R3_{ct}}$ by the law $\mathbf{R3_{ct}}(\mathbf{I3}(P)) = \mathit{II}_{ct}$ which states that an **I3** healthy process will behave as the identity if it is required to be $\mathbf{R3_{ct}}$ healthy. In addition, to make sure the interrupt event is not refused during the execution of the interruptible process, an alphabet extension operator is defined as

Definition 2

$$P^{+c} \mathrel{\widehat{=}} (P \wedge \mathit{possible}(\mathit{ref}, \mathit{ref}', c)); (\mathit{II} \lhd \mathit{wait} \rhd (\mathit{II}^{-\mathit{ref}} \wedge \mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref})))$$

The predicate $\mathit{II}^{-\mathit{ref}}$ is the relational identity without the variables *ref* and *ref′*. We use $\mathit{front}(\mathit{ref}') = \mathit{front}(\mathit{ref})$ to free the last element of refusals in P, since we usually request that the last refusal is arbitrary if a process terminates.

Furthermore, we develop a new predicate, $\mathit{interrupt}(c, Q)$, to describe that an event is forced to occur despite an apparent situation opposite to an ordinary action in *Circus Time*.

Definition 3. $\mathit{try}(c, Q) \mathrel{\widehat{=}} (\mathit{II} \lhd \mathit{wait}' \rhd \mathit{term\_com}(c)); Q$

The behaviour in $\mathit{try}(c, Q)$ is similar to the prefix operator in *Circus Time*. Compared with the simple prefix (Theorem 1), it simplifies the behaviour by terminating only with the immediate execution of c ($\mathit{term\_com}(c)$), or just behaving like the identity. Here, the usual non-refusal of c is achieved by the alphabet extension when sequentially composed with the interruptible action.

Definition 4. $\mathit{force}(c, Q) \mathrel{\widehat{=}} \mathbf{I3}(\mathit{try}(c, Q))$

The definition of $\mathit{force}(c, Q)$ in Definition 4 is an **I3**-healthy $\mathit{try}(c, Q)$ that states that it behaves as the identity (II_{ct}) when its predecessor terminates, and otherwise behaves like $\mathit{try}(c, Q)$. Thus, the predicate $\mathit{interrupt}(c, Q)$ is defined as a $\mathbf{CSP1_{ct}}$-healthy *force*, which considers divergences of its predecessor and Q.

Definition 5. $\mathit{interrupt}(c, Q) \mathrel{\widehat{=}} \mathbf{CSP1_{ct}}(\mathit{force}(c, Q))$

The definition for catastrophe is given as a sequential composition between the interruptible action P with an alphabet extension by augmenting the interrupt event c, and the newly-defined predicate $\mathit{interrupt}(c, Q)$.

Definition 6. $P \mathbin{\triangle_c} Q \mathrel{\widehat{=}} \mathbf{R3_{ct}} \circ \mathbf{CSP2_{ct}}(P^{+c}\,;\, \mathit{interrupt}(c, Q))$

Here, $\mathbf{R3_{ct}}$ restricts the bound of **I3**, and $\mathbf{CSP2_{ct}}$ requires that a divergence within this interrupt may also contain termination.

To calculate the reactive design of catastrophe, we adopt the theorem that any action P in *Circus Time* can be expressed as $\mathbf{R_{ct}}(\neg\, P_f^f \vdash P_f^t)$. For the reason of limited space, we show the calculation of its postcondition only (Lemma 3), and the full proof can be found in [19]. We give Lemma 1 and Lemma 2 that have no intuitive meaning but avoid verbose proofs. The reader who is interested in their proofs is referred to [19].

Lemma 1. $(P^{+c}\,;\, (\neg\, \mathit{ok} \wedge \mathit{RT}))_f = (P_f^f \wedge \mathit{possible}(\mathit{ref}, \mathit{ref}', c)); \mathbf{R1_{ct}}(\mathbf{true})$

Lemma 2. $P^{+c}\,;(ok \wedge try(c,Q) \wedge wait) =$
$$\begin{pmatrix}((P \wedge possible(ref,ref',c) \wedge ok' \wedge wait') \vee \\ ((P \wedge possible(ref,ref',c));(ok \wedge wait \wedge term\_com(c));(\neg\ wait \wedge Q_f)\end{pmatrix}$$

Lemma 3. $(P \triangle_c\ Q)^t_f =$
$$\begin{pmatrix}((P^f_f\ \wedge possible(ref,ref',c));\mathbf{R1_{ct}}(\mathbf{true}))\vee \\ (P^t_f \wedge (possible(ref,ref',c);front(ref')=front(ref)) \wedge \neg\ wait')\vee \\ (P^t_f\ \wedge possible(ref,ref',c) \wedge wait')\ \vee \\ ((P^t_f \wedge possible(ref,ref',c));(wait \wedge term\_com(c));(\neg\ wait \wedge Q^t_f))\end{pmatrix}$$

Proof.

