White Rose Research Online

eprints@whiterose.ac.uk
https://eprints.whiterose.ac.uk
Universities of Leeds, Sheffield and York

Deposited via The University of York.

White Rose Research Online URL for this paper:
https://eprints.whiterose.ac.uk/id/eprint/94630/

Version: Submitted Version

Proceedings Paper:
Wei, Kun, Woodcock, Jim and Burns, Alan (2011) Timed circus: Timed CSP with the miracle. In: Perseil, Isabelle, Breitman, Karin and Sterritt, Roy, (eds.) 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, Las Vegas, Nevada, USA, 27-29 April 2011. 16th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2011, 27-29 Apr 2011 IEEE Computer Society, GBR, pp. 55-64.

https://doi.org/10.1109/ICECCS.2011.13

Reuse
Items deposited in White Rose Research Online are protected by copyright, with all rights reserved unless indicated otherwise. They may be downloaded and/or printed for private study, or other acts as permitted by national copyright laws. The publisher or other rights holders may allow further reproduction and re-use of the full text version. This is indicated by the licence information on the White Rose Research Online record for the item.

Takedown
If you consider content in White Rose Research Online to be in breach of UK law, please notify us by emailing eprints@whiterose.ac.uk including the URL of the record and the reason for the withdrawal request.

2011 16th IEEE International Conference on Engineering of Complex Computer Systems

Timed Circus: Timed CSP with the Miracle

Kun Wei, Jim Woodcock and Alan Burns
Department of Computer Science
University of York, York, UK, YO10 5GH
{*kun, jim, burns*}*@cs.york.ac.uk*

***Abstract*—*Timed Circus* is a compact extension to *Circus*; that is, it inherits only the CSP part of *Circus* while introducing time. Although it looks much like timed CSP from the viewpoint of syntax, its semantics is very different from that of timed CSP because it uses a complete lattice in the implication ordering instead of the complete partial order of the standard failures-divergences model of CSP. The complete lattice gives rise to a number of strange processes which violate some axioms of CSP, especially when the miracle (the top element) and *SKIP* meet time. In this paper, compared with timed CSP, we will extensively explore such strange processes which turn out to be very useful in specifying a distinct property that 'something must occur'. Finally, we use a simple example to demonstrate how our model can contribute to modelling temporal behaviours with multiple time scales in complex systems.**

I. Introduction

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 the successful combinations, which unifies CSP [3], [7], [9] and Z [11], [16] and the refinement calculus [5], so that it can deal with 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. The first denotational semantics of *Circus*, based on UTP, was published in [15]. However it actually describes a *Circus* program as a Z specification in order to use tools like Z/EVES [8] to reason about properties. Later, Woodcock et al. [6] 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 embedding the theory of designs. The new semantics can adopt the sophisticated refinement laws of CSP in the refinement of *Circus* specifications.

We subsequently develop a new timed model [13] which is a compact extension of *Circus*. In fact, *timed Circus* does not inherit Z specification of *Circus*, and also makes use of the latest reactive design semantics in order to mechanically implement the refinement more easily. To some extent, *timed Circus* can be considered timed CSP with the miracle (the top element). However, its semantics is very different from that of timed CSP since it uses a complete lattice in the implication ordering instead of the complete partial order of the standard failures-divergences model of CSP. Prior to our work, Sherif and He [10] proposed a timed model of *Circus* that also took a subset of *Circus* and created an abstraction function to map the timed model to the original untimed model. However, our timed model uses different semantics for conveniently applying well-defined CSP refinement laws. Moreover, our model extensively explores the role of the reactive design miracle in system specifications, so that we can define brand-new operators to specify some distinct properties of a system which cannot be easily expressed by other approaches.

Hoare and He have given new semantics to CSP in their UTP book [4] where the theory of CSP is a complete lattice, rather than the complete partial orders of the standard models of CSP. Our *timed Circus*, based on the similar UTP semantics, overhauls the complete lattice and explores some elements which are usually considered useless in system specifications. For example, the miracle (*false*) in the theory of relations can never be implemented in engineering practice. Nevertheless this miracle is extremely useful as a mathematical abstraction to specify and reason about properties of a system. Woodcock [14] has intuitively discussed and proved some strange processes involving the reactive design miracle, each of which violates an axiom of the standard failures-divergences model of CSP. In this paper, we further discuss how the miracle impacts on the behaviour of a process particularly in a timed environment, how successful termination (*SKIP*) defined in *timed Circus* also results in some strange behaviours, and how these differences from timed CSP contribute to modelling temporal behaviours with multiple time scales in complex systems.

