Abstract—We propose a timed model of Circus which is a compact extension of original Circus. Apart from introducing time, this model uses UTP-style semantics to describe each process as a reactive design. One of significant contributions of our timed model is to extensively explore the reactive design miracle, the top element of a complete lattice with respect to the implication ordering. The employment of the miracle brings a number of brand-new features such as strong deadline and uninterrupted events, which provide a more powerful and flexible expressiveness in system specifications.

Keywords—Circus, UTP, Timed CSP, Miracle

I. INTRODUCTION

Real-time systems are rather complicated as their components may execute in parallel and may also interact with their physical environment, as well as satisfy certain critical timing requirements. Over the past decades, formal methods have been remarkably successful in their application to the analysis of real-time systems. Broadly speaking, the approaches to the analysis of real-time systems have fallen into two camps. Process algebra approaches, such as timed CSP [1], [2], describe the behaviour of a system in terms of events, mostly addressing the interaction of its components. Temporal logic approaches, such as DC [3] and RTL [4], are convenient in describing the change of states but lack support for concurrency.

Recently the combination of different approaches by means of unifying their semantics has been developed in order to tackle a wider variety of systems. Circus is one of successful combinations, which unifies CSP [5], [6], [2] and Z [7], [8], so that it can define both data and behavioural aspects of a system; that is, it can describe the change of states and define the data operations while dealing with concurrency. A well-defined syntax and a sound semantics [9], [10] have been given in Circus, based on the semantics of UTP [11]. Sherif and He [12] subsequently develop a timed model of Circus that takes a subset of Circus and creates an abstraction function to map the timed model to the original untimed model.

In real-time programming, the deadline command [13] is a simple and flexible language primitive to directly express the desired timing behaviour. A similar operator is also provided in process algebra approaches such as timed CSP by means of the timeout operator. However, the occurrence of events in timed CSP depends on their environment’s interaction; in other words, we cannot specify that ‘something must occur’. When formalising observation of complex real-time systems, there are a variety of factors, such as physical laws and physical variables, which are part of systems but beyond the control of any other participants. For example, in the modelling of a scheduling system, a constant sampling frequency is achieved by executing the sampling tasks at precise points of time. Therefore, we possibly need to define a punctual clock whose clock-tick events must arrive on time. It is also very convenient to define ‘atomic’ actions or uninterrupted events in specification, which usually mean that the execution of a sequence of events cannot be interfered. Our new timed Circus is able to elegantly describe such behaviours that something must take place.

Woodcock et al [14] recently have proposed new semantics, also based on UTP, for Circus in which each process is described as a reactive design. The so-called reactive designs come from the fact that the new semantics applies the well-defined healthiness conditions of reactive processes to embed the theory of designs. The first denotational semantics, also based on UTP, was published in [10]. However it actually describes a Circus program as a Z specification in order to use tools like Z/EVES [15] to reason about properties. Unfortunately, this semantics is insufficient to prove refinement laws. The new semantics completely overcomes the weakness and even the sophisticated refinement laws of CSP can be adopted in the refinement of Circus specifications. Our timed model, built on such new semantics, is a compact extension of Circus. It does not inherit Z specifications and is in fact closer to timed CSP, but partially preserves the ability of Circus to handle data by allowing processes to hold a set of local variables.

There is a lot of related work that combines two or more well-researched languages for the specification of behaviour, data and time. For example, He [16] proposes a hybrid system to integrate CSP and DC for specifying the behaviour of continuous devices in a digital world, in which CSP describes the interaction of its components, whereas DC is for specification of continuous changes of states. Similar to Circus, many combinations focus on integrating CSP and Z or their extensions such as Timed CSP and Object-Z [17]. For example, Hoenicke and Olderog propose a combination CSP-OZ-DC in which CSP specifies behaviour of processes and their communications, Object-Z describes data and
state information, and DC captures real-time constraints. Different from the semantics of our timed model of Circus, their model translates all specifications of OZ and DC into a form of traces and acceptance sets and then uses a combined application of the model checkers FDR [18] and UPPAAL [19] to verify properties. By comparison, our timed model exploits the reactive design miracle to specify some brand-new features of a system, which cannot easily expressed by other approaches.

This paper is structured as follows. We begin with introducing the theories of designs and reactive processes in Section II. The semantics of our timed model and the new definitions of various combinators are presented in Section III by showing some strange processes and laws resulting from the reactive design miracle. In Section IV, we explore the nature of the miracle and use a few simple examples to demonstrate the brand-new features in system specifications when applying the reactive design miracle. Finally, Section V concludes the paper.

II. REACTIVE DESIGNS

In UTP, Hoare and He use the alphabetised relational calculus to give a denotational semantics that can explain a wide variety of programming paradigms. A relation \( P \) is a predicate with an alphabet \( \alpha P \), composed of undashed variables \((a, b, ...)\) and dashed variables \((a', x', ...)\). The former, written as \( inaP \), stands for initial observations, and the latter as \( outaP \) for intermediate or final observations. The relation is then called homogeneous if \( outaP = inaP' \), where \( inaP' \) is simply obtained by putting a dash on all the variables of \( inaP \).