$(P \triangle_c\ Q)^t_f$ [Def-6]

$= (\mathbf{R3_{ct}} \circ \mathbf{CSP2_{ct}}(P^{+c}\,;interrupt(c,Q)))^t_f$ [$\mathbf{R3_{ct}}$ and substitution]

$= (\mathbf{CSP2_{ct}}(P^{+c}\,;interrupt(c,Q)))^t_f$ [$\mathbf{CSP2_{ct}}$]

$= ((P^{+c}\,;interrupt(c,Q))\,;J)^t_f$ [J-split]

$= ((P^{+c}\,;interrupt(c,Q))^f \vee (ok' \wedge (P^{+c}\,;interrupt(c,Q))^t))^t_f$ [subs.]

$= (P^{+c}\,;interrupt(c,Q))^t_f$ [Def-5]

$= (P^{+c}\,;\mathbf{CSP1_{ct}}(force(c,Q)))^t_f$ [$\mathbf{CSP1_{ct}}$]

$= (P^{+c}\,;((\neg\ ok \wedge RT) \vee (ok \wedge force(c,Q))))^t_f$ [relational calculus]

$= (P^{+c}\,;(\neg\ ok \wedge RT))^t_f \vee (P^{+c}\,;(ok \wedge force(c,Q)))^t_f$ [Lemma 1]

$= ((P^f_f \wedge possible(ref,ref',c));\mathbf{R1_{ct}}(\mathbf{true}))\vee(P^{+c}\,;(ok\wedge force(c,Q)))^t_f$ [Def-1,4]

$= \begin{pmatrix}((P^f_f\ \wedge possible(ref,ref',c));\mathbf{R1_{ct}}(\mathbf{true}))\vee \\ (P^{+c}\,;(ok \wedge (try(c,Q) \lhd wait \rhd \mathbb{I}_{ct})))^t_f\end{pmatrix}$ [relational calculus]

$= ((P^f_f\ \wedge possible(ref,ref',c));\mathbf{R1_{ct}}(\mathbf{true})) \vee (P^{+c}\,;(ok \wedge \neg\ wait \wedge \mathbb{I}_{ct}))^t_f$
$\vee\ (P^{+c}\,;(ok \wedge try(c,Q) \wedge wait))^t_f$ [$\mathbb{I}_{ct}$ and propositional calculus]

$= ((P^f_f\ \wedge possible(ref,ref',c));\mathbf{R1_{ct}}(\mathbf{true})) \vee (P^{+c}\,;(ok \wedge \neg\ wait \wedge \mathbb{I}))^t_f$
$\vee\ (P^{+c}\,;(ok \wedge try(c,Q) \wedge wait))^t_f$ [Lemma 2]

$= \begin{pmatrix}((P^f_f\ \wedge possible(ref,ref',c));\mathbf{R1_{ct}}(\mathbf{true})) \vee (P^{+c}\,;(ok \wedge \neg\ wait \wedge \mathbb{I}))^t_f\ \vee \\ (P \wedge possible(ref,ref',c) \wedge ok' \wedge wait')^t_f\ \vee \\ ((P \wedge possible(ref,ref',c));(ok \wedge wait \wedge term\_com(c));(\neg\ wait \wedge Q_f))^t_f\end{pmatrix}$
[Def-2 and relational calculus]

$= \begin{pmatrix}((P^f_f\ \wedge possible(ref,ref',c));\mathbf{R1_{ct}}(\mathbf{true}))\ \vee \\ ((P \wedge (possible(ref,ref',c);front(ref')=front(ref)));(ok \wedge \neg wait \wedge \mathbb{I}))^t_f \vee \\ (P \wedge possible(ref,ref',c) \wedge ok' \wedge wait')^t_f\ \vee \\ ((P \wedge possible(ref,ref',c));(ok \wedge wait \wedge term\_com(c));(\neg\ wait \wedge Q_f))^t_f\end{pmatrix}$
[relational calculus]

$$= \begin{pmatrix} ((P_f^f \wedge possible(ref, ref', c)); \mathbf{R1_{ct}}(\mathbf{true})) \vee \\ (P \wedge (possible(ref, ref', c); front(ref') = front(ref)) \wedge \neg\ wait')_f^t \vee \\ (P \wedge possible(ref, ref', c) \wedge ok' \wedge wait')_f^t \vee \\ ((P \wedge possible(ref, ref', c)); (ok \wedge wait \wedge term\_com(c)); (\neg\ wait \wedge Q_f))_f^t \end{pmatrix}$$