This paper is structured as follows. We begin with introducing the theories of designs and reactive processes in Section II. In Section III, we discuss our model's difference from timed CSP by exploring the nature of the miracle and the strange behaviour of successful termination. In Section IV we use a simple example to demonstrate how useful *timed Circus* is in dealing with temporal behaviours with multiple time scales, and in Section V we conclude the paper.

II. Reactive designs

In UTP, Hoare and He use the alphabetised relational calculus to give denotational semantics that can explain a

978-0-7695-4381-9/11 $26.00 © 2011 IEEE
DOI 10.1109/ICECCS.2011.13

IEEE computer society

wide variety of programming paradigms. A relation P is a predicate with an alphabet αP, composed of *undashed* variables $(a, b, ...)$ and *dashed* variables $(a', x', ...)$. The former, written as $in\alpha P$, stands for initial observations, and the latter as $out\alpha P$ for intermediate or final observations. The relation is then called *homogeneous* if $out\alpha P = in\alpha P'$, where $in\alpha P'$ is simply obtained by putting a dash on all the variables of $in\alpha P$.

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 $P \vdash Q$, is defined as follows:

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

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

A reactive process in UTP is a program whose behaviour may depend on interactions with its environment. To represent intermediate waiting states, a boolean variable *wait* is introduced to the alphabet of a reactive process. For example, if $wait'$ is true, then the process is in an intermediate state. If *wait* is true, it denotes an intermediate observation of its predecessor. Thus, we are able to represent any case of states of a process by combining the values of ok and *wait*. 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 $wait'$. If $wait'$ is true, the process is in an intermediate state; otherwise it successfully terminates if $wait'$ is false. Similarly, the values of undashed variables represent the states of a process's predecessor.

Timed reactive processes have another four pairs of observational variables: t, tr, ref, v, and their dashed conterparts. The tr and ref observations denote timed traces and its refusal sets. Note that a timed trace is a sequence of timed events which are pairs drawn from $\mathbb{R}^+ \times \Sigma$ (Σ denotes a universal set of events). A refusal is simply a set of events, rather than a set of time events in timed CSP, since other variables can assist in representing enough information of when those events are refused. The v observation expresses a process's local variables, and t represents a time point when observing the process. There are three healthiness conditions in UTP that untimed reactive processes must satisfy. Our timed model inherits and extends them to embrace the factor of time.

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

If a relation P describes a reactive process behaviour, $R1$ states that it never changes history, or the trace is always increasing. The second, $R2$, states that the undashed variable tr has no influence on the behaviour of the process, and therefore P does not change if tr is an empty sequence. The final healthiness condition, $R3$, defines that a process should not start if its predecessor has not finished, while it preserves states unchanged. Here, the reactive identity, II_{rea}, is defined as follows:

$$\begin{aligned} \mathit{II}_{rea} \mathrel{\widehat{=}} (\neg\, ok \wedge tr \leq tr' \wedge t \leq t') \vee (ok' \wedge tr' = tr \\ \wedge\, ref' = ref \wedge v' = v \wedge wait' = wait \wedge t' = t) \end{aligned}$$

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:

$$\begin{aligned} R4: &\quad P = P \wedge t \leq t' \\ R5: &\quad P = P \wedge \forall\, i, j : dom(tr' - tr) \bullet i \leq j \Rightarrow \\ &\qquad\qquad strip(tr' - tr)(i) \leq strip(tr' - tr)(j) \\ R6: &\quad P = P \wedge \forall\, u \in ran \circ strip(tr' - tr) \bullet t \leq u \leq t' \end{aligned}$$

where the function *strip* removes the event of each element of a timed trace and returns a sequence of time points, and $\circ$ is the function composition. $R4$ states time always moves forward;$R5$ requires that the events occur in an ascending order; $R6$ constrains that all the events taking place during the execution of the process happen within a correct time frame. As a result, P is a timed reactive process if and only if it is a fixed point of $R \mathrel{\widehat{=}} R1 \circ R2 \circ R3 \circ R4 \circ R5 \circ R6$. For a more detailed introduction to the theory of reactive designs, the reader is referred to the tutorial [2].