Following the theory of alphabetised relations, many interesting sub-theories are constructed by applying healthiness conditions to characterise different aspects of the sub-theories. Hoare and He first build the theory of designs within the relational calculus. They subsequently build the theory of reactive processes that is disjoint from the theory of designs. The theory of CSP processes results from using reactive healthiness conditions to embed designs. Our timed model can also be considered an extension of the theory of CSP\(^1\), in which each process is described as a reactive design. The relation of these theories in UTP can be further illustrated in Figure 1 which is cited from [20].

In UTP a design is a relation that can be expressed as a precondition-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 precondition \( P \) and postcondition \( Q \) are predicates not containing \( ok \) and \( ok' \), a design with \( P \) and \( Q \), written as \( P \parallel Q \), is defined as follows:

\[
P \parallel Q \Leftrightarrow ok \land P \Rightarrow ok' \land Q
\]

which means 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 UTP are a collection of some fundamental laws that must be satisfied by relations belonging to the theory. They are usually expressed by means of an idempotent function \( \phi \). For example, a relation \( P \) is called \( \phi \)-healthy if \( \phi(P) = P \). There are four healthiness conditions identified by Hoare and He in the theory of designs, and here we discuss two of them only. A relation \( P \) is \( H1 \)-healthy if and only if \( H1(P) = P \) where the idempotent is defined as follows:

\[
H1(P) = (ok \Rightarrow P)
\]

which means observations can only be made after the program has started. This is simple but reasonable because it is impossible to make the observations if \( ok \) is false.

The second healthiness condition is

\[
H2 : [P[false/ok'] \Rightarrow P[true/ok']]\]

where square brackets denote universal quantification over all variables in the alphabet. Woodcock [21] rewrites \( H2 \) by removing the universal quantifier,

\[P = P ; J\]

where \( J \Leftrightarrow (ok \Rightarrow ok') \land v' = v \) and the semicolon denotes the sequential composition. Note that \( v \) and \( v' \) in the alphabet of \( J \) denotes all variables except for \( ok \) and \( ok' \). \( H2 \) actually states a design cannot require nontermination, since if \( P \) is satisfied when \( ok' \) is false, it must also be satisfied when \( ok' \) is true. Thus, a design is a relation that is \( H1-H2 \)-healthy. For a tutorial introduction to designs, the reader is referred to [21].

In UTP 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. For example,
if \(\text{wait}'\) is true, then the process is in an intermediate state.

If \(\text{wait}\) is true, it denotes an intermediate observation of its predecessor.

To record communications of a reactive process with its environment and time intervals over its observations, we need four additional observational variables: \(t, \text{tr}, \text{ref}\) and \(v\), which are explained in detail as follows:

- \(t\) and \(t'\) are the start point and end point respectively of a time interval over an observation of a process. Time is modelled as non-negative real numbers.
- \(\text{tr}\) specifies the trace of timed events in which a process has engaged until it starts, and \(\text{tr}'\) records all timed events that have occurred so far, up to the end of an observation. A trace is a sequence of timed events ordered by time.
- \(\text{ref}\) records the set of events that could be refused in the last observation; \(\text{ref}'\) contains the set of events that could be refused in the next observation.
- \(v\) represents the initial values of a process’s local variables, and \(v'\) records the final values.

In addition, we are able to represent any case of states of a process by combining the values of \(ok\) and \(\text{wait}\). For example, if \(ok'\) is false, the process diverges. Since a divergent process can do anything, there is no constraint on any of the dashed variables. If \(ok'\) is true, the state of the process depends on the value of \(\text{wait}'\). If \(\text{wait}'\) is true, the process is deadlocked; otherwise it successfully terminates if \(\text{wait}'\) is false. Similarly, the values of undashed variables represent the states of a process’s predecessor. For a more detailed introduction to the theory of reactive designs, the reader is referred to the tutorial [22].

There are three healthiness conditions in UTP that untimed reactive processes must satisfy. Our timed model simply inherits and extends them to embrace the factor of time. If a relation \(P\) describes a reactive process behaviour, \(R1\) states that it never changes history, or the trace is always extending.

\[
R1: \quad P = P \land \text{tr} \leq \text{tr}'
\]

The second healthiness condition, \(R2\), states that the undashed variable \(\text{tr}\) has no influence on the behaviour of the process, and therefore \(P\) is not changed if \(\text{tr}\) is an empty sequence.

\[
R2: \quad P(\text{tr}, \text{tr}') = P(\langle \rangle, \text{tr}' - \text{tr})
\]

where \(\text{tr}' - \text{tr}\) represents the trace of events that has occurred since the last observation, and \(P(\text{tr}, \text{tr}')\) means that \(\text{tr}\) and \(\text{tr}'\) satisfy \(P\).