[substitution and relational calculus]

$$= \begin{pmatrix} ((P_f^f \wedge possible(ref, ref', c)); \mathbf{R1_{ct}}(\mathbf{true})) \vee \\ (P_f^t \wedge (possible(ref, ref', c); front(ref') = front(ref)) \wedge \neg\ wait') \vee \\ (P_f^t \wedge possible(ref, ref', c) \wedge wait') \vee \\ ((P_f^t \wedge possible(ref, ref', c)); (wait \wedge term\_com(c)); (\neg\ wait \wedge Q_f^t)) \end{pmatrix}$$

In the postcondition of the catastrophic interrupt operator, the first clause captures the divergent behaviour of P that will be absorbed by the precondition of catastrophe; the second clause states that P terminates without an interrupt, in which $front(ref') = front(ref)$ makes the last element of ref' arbitrary; the third clause expresses that P has not terminated before c; and the last one states that P is interrupted by c and it sequentially behaves like Q.

Before we calculate the final reactive design for $(P \triangle_c Q)$, we make a change to the fourth clause in Lemma 3 by adding $state' = state$ into its first component of the sequential composition. Introduction of $state$ and $state'$ is one of the differences of the *Circus Time* model with the standard CSP models and Timed CSP. To retain some important refinement laws of CSP in *Circus Time*, such as the unit law for external choice $P \,\Box\, Stop = P$, we do not constrain $state'$ at any waiting state or deadlock. However, we, here, have to impose $state' = state$ to enable Q to gain the initial value of $state$ when P is interrupted. This change is reflected in Theorem 3.

Thus, we use the similar approach to calculate $(P \triangle_c Q)_f^f$, and finally get the following reactive design for the catastrophic interrupt operator.

Theorem 3

$$(P \triangle_c Q) = \mathbf{R_{ct}} \begin{pmatrix} \neg\ ((P_f^f \wedge possible(ref, ref', c)); \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\ ((P_f^t \wedge possible(ref, ref', c)); (wait \wedge term\_com(c)); (\neg\, wait \wedge Q_f^f)) \\ \vdash \\ (P_f^t \wedge (possible(ref, ref', c); front(ref') = front(ref)) \wedge \neg\ wait') \vee \\ (P_f^t \wedge possible(ref, ref', c) \wedge wait') \vee \\ \begin{pmatrix} P_f^t \wedge possible(ref, ref', c) \\ \wedge\ state' = state \end{pmatrix}; (wait \wedge term\_com(c)); (\neg\ wait \wedge Q_f^t) \end{pmatrix}$$

The precondition from the above definition states that either P diverges while the interrupt event c has not occurred, or P is interrupted by c, sequentially composed with the divergence of Q.

This reactive design is derived from rigorous calculation of Definition 6, which is fully based on the work in [10]. The validation of this definition is also similar

to that in [10] by proving that it respects a number of laws. For example, the step law (Law 1) for the catastrophic interrupt operator given in [5,13] is still valid, and its proof is fully based on the distributive and eliminative laws, and the approach adopted in [10].

Law 1. $(a \to P) \triangle_c Q = (a \to (P \triangle_c Q)) \Box (c \to Q)$

3.2 Generic Interrupt

In the generic interrupt, $P \triangle Q$, Q is executed concurrently with P until either P terminates the execution, or Q performs an interrupt event. However, the approach that we used in Section 3.1 cannot be applied here because some interruptions by a generic interrupt operator cannot be easily expressed by sequential composition, such as the example we give in Section 1. Alternatively, we use the idea in Timed CSP to consider it parallel composition.