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. The theory of *timed Circus* is built by the same approach, in which processes are expressed in the form of R-healthy designs. For example, the reactive design miracle, which is the top element of our timed model, is defined in terms of the design miracle made R-healthy:

$$\begin{aligned} \top_R &\mathrel{\widehat{=}} R(true \vdash false) \\ &= R(ok \wedge true \Rightarrow ok' \wedge false) \\ &= R(ok \Rightarrow false) \\ &= R(\neg\, ok) \end{aligned}$$

Therefore, for a reactive design process[1], it also automatically satisfies the following two healthiness conditions:

$$\begin{aligned} &CSP1(P) = P \vee (\neg\, ok \wedge tr \leq tr' \wedge t \leq t') \\ &CSP2(P) = P\,;J \end{aligned}$$

[1]As proved in [2], a reactive process defined in terms of a design is always $CSP1$ healthy, and $CSP2$ is simply a recast of $H2$, a healthiness condition of the theory of designs.

where $J = (ok \Rightarrow ok') \wedge wait' = wait \wedge tr' = tr \wedge ref' = ref \wedge t' = t \wedge v' = v$. The first healthiness condition requires that, in case of divergence, extension of the trace and the time should be the only guaranteed property. The second one means that P cannot require nontermination, so that it is always possible to terminate.

Except for the deadline and assignment operators, the syntax of *timedCircus* is similar to that of timed CSP, as described by the following grammar:

$$P ::= \top_R \mid \bot_R \mid SKIP \mid STOP \mid a \rightarrow P \mid P_1 ; P_2 \mid x :=_A e \mid g \& P \mid P_1 \Box P_2 \mid P_1 \sqcap P_2 \mid P_1 \underset{A}{\parallel} P_2 \mid P \setminus A \mid WAIT\ d \mid P_1 \triangleright \{d\} P_2 \mid P \blacktriangleright d \mid P_1 \triangle \{a\} P_2 \mid \mu X.P$$

The miracle $\top_R$ is the top element in the implication ordering, which expresses a process that has not started yet. The bottom element $\bot_R$ is called *Abort* which can do absolutely anything. The process *STOP* is deadlocked and its only behaviour is to allow time to elapse. The process *SKIP* simply terminates immediately.

The sequential composition $P_1 ; P_2$ behaves as P_1 until P_1 terminates, and then behaves as P_2. In the meanwhile the final state of P_1 is passed on as the initial state of P_2. The prefix process $a \rightarrow P$ is able to execute the event a ($a \in \Sigma$) and then behaves as P. The process $g \& P$ has a boolean expression g which must be satisfied before P starts. The notation $(x :=_A e)$ represents that a process simply assigns the value of an expression e to a process variable x, and then any other variable in the alphabet A remains unchanged.

The process $P_1 \Box P_2$ behaves either like P_1 or P_2, but the first event of which can resolve the choice. Compared with the external choice, the internal choice $P_1 \sqcap P_2$ can also behave either like P_1 or like P_2, but it is out of control of its environment. Both external and internal choices have indexed choices. For example, if I is a finite indexing set such that P_i is defined for each $i \in I$, written as $\Box_{i \in I} P_i$. The indexing external choice is also used to define the input operator. 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 inputting process $c?x : T \rightarrow P(x)$ describes a process that is ready to accept any value x of type T, and it is defined as $\Box_{x \in T} c.x \rightarrow P(x)$.

The process $P_1 \underset{A}{\parallel} P_2$ is the process where all events in the set A must be synchronised, and the 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. An interleaving of two processes, $P_1 \,|||\, P_2$, executes each part independently and is equivalent to $P_1 \underset{\emptyset}{\parallel} P_2$. The hiding operator $P \setminus A$ makes the events in the set A become invisible or internal to the process. The process $P_1 \triangle \{a\} P_2$ behaves as P_1, but at any stage before its termination the occurrence of a ($a \notin \alpha P_1$) will interrupt P_1 and pass the program control to P_2. The recursive process $\mu X.P$ behaves like P with every occurrence of the system variable X in P representing a recursive invocation.