The final healthiness condition, \(R3\), defines that a process should not start if its predecessor has not finished, while it preserves states unchanged.

\[
R3: \quad P = \Pi_{\text{rea}} \triangleright \text{wait} \triangleright P
\]

where the reactive identity, \(\Pi_{\text{rea}}\), is defined as follows:

\[
\Pi_{\text{rea}} \triangleq (\neg \, \text{ok} \land \text{tr} \leq \text{tr}' \land t \leq t') \lor \\
(\text{ok}' \land \text{tr}' = \text{tr} \land \text{ref}' = \text{ref} \land v' = v \land \\
\text{wait}' = \text{wait})
\]

which states that if \(\text{ok}\) is false, its predecessor diverges and then the only guarantee is that \(\text{tr}\) and \(t\) are extending; if \(\text{ok}'\) is true, it keeps states unchanged except that time can elapse. Further, if \(P\) and \(Q\) are predicates, \(P \triangleleft b \triangleright Q\) describes a program which behaves like \(P\) if the condition \(b\) is true, or like \(Q\) if \(b\) is false.

In consideration of our time model of reactive processes, additional healthiness conditions must also be satisfied in order to constrain the time and the behaviour of timed traces. As idempotent functions, they are defined as follow:

\[
R4(P) = P \land t \leq t' \\
R5(P) = \forall u \in \text{dom}(\text{tr}' - \text{tr}) \cdot t \leq u \leq t' \\
R6(P) = \forall i, j : i \leq j \\
\Rightarrow \, \text{strip}(\text{tr}' - \text{tr})(i) \leq \text{strip}(\text{tr}' - \text{tr})(j)
\]

where \(\text{dom}\) returns a collection of all time points when events occur in \(P\), and \(\text{strip}\) removes the event of a timed event and returns a sequence of time points. \(R4\) states time always moves forward; \(R5\) constrains that all the events taking place during the execution of the process happen within the correct time frame; \(R6\) requires that the events occur in an ascending order. As a result, \(P\) is a timed reactive process if and only if it is a fixed point of \(R \triangleq R1 \circ R2 \circ R3 \circ R4 \circ R5 \circ R6\), in which \(\circ\) is the function composition. As Figure 1, designs and reactive processes are disjoint relations because different healthiness conditions make them become different relations.

The miracle, the top element of the complete lattice in the implication ordering, is rather unexplored in UTP, as it can never be implemented in engineering practice. Nevertheless the miracle is extremely useful as a mathematical abstraction to specify and reason about properties of a system. For example, \(\text{false}\) is a miracle (the top element) in the complete lattice of \(\text{relations}\) because it can never give rise to any observation although it satisfies every specification. We also have the design miracle \((\text{true} \vdash \text{false})\) for the theory of designs, the reactive miracle \((R(\text{false}))\) for the theory of reactive processes and the reactive design miracle for the theory of CSP.

Hoare and He have given a new semantic to CSP in their UTP book [11] where the theory of CSP is a complete lattice, rather than the complete partial orders of the standard models of CSP [5], [6]. In UTP, the theory of CSP is built by applying a number of healthiness conditions to reactive processes. However it can also be achieved by using the healthiness conditions \(R\) to embed designs within the theory of reactive processes. As an extension of the theory of CSP, the theory of our timed model is built by the same approach,
in which processes are represented in the form of $R$-healthy designs. For example, the reactive design miracle $\top_R$ is defined in terms of the design miracle made $R$-healthy:

$$\top_R \equiv R(true \vdash false)$$

[design]

$$= R(ok \land true \Rightarrow ok' \land false)$$

[true unit for conjunction]

$$= R(ok \Rightarrow false)$$

[contradiction]

$$= R(\neg ok)$$

The top element of the theory of CSP in UTP, the reactive design miracle, had been completely unexplored until Woodcock [23] gave an insight into the nature of the reactive design miracle and provides a few interesting applications of miracles. Our new timed model further applies the reactive design miracle to show how useful it might be in developing timed reactive systems.

### III. Semantics

As an extension of Circus, our timed model is very close to timed CSP but capable of describing data and time. Most of the operators come from timed CSP, and the operators of assignment and guarded processes are obtained from Circus. A new operator, deadline, is defined by means of the reactive design miracle. The reactive design semantics based on UTP has been firstly proposed for CSP processes in [22], and the similar semantics for Circus in [10], [14]. Here we rewrite the semantics on account of the involvement of time.