The basic idea of the semantics for parallel composition in the UTP is parallel-by-merge. That is, by labelling the variables of P and Q, we make them become disjoint ($\alpha P \cap \alpha Q = \emptyset$), and then merge these different variables by synchronisation to generate the final observation. As usual, we label the dashed variables of P with 0 and Q with 1 as

$$(P\,;U0(out\alpha P) \wedge (Q\,;U1(out\alpha Q))_{+\{tr,ref\}}$$

The labelling process $Ul(m)$ simply passes dashed variables of its predecessor to labelled variables and also removes these dashed variables from its alphabet. Through the labelling process, the output alphabet of $Ul(m)$ consists of $l.m$ only. However, under some circumstances we do need the initial values of P or Q. For this reason, we expand the alphabet after the labelling process. For example, $P_{+\{n\}}$ denotes $P \wedge n' = n$. Here, we are only interested in tr and ref that will be used in the merge operation.

First of all, we consider the merge function of timed traces and the sequences of refusals of P and Q.

$$ISync(\langle\rangle, S_2, ref_1, ref_2) = (\langle\rangle, ref_1) \tag{6}$$

$$ISync(S_1, \langle\rangle, ref_1, ref_2) = (S_1, ref_1) \tag{7}$$

$$\begin{aligned} &ISync(\langle t_1\rangle \frown S_1, \langle t_2\rangle \frown S_2, \langle r_1\rangle \frown ref_1, \langle r_1\rangle \frown ref_2) \\ &\quad = (\langle t_1\rangle, \langle r_1 \cap r_2\rangle) \odot ISync(S_1, S_2, ref_1, ref_2) \quad \textit{iff}\ \ t_2 = \langle\rangle \end{aligned} \tag{8}$$

$$ISync(\langle t_1\rangle \frown S_1, \langle t_2\rangle \frown S_2, ref_1, ref_2) = (\langle t_2\rangle \frown S_2, ref_2) \quad \textit{iff}\ \ t_2 \neq \langle\rangle \tag{9}$$

where $\odot$ is a new operator to concatenate a sequence of pairs, defined as

$$(S_1, ref_1) \odot (S_2, ref_2) = (S_1 \frown S_2, ref_1 \frown ref_2) \tag{10}$$

In *Circus Time*, we split a failure into a trace and a refusal for the convenience of expression or even simpler mechanisation. Unfortunately, here we have to reunite them again as a pair because they are manipulated together. Note that *ISync*

does not support commutativity. The rule (6) states that P has no further trace to interact with Q. That is, P may terminate or diverge in practice. The rule (7) just describes a similar situation for Q. The rule (8) presents the behaviour that no interrupt happens within the current time unit. The rule (9) underlines that Q interrupts P.

We also consider the values of ok' and $wait'$ that are determined by whether the interrupt has occurred or not. For example, their values are those of Q if interrupted. Otherwise, we take those of P. Hence, we define two predicates to show whether one trace can interrupt another.

$$enable(tr, 0.tr, 1.tr) \mathrel{\widehat=} \exists\, tr_0 \bullet \begin{pmatrix} tr_0 \leq diff\,(1.tr, tr) \land \frown\!/front(tr_0) = \langle\rangle \land \\ last(tr_0) \neq \langle\rangle \land \#tr_0 \leq \#diff\,(0.tr, tr) \end{pmatrix}$$

$$disable(tr, 0.tr, 1.tr) \mathrel{\widehat=} \left(\exists\, tr_0 \bullet \begin{pmatrix} (\frown\!/1.tr = \frown\!/tr \land \#1.tr \leq 0.tr) \lor \\ tr_0 \leq diff\,(1.tr, tr) \land \\ \frown\!/tr_0 = \langle\rangle \land \#diff\,(0.tr, tr) \leq tr_0 \end{pmatrix} \right)$$

The predicate *enable* states that if there exists a subsequence of $diff\,(1.tr, tr)$, which has not executed any event ($\frown\!/front(tr_0) = \langle\rangle$) except for the last element ($last(tr_0) \neq \langle\rangle$), and meanwhile $0.tr$ has not terminated ($\#tr_0 \leq \#diff\,(0.tr, tr)$), we conclude that $1.tr$ can interrupt $0.tr$ and the merge of them must finish with the rule (9). The predicate *disable* states that $1.tr$ cannot interrupt $0.tr$ if, either that the length of $1.tr$ is shorter or equal to the length of $0.tr$ and has not executed any external events, or that there exists a subsequence of $diff\,(1.tr, tr)$, which contains empty traces ($\frown\!/tr = \langle\rangle$) only and whose length is longer or equal to $\#diff\,(0.tr, tr)$.