The delay process *WAIT d* does nothing except that it allows d time units to pass. The timeout operator $P_1 \triangleright \{d\} P_2$ resolves the choice in favour of P_1 if P_1 is able to execute observable (external) events by d time units, otherwise executes P_2. The deadline operator $\blacktriangleright$ is similar to the timeout operator, but it uses the miracle to force that *P must* execute observable events by d.

Here, we will not give detailed definitions to these operators because this paper focuses on how those primitive processes such as the miracle, termination and assignment lead to some strange behaviours when connected with other operators. For full explanation of all operators, the reader is referred to the technical report [12].

III. The difference from timed CSP

Although *timed Circus* is similar to timed CSP in syntax and also inherits assumptions of timed CSP such as maximal parallelism and maximal progress, the introduction of the miracle makes it different from timed CSP in many aspects.

A. The reactive design miracle

The miracle itself is a very 'strange' process since it expresses a process that has not started yet. However, it is very useful as a mathematical abstraction in reasoning about properties of a system. The semantics for the reactive design miracle introduced in Section II can be furthermore simplified somewhat:

$$\top_R = (tr \leq tr' \wedge t \leq t' \wedge \neg\, ok) \vee (wait \wedge ok' \wedge \mathit{II})$$

where II is called relational identity which simply means that all dashed variables in the alphabet are equivalent to correspondingly undashed variables. The observation of the miracle consists of two parts: the left part of the disjunction states that, since *ok* is false, its predecessor diverges and the miracle is in an unstable state; the second one states that the miracle is waiting for its predecessor's termination (e.g., *wait* is true) but in a stable state (e.g., ok' is true). However, in both cases, the miracle has not started yet.

In fact, the key idea of figuring out the role of the miracle in a process is that the program control should never meet the miracle if the process has started. This idea can be applies to intuitively getting a number of laws even without the semantic proofs[2].

1) Sequential composition with the miracle: Since the miracle never starts, the left zero law is immediately validated if P is a timed reactive process:

L1: $\top_R ; P = \top_R$

Also, the following two laws can be easily proved. For example, in **L2**, the process should never start because, once

[2]The proofs of all theorems and lemmas can be found in the appendix.

it starts, the program will inevitably meet $\top_R$ during the execution.

L2: $SKIP ; \top_R = \top_R$

L3: $STOP ; \top_R = STOP$

What happens if a process executes a delay and then behaves miraculously? Should it be a miracle too, similar to **L**2? The following theorem will provide an answer.

***Theorem** 1:*

$$WAIT\ d ; \top_R = R(true \vdash tr' = tr \wedge v' = v \wedge wait' \wedge t' - t < d)$$

The result is clearly feasible, so it is not a miracle but very interesting. This process behaves just like *STOP* when $t' - t < d$, and like the miracle when otherwise. However, does it contradict with our previous conclusion that a process should never meet the miracle during the execution? In fact, this process still preserves such a conclusion because the postcondition of the design in the semantics constrains that the observation after the process starts in a stable state should be made within d, and the control can never be passed over to $\top_R$ (since $wait'$ is always true). Note that this strange process is very useful in the definition of the deadline operator, which forces other participants to happen as soon as possible so as to avoid the miracle.

2) Prefixing the miracle: Woodcock [14] has proved that, when combining the miracle with a simple prefix, it violates an axiom of the failures-divergences model of CSP. For example, the semantics for the following simple process is given in [14]:

***Theorem** 2:*

$$a \rightarrow \top_R = R(true \vdash tr' = tr \wedge a \notin ref' \wedge wait' \wedge v' = v)$$

which states that the process never refuses to execute the event a, but it never actually does. Of course, such a strange process has the same behaviour in *timed Circus*. Notice that the process results in an unexpected fact to the timeout operator.

Usually, we use the external choice and the hiding operator to define the timeout, as follows:

$$P_1 \triangleright \{d\} P_2 \mathrel{\widehat{=}} (P_1 ; e \rightarrow SKIP \ \Box\ WAIT\ d ; e \rightarrow P_2) \setminus \{e\}$$

Obviously, if P_2 is replaced by $\top_R$, we are not going to see that the external choice is resolved by e ($e \notin (\alpha P_1 \cup \alpha P_2)$) because it will never happen. Therefore, to define the deadline operator, we can not simply replace P_2 with the miracle in the timeout definition.