The full syntax of our timed Circus is described by the following grammar:

$$P ::= \top_R \mid \bot_R \mid SKIP \mid STOP \mid a \rightarrow P \mid \assign{P} \mid \gall{P} \mid \update{P} \mid \assert{P} \mid [P] \mid \forall X.P$$

**Reactive design miracle**

The reactive miracle can never be executed and it is defined in terms of the design miracle made $R$-healthy:

$$\top_R \equiv R(true \vdash false)$$

**Chaos**

The semantics of CHAOS is the reactive abort which is the bottom element, defined as follows:

$$\bot_R \equiv R(false \vdash true)$$

**Stop**

The process $STOP$ is a deadlocked process which can neither perform anything nor refuse anything but allow time to elapse.

$$STOP \equiv R(true \vdash t' = tr \land wait')$$

**Skip**

The process $SKIP$ terminates immediately without changing the trace and the local variables:

$$SKIP \equiv R(true \vdash tr' = tr \land v' = v \land \neg wait' \land t' = t)$$

where the refusal set is irrelevant after termination and no time elapses here. However, $SKIP$ does allow time to pass if its predecessor diverges.

**Sequential composition**

The definition of sequential composition is the same as the one in alphabetised relational calculus. If two processes $P_1$ and $P_2$ are in sequence, $P_1$ is executed firstly and then $P_2$ once $P_1$ terminates, and meanwhile the final state of $P_1$ is passed on as the initial state of $P_2$.

$$P_1 ; P_2 \equiv \exists x_0 \bullet P_1[x_0/x'] \land P_2[x_0/x]$$

where $x$ is a set of variables including all variables used in this model.

In addition, the following useful laws involving the miracle in sequential compositions can be described as follows:

**Law 1.** $\top_R ; P = \top_R$

**Law 2.** $SKIP ; \top_R = \top_R$

Law 1 states that the reactive design miracle is a left-zero for relational composition and Law 2 says that the miracle is a right-zero for the reactive design identity.

**Prefix**

The process $a \rightarrow P$ in CSP denotes the process that behaves like $P$ after performing the event $a$. The prefix process in UTP can be represented by a composition of a simple prefix and $P$ itself, written as $(a \rightarrow SKIP) : P$. As the reactive design definition of simple prefix in [14], its timed version can be expressed as follows:

$$a \rightarrow SKIP \equiv R (true \vdash \begin{cases} tr' = tr \land a \notin ref' \\ < wait' \triangleright \begin{cases} wait' \equiv (t', a) \\ tr' = tr' \land (t', a) \end{cases} \land v' = v \end{cases})$$

In designs the postcondition always describes the behaviour when a process starts in a stable state. Meanwhile, if $wait'$ is true, it is waiting for the interaction with its environment; that is, no event occurs and $a$ is not in the refusal set. If $wait'$ is false, the process terminates with the result that the trace is extended.

**Assignment**

Suppose that $x$ is a program variable, and $e$ is an expression of program variables. The notation $(x := a e)$ represents the process that simply assigns the value of $e$ to $x$ and
terminates immediately, and then any other variables in the alphabet $A$ remain unchanged.

$$x :=_A e \triangleq R(\text{true} \vdash tr' = tr \land \neg wait' \land t' = t$$
$$\land x' = e \land y' = y \land \ldots \land z' = z)$$

where the set $A$ is defined as $A = \{x, y, \ldots, z, x', y', \ldots, z'\}$ and $\alpha(x :=_A e) = A$

**Guarded processes**

The process $g \& P$ has a boolean expression $g$ which must be satisfied before the process $P$ starts. The whole process starts if its predecessor terminates, but $P$ will not be executed unless $g$ is true. Such a process is defined as follows:

$$g \& P \triangleq R((g \Rightarrow \neg P'_f) \land ((g \land P'_f) \lor$$
$$\land (\neg g \land tr' = tr \land wait')))$$

where $P'_h$ represents $P[a/ok'][b/wait]$, and the abbreviation follows the definitions throughout this paper. For example, $P'_f$, representing $ok' = \text{false}$ and wait = false, denotes that $P$ diverges while its predecessor has terminated, and $P'_i$ that $P$ starts normally and is willing to terminate successfully.

**External choice**

The process $P_1 \parallel P_2$ may behave either like the conjunction of $P_1$ and $P_2$ if no event has been observed yet, or like their disjunction. Its reactive design semantics is defined as follows:

$$P_1 \parallel P_2 \triangleq R((\neg P'_f \land \neg P'_2) \lor$$
$$\land (P'_1 \land P'_2) \land tr' = tr \land wait' \lor P'_1 \lor P'_2))$$

The precondition of the design states neither $P_1$ nor $P_2$ can diverge; the postcondition represents that the observation is agreed by both $P_1$ and $P_2$ if the trace remains unchanged and the process is in an intermediate state; otherwise it behaves either like $P_1$ or like $P_2$ if the choice has been made.

There are two interesting laws with respect to the external choice of two primitive processes and the miracle. They are obvious but useful to understand the role of the miracle.

**Law 3. STOP $\parallel \top_R = \top_R$**

**Law 4. SKIP $\parallel \top_R = \text{SKIP}$**

**Input and output**

The output and input constructors are special cases of the prefix and external prefix choice operators. For example, if $c$ is a channel name of type $T$ and $v$ is a particular value, the process $c.v \rightarrow P$ outputting $v$ along the channel $c$ is equal to $c.v \rightarrow P$. The process $c?x : T \rightarrow P(x)$ describes a process which is ready to accept any value $x$ of type $T$, and it can also be expressed as a indexed external choice, $\square_{x \in T} c.x \rightarrow P(x)$.