We consider the merge predicate of the postcondition first, which describes non-divergent behaviours. If P does not diverge and Q cannot interrupt P, no matter Q can diverge or not, the behaviour will not become divergent. In the meantime the values of ok' and $wait'$ depend on those of P.

$$IM1 \mathrel{\widehat=} \begin{pmatrix} IMTR(tr,tr',0.ref,1.ref,ref,ref',0.ref,1.ref) \land disable(tr,0.tr,1.tr) \\ \land\ ok' = 0.ok \land state' = 0.state \land wait' = 0.wait \end{pmatrix}$$

$$IMTR(tr, tr', 0.tr, 1.tr, ref, ref', 0.ref, 1.ref) \mathrel{\widehat=} \left(\begin{pmatrix} diff\,(tr', tr), \\ ref' - front(ref) \end{pmatrix} = ISync \begin{pmatrix} diff\,(0.tr, tr), diff\,(1.tr, tr), \\ 0.ref - front(ref), 1.ref - front(ref) \end{pmatrix} \right)$$

Similarly, if Q does not diverge but does interrupt P, the behaviour is still stable regardless of the state of P.

$$IM2 \mathrel{\widehat=} \begin{pmatrix} IMTR(tr,tr',0.ref,1.ref,ref,ref',0.ref,1.ref) \land enable(tr,0.tr,1.tr) \\ \land\ ok' = 1.ok \land state' = 1.state \land wait' = 1.wait \end{pmatrix}$$

For the precondition of the reactive design, we are only interested in the divergence of P if the interruption has not happened, and the one of Q if it has done.

As a result, the divergent behaviour can be captured as follows.

$$\exists \begin{pmatrix} 0.tr, 0.ref, \\ 1.tr, 1.ref \end{pmatrix} \bullet \begin{pmatrix} P_f^f[0.tr, 0.ref/tr', ref'] \wedge Q_f[1.tr, 1.ref/tr', ref'] \\ \wedge\ disable(tr, 0.tr, 1.tr) \wedge \\ IMTR(tr, tr', 0.tr, 1.tr, ref, ref', 0.ref, 1.ref) \end{pmatrix} ; \mathbf{R1_{ct}}(\mathbf{true})$$

$$\exists \begin{pmatrix} 0.tr, 0.ref, \\ 1.tr, 1.ref \end{pmatrix} \bullet \begin{pmatrix} P_f[0.tr, 0.ref/tr', ref'] \wedge Q_f^f[1.tr, 1.ref/tr', ref'] \\ \wedge\ enable(tr, 0.tr, 1.tr) \wedge \\ IMTR(tr, tr', 0.tr, 1.tr, ref, ref', 0.ref, 1.ref) \end{pmatrix} ; \mathbf{R1_{ct}}(\mathbf{true})$$

Thus, the integrated definition of the interrupt operator is a combination of the above cases, including an extra predicate to tackle the immediate divergence of P or Q. That is, the divergent cases are given in the precondition, and the other are given in the postcondition.

Definition 7

$$P \triangle Q \mathrel{\widehat{=}}$$

$$\left(\begin{array}{c} \neg\ (((P_f^f \vee Q_f^f) \wedge tr' = tr)\,;\, \mathbf{R1_{ct}}(\mathbf{true})) \wedge \\ \neg\ \exists \begin{pmatrix} 0.tr, 0.ref, \\ 1.tr, 1.ref \end{pmatrix} \bullet \begin{pmatrix} P_f^f[0.tr, 0.ref/tr', ref'] \wedge \\ Q_f^t[1.tr, 1.ref/tr', ref'] \wedge \\ disable(tr, 0.tr, 1.tr) \wedge \\ IMTR \begin{pmatrix} tr, tr', 0.tr, 1.tr, \\ ref, ref', 0.ref, 1.ref \end{pmatrix} \end{pmatrix} ; \mathbf{R1_{ct}}(\mathbf{true}) \wedge \\ \neg\ \exists \begin{pmatrix} 0.tr, 0.ref, \\ 1.tr, 1.ref \end{pmatrix} \bullet \begin{pmatrix} P_f^t[0.tr, 0.ref/tr', ref'] \wedge \\ Q_f^f[1.tr, 1.ref/tr', ref'] \wedge \\ enable(tr, 0.tr, 1.tr) \wedge \\ IMTR \begin{pmatrix} tr, tr', 0.tr, 1.tr, \\ ref, ref', 0.ref, 1.ref \end{pmatrix} \end{pmatrix} ; \mathbf{R1_{ct}}(\mathbf{true}) \\ \vdash \\ ((P_f^t\,;\, U0(out\alpha P)) \wedge (Q_f^t\,;\, U1(out\alpha Q)))_{+\{tr,ref\}}\,;\, (IM1 \vee IM2) \end{array} \right)$$

Here, we use the $\mathbf{CSP2_{ct}}$**-converge** law that is proved in [19], $P^t = P^t \vee P^f$ if P is $\mathbf{CSP2_{ct}}$ healthy, to replace P_f and Q_f with P_f^t and Q_f^t respectively.

The generic interrupt operator is unexpectedly different from the catastrophic interrupt operator even if we make the interrupting action as $c \to Q$. In fact, their refinement can be expressed as

$$P \triangle (c \to Q) \sqsubseteq P \triangle_c Q \tag{11}$$

since $P \triangle (c \to Q)$ contains more behaviours. The proof of this refinement can be found in [19]. The idea adopted in this paper to calculate the definition of catastrophe is to consider Q sequentially composed with P but lifted forward to happen whenever P is waiting for the interaction. However, in *Circus Time*, P may execute an event immediately only so that it cannot be interrupted by means of the definition in Theorem 3. For example, the interruptible action in Lemma 4 is not interruptible, and its proof can also be found in [19].

Lemma 4. $((a \rightarrow Skip) \Box Miracle) \triangle_c Q = ((a \rightarrow Skip) \Box Miracle)$

Here, *Miracle* can force the event a to occur immediately and then terminate. More discussion about the interaction between *Miracle* with other operators can be found in [19]. The behaviour in Lemma 4 can be captured only by the first clause in the postcondition in Theorem 3. However, Lemma 4 does not hold for the generic interrupt operator, because the event c is able to interrupt as long as it occurs immediately too, via the rule 9 in *ISync*.

The generic interrupt in *Circus Time* can satisfy a number of algebraic laws in CSP. For example, it respects a step law, which is also proved by a similar approach as Law 1.

Law 2. $(a \rightarrow P) \triangle (c \rightarrow Q) = (a \rightarrow (P \triangle c \rightarrow Q)) \Box (c \rightarrow Q)$

Since *Stop* offers no external event, it can never interrupt any action. Similarly, if *Stop* is interruptible, only interrupt can occur.

Law 3. $(P \triangle Stop) = P = (Stop \triangle P)$

If *Skip* is the interrupting action, similar to *Stop*, the interrupt always behaves just like the interruptible action.

Law 4. $P \triangle Skip = P$

However, in *Circus Time*, *Skip* can be interrupted because we can allow events to happen without any delay, which can even occur prior to the start of *Skip*.

Law 5. $Skip \triangle P \sqsubseteq Skip$

In addition, the divergent action cannot be cured by interrupting it, or it is not safe to specify a divergent action after the interrupt.

Law 6. $(P \triangle Chaos) = Chaos = (Chaos \triangle P)$

The proofs of Law 2–Law 6 rely on the unfolding of the reactive designs of the generic interrupt operator and related operators. The hand-written proofs of these laws can be found in [19].

3.3 Timed Interrupt

A timed interrupt, $P \triangle_d Q$, allows P to run for no more than a particular length of time, and then performs an interrupt to pass the control of the process to Q. Compared with the event-driven interrupt where the environment of the process can prevent the interrupt event from happening, this time-driven interrupt cannot be avoided (if P does not terminate before time d) since its environment is not involved. The timed interrupt operator can be defined via the event-driven interrupt and hiding as follow

Definition 8. $P \triangle_d Q \mathrel{\widehat{=}} (P \triangle Wait\ d\,; (e \rightarrow Q)) \setminus \{e\} \qquad e \notin \alpha(P) \cup \alpha(Q)$

where the special event e becomes urgent to interrupt P immediately after d time units.

However, to avoid the complex semantics introduced by hiding, we directly give its definition to describe the behaviour of the timed interrupt, rather than calculating its reactive design from Definition 8.