**Internal choice**

The internal choice $P_1 \sqcap P_2$ can behave either like $P_1$ or like $P_2$, but it is out of control of its environment. It can be simply defined as $P_1 \sqcap P_2 \triangleq P_1 \lor P_2$.

**Law 5. $P \sqcap \top_R = P$**

**Parallel composition**

The process $P_1 \parallel P_2$ is the process where all events in the set $A$ must be synchronised, and events outside $A$ can execute independently. The parallel process terminates only if both $P_1$ and $P_2$ terminate, and it becomes divergent after either one of $P_1$ and $P_2$ does so.

The definition of parallel composition in the reactive-design style is the most complicated one, in which its precondition describes the behaviour of the process when it diverges, and its postcondition represents the parallel-by-merge semantics. For example, if $P_1$ diverges, we represent this case in the preconditions as follows:

$$\exists 1.tr', 2.tr' \bullet (P'_1; (1.tr' = tr)) \land (P'_2; (2.tr' = tr))$$
$$\land (1.tr' - tr) \models A = (2.tr' - tr) \models A$$

where $\models$ is the projection operator. Note that we are not interested in the divergence of $P_2$ because the whole process diverges as long as any of them does. In a very similar way, we can define the case that $P_2$ diverges.

The postcondition in the definition of parallel composition is based on the parallel-by-merge technique in UTP. The basic idea is to make processes become disjoint processes by labelling the shared variables of their alphabets, so that each process can execute independently. At the end of execution, these labelled variables are merged to produce the real values for the final observation. As a result, the integrated reactive design semantics of parallel composition is described as follows:

$$P_1 \parallel_A P_2 \triangleq$$
$$\left(\begin{array}{l}
\models (\neg \exists 1.tr', 2.tr' \bullet (P'_1; (1.tr' = tr)) \land (P'_2; (2.tr' = tr)) \land (1.tr' - tr) \models A = (2.tr' - tr) \models A)
\land
(\neg \exists 1.tr', 2.tr' \bullet (P'_1; (1.tr' = tr)) \land (P'_2; (2.tr' = tr)) \land (1.tr' - tr) \models A = (2.tr' - tr) \models A)
\end{array}\right)$$

In the postcondition $P'_1$ and $P'_2$ denote that they are not divergent. The labelling process $Ul(m)$ simply passes dashed variables of its predecessor to labelled variables, which is defined as $Ul(m) \triangleq \var l.m := m; \text{end} m$ where
\(\alpha Ul(m) \equiv \{m, lm'\}\). Notice that \(Ul(m)\) only obtains the values of dashed variables of its predecessor. However under some circumstances we do need the initial values of its predecessor’s variables. For this reason, we expand the alphabet after the labelling process. For example, \(P_+\{n\}\) denotes \(P \land n' = n\). Here we are only interested in \(v\) and \(tr\) that are used in \(M_1\).

The process \(M\parallel\) merges the traces and refusal sets from its predecessor in regard to the interface \(A\), and also combines other variables, defined as follows:

\[
M\parallel(A) \equiv \text{tr'} - \text{tr} \in (1.\text{tr} - \text{tr} \parallel 2.\text{tr} - \text{tr})
\]

\[
\land' = \max (1.t, 2.t)
\]

\[
\land (1.\text{tr} - \text{tr}) \parallel A = (2.\text{tr} - \text{tr}) \parallel A
\]

\[
\land\left(\begin{align*}
1.\text{wait} \land 2.\text{wait} \land \land' & \subseteq MRef \\
<\text{wait'} \equiv \sim 1.\text{wait} \land \sim 2.\text{wait} \land MSt
\end{align*}\right)
\]

The function \(tr_1 || tr_2\) states that \(tr_1\) and \(tr_2\) can combine in terms of \(A\) but must agree on \(A\). The set \(MRef\) calculates the corresponding refusal set defined as: \(MRef \equiv \{(1.\text{ref} \cup 2.\text{ref}) \cap A\} \cup \{(1.\text{ref} \cap 2.\text{ref}) \setminus A\}\). The predicate \(MSt\) describes how to merge the local variables \(v\).

An interleaving of two processes \(P_1\) and \(P_2\) executes each part independently and is equivalent to \(P_1\parallel P_2\). Another interesting law \(^2\) about the reactive design miracle is as follows:

**Law 6.** \(P\parallel \top_R = \top_R\)

**Hiding**

The process \(P \setminus A\) will pass through the same performance with \(P\), but events in the set \(A\) become invisible. The hiding operator is also not defined as a reactive design. Suppose that \(\Sigma\) is a universal set of events, and then the hiding is defined as follows:

\[
P \setminus A \equiv R(\exists s \bullet P[s, (\text{ref}' \cup A), t' / \text{ref}', \text{ref}', t'])
\]

\[
\land (t' - tr) = (s - tr) \mid (\Sigma \setminus A); \text{SKIP}
\]

The hiding may introduce divergence; therefore, the process \(\text{SKIP}\) is used to capture the observation of possible divergences. In addition, this operator forces hidden events to become urgent, occurring at the instant they are enabled. For instance, the unchanged \(t'\) means that the hidden events do not evolve at all.