Definition 9

$$P \triangle_d Q \mathrel{\widehat{=}} \left(\begin{array}{c} \neg\left((P_f^f \wedge \#tr' - \#tr \leq d)\,;\, \mathbf{R1_{ct}}(\mathbf{true})\right) \wedge \\ \neg\left((P_f^t \wedge \#tr' - \#tr = d)\,;\, (wait \wedge \mathit{II}^{-wait} \wedge \neg\, wait')\,;\, Q_f^f\right) \\ \vdash \\ (P_f^t \wedge \#tr' - \#tr \leq d) \vee \\ \left(\left(\begin{array}{c} P_f^t \wedge \#tr' - \#tr = d \\ \wedge\ state' = state \end{array}\right)\,;\, (wait \wedge \mathit{II}^{-wait} \wedge \neg\, wait')\,;\, Q_f^t\right) \end{array} \right)$$

The precondition of the above reactive design states that if P does not diverge within d time units, and if Q does not diverge after an interruption, and the postcondition guarantees the observation of P within d, or the sequential composition of the observation of P before the interruption (during d time units) and the behaviour of Q. In fact, the timed interrupt is quite like a kind of sequential composition because the interrupting action has no influence on the interruptible action.

The validation of this reactive design semantics for the timed interrupt operator is retained by proving a number of algebraic laws. There are some interesting laws for timed interrupts and delays. Law 7 states that a delay can be lifted forward from the interruptible action if its duration is no longer than the allowed waiting time units. Law 8 states that $\triangle_d$ can be eliminated if a delay has still not terminated when an interruption occurs. Law 9 states that a timed interrupt can be sequentialised if the interruptible action is *Stop*.

Law 7. $(Wait\ d\,;\, P) \triangle_{d+d'} Q = Wait\ d\,;\, (P \triangle_{d'} Q)$

Law 8. $(Wait\ (d + d')\,;\, P) \triangle_d Q = Wait\ d\,;\, Q$

Law 9. $Stop \triangle_d P = Wait\ d\,;\, P$

In addition, we have three zero laws for $\triangle_d$. Law 10 states that a divergence cannot be recovered, Law 11 states that termination can eliminate $\triangle_d$ if an interruption has not occurred, and Law 12 states that $\triangle_d$ cannot make an unstarted action start.

Law 10. $Chaos \triangle_d P = Chaos$

Law 11. $Skip \triangle_d P = Skip\ if\ d > 0$

Law 12. $Miracle \triangle_d P = Miracle$

The proofs of Law 7–Law 12 and more detailed discussions about these laws can be found in [19].

4 Conclusion

The reactive designs of the three interrupt operators developed in this paper carry on our previous work [16,17,18] to enhance the expressiveness of the *Circus Time* model. The well-established semantics of these interrupt operators is

significant in proving refinement laws in *Circus Time*. In line with the strong capability to deal with data in *Circus Time*, each action has two observational variables, *state* and $state'$, in its alphabet to record values of local variables. The two observational variables are carefully constrained in order to respect some important refinement laws. For example, we allow $state'$ to be arbitrary at any waiting state or deadlock to retain the unit law, $P \Box Stop = P$. That is to say, we can observe the value of $state'$ only when a program terminates. This claim, here, affects the mechanism for handling local variables within the interrupt operators. For example, a program in *Circus Time* may have the following behaviour involving updates of a program variable in an interruption.

$$x := 0;\ ((x := x + 1;\ Stop) \triangle (c \rightarrow x := x + 1))$$

The value of x when the program terminates is 1 if c occurs. From the reactive designs of the three kinds of interrupt operators, the interrupting action always obtains *state* from the beginning of an interrupt operator, no matter whether the interruptible action has changed the values of local variables. In other words, the interrupt operators in *Circus Time* cannot capture interruptions, such as a recovered program from a deadlock can proceed with the latest values of local variables before falling into the deadlock.

We demonstrate that these reactive designs of the interrupt operators preserve all related properties in [5,13,14], and also discuss their relations. Different from the approach used in the work [10], we adopt the parallel-by-merge to define the generic interrupt operator. However, establishing the proof of parallelism is always complicated. For simpler cases, the catastrophic interrupt operator is recommended if enough. A number of algebraic laws of the three operators have been proved by hand. As such hand-proofs are well-knowingly error prone, the mechanised proofs in a theorem prover is of course our future work in a short term. The work of mechanising *Circus* in ProofPower [12] and Isabelle [4] can help us to embed this semantics.

Acknowledgments. I thank Jim Woodcock for his advice on the semantics. This work was fully supported by the hiJaC project (EPSRC-EP/H017461/1).