**Delay**

The delay process \(\text{WAIT} d\) does nothing except that it allows an interval of time to pass. Its reactive design semantics is defined as follows:

\[
\text{WAIT} d \equiv R(\text{true} \equiv tr' = tr \land v' = v \land
\]

\[
(wait' < t' - t < d \equiv wait')
\]

where the trace \(tr\) and the local variable \(v\) always remain unchanged.

**Timeout**

The time-sensitive choice \(P_1 \triangleright \{d\}P_2\) resolves the choice in favour of \(P_1\) if \(P_1\) is able to execute observable events by the time \(d\), or resolves the choice in favour of \(P_2\). This operator can be defined by means of the external choice and the hiding operator:

\[
P_1 \triangleright \{d\}P_2 \equiv (P_1 \sqcup P_2) \setminus \{d\}
\]

which uses the event \(e\) to resolve the external choice if nothing happens in \(P_1\) by the time \(d\). Also, \(e\) is not included in the alphabets of \(P_1\) and \(P_2\).

**Deadline**

Compared with the above timeout operator, the deadline operator claims that \(P\) must perform observable events by the time \(d\) units, defined in terms of the reactive design miracle:

\[
P \triangleright d \equiv P \sqcup \{d\} \top_R
\]

This is really a very strong requirement in which there is no alternative but to meet the deadline; that is, within \(d\) time units, either \(P\) performs observable events or \(P\) terminates without doing anything, otherwise the process will never start.

**Interrupt**

The process \(P_1 \triangleright \{a\}P_2\) executes as \(P_1\), but at any stage before termination it can begin executing \(P_2\) if the event \(a\) occurs. We adopt the UTP semantics of the interrupt operator proposed by McEwan and Woodcock [25]. The basic idea of this semantics is to concatenate \(a\) and \(P_2\), such as \(a \rightarrow P_2\), and put it in any state of \(P_1\) using the external choice, whereafter the occurrence of \(a\) resolves the external choice to interrupt \(P_1\) and the process subsequently behaves like \(P_2\). For example, the following interrupting process illustrates such an idea:

\[
(b \rightarrow c \rightarrow \text{SKIP}) \triangleright \{a\}P_2 = (b \rightarrow (c \rightarrow \text{SKIP})
\]

\[
\sqcup a \rightarrow P_2)
\]

For inserting \(a\) into \(P_1\), they apply a new healthiness condition for interrupting processes as follows:

\[
I3(P) = P \triangleright \text{wait} \sqcup R
\]

which, opposite to \(R3\), states that \(P\) behaves like itself if its predecessor has not finished yet. This is a very strange healthiness condition, but it works surprisingly effective to

\(^2\)All the laws listed in the paper have been proved by hand. We omit the proof here due to the limitation of space. The interested reader can request the details from the authors or refers to the technical report [24].
bring forward $a \rightarrow P_2$. In their definition of the interrupt operator, they firstly redefine $a \rightarrow P_2$ by using $I3$ to force $a$ to be available in any state of its predecessor, expand the alphabet of $P_1$, and finally restrict $I3$ to be valid within $P_1$ and $P_2$. Our definition, entirely based on the above procedure, is given as follows:

$$\begin{align*}
\text{try}(a, P) & \equiv (a \notin \text{ref'} \land \\
\Pi < \text{wait} \triangleright tr' = tr \land \langle (t', a) \rangle) \land P \\
\text{force}(a, P) & \equiv I3 \triangleright \text{try}(a, P)
\end{align*}$$

where $\Pi$ is the relational identity that just keeps all variables unchanged, and $CSP1$, a healthiness condition proposed in [11], describes the behaviour after divergence, expressed as $CSP1(P) = P \lor (\neg a \land tr \leq t' \land t \leq t')$. In (III.1), $P_1^{+a}$ expands the alphabet of $P_1$ by adding $a$, and the combination of healthiness conditions $R$ restricts the boundary of $I3$.

**Recursion**

The set of processes in this model is a complete lattice with respect to the implication ordering where the top element is the reactive design abort CHAOS. Let $F(X)$ be a system description where $F$ is a monotonic function and $X$ is a system variable. The notation $\mu X.F(X)$ stands for the least fixed point of the equation $X = F(X)$.

**IV. Applications**

The reactive design miracle is rarely explored in the theory of reactive processes and the theory of CSP. The involvement of miracles with the combinators of CSP gives rise to some very strange processes, each of which violates axioms of the standard CSP failures-divergences model. Woodcock has discussed and proved a few such strange processes in [23]. For example, we combine the miracle with a simple prefix, and then get the following miraculous process:

$$a \Delta P \equiv CSP1(\alpha k \land \text{force}(a, P))$$

$$P_1 \{a\} P_2 \equiv R(P_1^{+a} \triangleright (a \Delta P_2)) \land a \notin \alpha P_1 \quad \text{(III.1)}$$

In our timed model, this process reveals more interesting features. Because of no constraint on timing in (IV.3), the event $a$ will occur when the environment is willing to interact with it. However, there is still no state between the start of the process and the occurrence of $a$; that is, it discretises continuous time. As traces in (IV.4), besides the absence of the empty trace, the value of $t'$ completely depends on the environment. One possible application of this strange process is that we can construct **uninterrupted** events or an **uninterrupted** trace in which either all of the elements can happen or none of them can happen individually, but the time when they occur is still controlled by their environment. For example, the traces of the following process are produced by applying (IV.3) twice:

$$T((a \rightarrow \text{SKIP} \square \top_R) ; (b \rightarrow \text{SKIP} \square \top_R)) \equiv \{(i, a), (i + j, b)\}$$

where $i$ is the time point when $a$ is offered and $i + j$ denotes the occurrence of $b$. The 'uninterrupted' property comes from the absence of the trace $\langle (i, a) \rangle$. Therefore, such a process either starts when both $a$ and $b$ are ready, or does not start at all.

**B. Urgent events and punctual clocks**

The strong deadline operator in our timed model is different from the deadline operator used in most other models. For example, the 'hard' deadline in real-time programming [13] plays a role of a compiler directive, which acts like an assertion to statically check whether a program will meet its timing requirements, and returns errors if these requirements are not satisfied. In timed CSP, the deadline operator is usually constructed by the timeout operator and the process $\text{STOP}$, i.e., the process will be deadlocked if the deadline is breached. By comparison, our strong deadline operator is constructed by the timeout operator and the
reactive design miracle. Due to the fact that the miracle cannot be executed, the strong deadline operator can push the process to the limit, or even drive the process to run the shorter path to meet the deadline. If the deadline cannot be satisfied anyway, the whole process will not start at all.

Setting the value of the hard deadline as zero can make a process or an event become urgent. Note that urgent events defined by the strong deadline are different from urgent internal events because they still have observable events. The substance of the urgency is to use the inexecutable reactive design miracle to force a process to instantly occur. For the sake of convenience, we use the following abbreviations as a shorthand to represent urgent events or processes:

\[ \uparrow P = P \gg 0 \]
\[ P_1 \uparrow P_2 \triangleq P_1 ; (P_2 \gg 0) \]
\[ a \uparrow b \triangleq (a \to \text{SKIP}) \uparrow (b \to \text{SKIP}) \]

Here the urgency operator squeezes the ‘distance’ of events and processes to zero. In fact, urgent events are an extreme instance of uninterrupted events.

One of significant contributions of the strong deadline operator to system specifications is that we can define punctual clocks which allow us to express discrete clock-tick events in the continuous-time environment. The timed CSP model is not likely to explicitly represent clock-tick events because it can never guarantee that an event is able to happen precisely at a specific time point. The occurrence of events in timed CSP depends on their environment’s interaction even if the timeout operator is applied. For example, a simple CSP process is defined as follows to denote that \( a \) immediately happens after one time units:

\[
C = \text{WAIT } 1 ; ((a \to \text{SKIP}) \triangleright \{0\} \text{STOP})
\]
\[
= \text{WAIT } 1 ; ((a \to \text{SKIP} \blacktriangledown \text{WAIT } 0) \land \{e\} \land \{e\})
\]
\[
= \text{WAIT } 1 ; ((a \to \text{SKIP} \blacktriangledown \tau \to \text{STOP})
\]

where, obviously, \( a \) may not occur because of the non-determinism. However, the situation can entirely change if we use the deadline by replacing \text{STOP} with the miracle:

\[
C' = \text{WAIT } 1 ; ((a \to \text{SKIP}) \triangleright \{0\} \top_R)
\]
\[
= \text{WAIT } 1 ; ((a \to \text{SKIP} \blacktriangledown \tau \rightarrow \top_R)
\]

where \( a \) must occur instantly after one time unit otherwise the process will behave like the miracle. The punctual clocks are the key factor to the unification of the continuous change of states and the discrete behaviour of events, which especially contributes to the semantics of hybrid systems.