References

1. Roscoe, A.W.: Model-checking CSP. In: A Classical Mind: essays in Honour of C.A.R. Hoare, ch. 21. Prentice-Hall (1994)
2. Cavalcanti, A.L.C., Woodcock, J.C.P.: A Tutorial Introduction to CSP in *Unifying Theories of Programming*. In: Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2004. LNCS, vol. 3167, pp. 220–268. Springer, Heidelberg (2006)
3. Cavalcanti, A.L.C., Sampaio, A.C.A., Woodcock, J.C.P.: A Refinement Strategy for *Circus*. Formal Aspects of Computing 15(2-3), 146–181 (2003)
4. Feliachi, A., Gaudel, M.-C., Wolff, B.: Isabelle/Circus : A Process Specification and Verification Environment. Technical Report 1547, LRI, Université Paris-Sud XI (November 2011), `http://www.lri.fr/Rapports-internes`

5. Hoare, C.A.R.: Communicating Sequential Processes. Prentice-Hall International (1985)
6. Hoare, C.A.R., Jifeng, H.: Unifying Theories of Programming. Prentice-Hall International (1998)
7. Huang, Y., Zhao, Y., Shi, J., Zhu, H., Qin, S.: Investigating time properties of interrupt-driven programs. In: Gheyi, R., Naumann, D. (eds.) SBMF 2012. LNCS, vol. 7498, pp. 131–146. Springer, Heidelberg (2012)
8. Jifeng, H.: From CSP to hybrid systems, pp. 171–189 (1994)
9. Leuschel, M., Butler, M.: ProB: A Model Checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
10. McEwan, A., Woodcock, J.C.P.: Unifying theories of interrupts. In: Proceedings of the Second UTP Symposium, Trinity College Dublin (2008)
11. Morgan, C.: Programming from specifications. Prentice-Hall, Inc., Upper Saddle River (1990)
12. Oliveira, M., Cavalcanti, A.L.C., Woodcock, J.C.P.: Unifying theories in ProofPower-Z. Formal Aspects of Computing Journal (2007)
13. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice-Hall International (1998)
14. Schneider, S.A.: Concurrent and real-time systems: the CSP approach. John Wiley & Sons (1999)
15. Sherif, A., Cavalcanti, A.L.C., Jifeng, H., Sampaio, A.C.A.: A process algebraic framework for specification and validation of real-time systems. Formal Aspects of Computing 22(2), 153–191 (2010)
16. Wei, K., Woodcock, J.C.P., Burns, A.: A timed model of *Circus* with the reactive design miracle. In: 8th International Conference on Software Engineering and Formal Methods (SEFM), Pisa, Italy, pp. 315–319. IEEE Computer Society (September 2010)
17. Wei, K., Woodcock, J.C.P., Burns, A.: Timed *Circus*: Timed CSP with the Miracle. In: ICECCS, pp. 55–64 (2011)
18. Wei, K., Woodcock, J.C.P., Cavalcanti, A.L.C.: *Circus Time* with reactive designs. In: UTP, pp. 68–87 (2012)
19. Wei, K., Woodcock, J.C.P., Cavalcanti, A.L.C.: New *Circus Time*. Technical report, Department of Computer Science, University of York, UK (March 2012), `http://www.cs.york.ac.uk/circus/hijac/publication.html`
20. Woodcock, J.C.P., Davies, J.: Using Z: Specification, Refinement and Proof. Prentice-Hall, Inc., Upper Saddle River (1996)
21. Woodcock, J.C.P., Cavalcanti, A.L.C.: A concurrent language for refinement. In: Butterfield, A., Pahl, C. (eds.) IWFM 2001: 5th Irish Workshop in Formal Methods, BCS Electronic Workshops in Computing, Dublin, Ireland (July 2001)
22. Woodcock, J.C.P., Cavalcanti, A.L.C.: The Semantics of $*Circus*$. In: Bert, D., Bowen, J.P., Henson, M.C., Robinson, K. (eds.) ZB 2002. LNCS, vol. 2272, pp. 184–203. Springer, Heidelberg (2002)
23. Woodcock, J.C.P., Cavalcanti, A.L.C.: A Tutorial Introduction to Designs in Unifying Theories of Programming. In: Boiten, E.A., Derrick, J., Smith, G.P. (eds.) IFM 2004. LNCS, vol. 2999, pp. 40–66. Springer, Heidelberg (2004)
24. Zhao, Y., Huang, Y., He, J., Liu, S.: Formal Model of Interrupt Program from a Probabilistic Perspective. In: ICECCS, pp. 87–94 (2011)