C. Shared variables

In addition, the combination of urgency and parallel composition gives rise to a number of strange but extremely useful processes. For example,

\[ a \uparrow b \mid \mid c \to \text{SKIP} = (a \uparrow b ; (c \to \text{SKIP})) \blacktriangle (c \to a \uparrow b) \]

where \( c \) cannot occur between \( a \) and \( b \) in spite of the inter-leaving process. This unique feature can be applied to solve the tricky problem of how to communicate values of shared variables between different components. For example, we may define the following process which holds a shared variable and updates it by communication:

\[ SV(v) = \text{read}(v) \rightarrow SV(v) \blacktriangle \text{update}(x) \rightarrow SV(x) \]

which uses the channel read to give the value of \( v \) and the channel update to change the shared variable. In practice, we can use a locking mechanism to prevent other components from changing \( v \) when one component has occupied the variable. However, modelling the algorithm in specification languages is very hard and cumbersome. For example, it should be very careful to code the locking algorithm in CSP in case the system falls into deadlock. In our new model, we can simulate this algorithm by simply using urgent events or uninterrupted events. For instance, we may define two processes, \( P_1 \) and \( P_2 \), which can access the shared variable but not realise each other’s existence.

\[
P_1 = \text{read}(x) \uparrow \text{update}(x + 1)
\]
\[
P_2 = \text{read}(x) \uparrow \text{update}(x + 2)
\]
\[
P = (P_1 \mid \mid P_2) \blacktriangle SV(0)
\]

where the final value of \( v \) is 3 other than \( \{1,2,3\} \) because any pair of read and update can not be split. That is, \( P_1 \) and \( P_2 \) can exclusively access and change the value of the shared variable.

V. Conclusion

In this paper we present a timed model for Circus involving the reactive design miracle. We have established the reactive design semantics based on UTP and Woodcock’s previous work [14], [10], and explored various relations between miracles and some combinators of Circus.

Compared to other specification languages, this model largely enriches the expression for system specifications and is able to naturally describe some brand-new features of systems. For example, we are able to use the strong deadline operator to define urgent events, which cannot be properly represented in other process algebra approaches. In timed CSP two or more events may occur without any delay, but it completely depends on its environment. For instance, a process, \( a \to b \to \text{SKIP} \), may have a trace like \( \{\{t,a\},\{t,b\}\} \) if its environment is friendly. However, if its environment is unpredictable, there is no guarantee that the above process behaves in that way. Lawrence proposes CSPP [26] which is an extension of CSP and HCSP [27] to capture the semantics of hardware compilation. One of the interesting features of CSPP is that it allows true concurrency in which multiple events can occur instantly but without any order. Similarly, this approach cannot guarantee the instantaneous of events because the rest of the events can...
still happen if some of them have been blocked. The true meaning of instantaneity of events in our timed model is that not only are we unable to identify events by timing, but also they are so tightly attached that none of them can happen individually; in other words, if any of these events is blocked, none of them will happen.

An application of urgent events is to simulate the locking algorithm by making a sequence of events become ‘atomic’. Interestingly, this application has been partially achieved in some specification languages. For example, RAISE Specification Language (RSL) [28], [29] has an interlock operator which can prevent the interleaved processes from communicating with other processes until one of them terminates. Of course, the communication can take place between the locked processes if they are able to. Promela/SPIN [30], [31] can define atomic sequences which encapsulate a fragment of code to be executed uninterruptedly and individually. In the interleaving of process executions, no other process can execute statements from the moment that the first statement of an atomic sequence is executed until the last one has completed. Unfortunately, to our best knowledge, neither of the two operators has denotational semantics probably because of the insufficient capability of current languages to express the property that something must occur. Therefore, our time model is very likely to give denotational semantics to the two specification languages, so that the soundness of the languages can be proved and specification can be verified in theorem provers.

There are a number of directions that lead to future work. We will continue to explore the relationship between miracles and other combinators, study the role of miracles in healthiness conditions and discuss the refinement of the model. How useful the reactive design miracle is in the modelling of real-time systems has been demonstrated via some simple examples. We believe that these new features derived from the miracle can be applied to more applications. For example, we can simply extend the CSP model to include atomic events or uninterrupted events that are very useful in modelling industrial-scale examples when massive interleaved components are involved.

To prove the correctness and consistency of the model, we are undertaking a shallow embedding [32] of the semantics of our timed Circus in the theorem prover PVS, in order to mechanically prove the soundness of the model and a number of algebraic laws. One of our aims to develop this model is trying to find an approach to formalise the timebands model [33] in which a system is decomposed to reveal different behaviours in different time bands (granularities). For example, an event in a higher (coarser) band can be mapped into an activity with duration in a lower (finer) bands. To achieve the goal, it seems useful to use urgent events to maintain the consistency and coordination of the two bands. In addition, this model also provides an excellent platform for exploring a hybrid system that models continuous changes of the physical environment and communications of discrete events.

ACKNOWLEDGMENT

We would like to thank Ana Cavalcanti, Leo Freitas, Andrew Butterfield and Pawel Gancarski for discussions on the role of reactive miracles in programming logic, and thank Cliff Jones and Ian Hayes for discussion on the time band model and possible approaches of formalisation. This work was partially supported by INDEED project funded by EPSRC:Grant EP/E001297/1.

REFERENCES