3) External choice with the miracle: Another strange process, $(a \rightarrow SKIP) \ \Box\ \top_R$, is given in [14] when we offer a choice between the miracle and engaging in an event. In an untimed model this process performs the event a and terminates immediately. There is no state in which the process is waiting for the environment to offer a. It simply occurs instantly and no empty trace exists for such a process. Obviously, it violates another important axiom of the failures-divergences model of CSP where traces are prefix closed.

In *timed Circus*, this process reveals a more interesting feature that is revealed in the following theorem:

***Theorem** 3:*

$$(a \rightarrow SKIP) \ \Box\ \top_R = R(true \vdash \neg\ wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v)$$

The result is very similar to the one initially given by Woodcock [14], but a does not happen instantly as it does in an untimed environment. Because of no constraint on timing in the above semantics, 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, we observe nothing during $[t, t']$ and directly get the observation at the time point t'. Such an event is called an *urgent* event, and the miracle forces the event to become urgent. The traces (expressed by $\mathcal{T}$) of the process in Theorem 3 can partially illustrate the strange behaviour, as follows:

$$\mathcal{T}(a \rightarrow SKIP \ \Box\ \top_R) = \{ \langle (t', a) \rangle \mid t' \in \mathbb{R}^+ \}$$

where, besides the absence of the empty trace (because $wait'$ is always false.), the value of t' completely depends on the environment.

Understanding the behaviour of the process in Theorem 3 is very helpful for explaining the definition of the deadline operator which is described as follows:

$$P \blacktriangleright d \mathrel{\widehat{=}} (((P; e_1 \rightarrow SKIP) \ \Box\ (WAIT\ d; e_2 \rightarrow STOP)) \setminus \{e_2\} \ \Box\ WAIT\ d ; \top_R) \setminus \{e_1\}$$

which uses e_1 ($e_1 \notin \alpha P$) to resolve both two external choices if P does not execute external events and terminates before d, and e_2 ($e_2 \notin \alpha P$) to resolve the first external choice if P does nothing when d is due. For example, if $d = 0$ and P will execute an external event such as $a \rightarrow SKIP$, it can be simplified as follows:

$$\begin{aligned} &(a \rightarrow SKIP) \blacktriangleright 0 \\ =&(((a \rightarrow SKIP) ; e_1 \rightarrow SKIP \ \Box\ (e_2 \rightarrow STOP)) \setminus \{e_2\}) \\ &\ \Box\ \top_R) \setminus \{e_1\} \end{aligned}$$

where a must occur instantly, or e_2 will resolve the first external choice and then make the whole process behave like the miracle. In other words, this is really a very strong requirement in which there is no alternative but to meet the deadline, otherwise P will never start.

4) The miracle in parallel: What happens if we put the miracle and an ordinary process in parallel? It should be the miracle deduced from our intuitive conclusion that all processes participating in a parallel must start at the same

time, but the miracle can never start, and thereby the whole process can never start either.

L4: $P \underset{A}{\parallel} \top_R = \top_R$

B. Successful termination

Successful termination, *SKIP*, is the point that a process reaches when its execution has completed. However, the interpretation of *SKIP* is different between the standard CSP model and our timed *Circus* model. In CSP, a special event ✓ is used to purely denote termination, and so it is not an element of the universal alphabet Σ. In timed CSP, *SKIP* is immediately ready to terminate, or unable to refuse ✓, and meanwhile it remains in the same state except for the lapse of time. When *SKIP* is combined with other operators, the employment of ✓ enables these processes to behave as we expect, with a careful treatment. For example, a policy called *distributed termination* is used to make ✓ coordinate all participants to terminate when all have terminated.

The interpretation of *SKIP* in *timed Circus* is that it terminates instantly without changing anything, as its reactive design semantics is given:

$$SKIP \mathrel{\widehat{=}} R(true \vdash tr' = tr \wedge v' = v \wedge \neg\, wait' \wedge t' = t)$$

where the refusal set is irrelevant after termination and no time elapses here if it starts in a stable state. Note that *SKIP* does have an empty trace because of $tr' = tr$, though $wait'$ is false. We do not need the special event ✓ in the semantics, and so may remove those constraints on processes, e.g., ✓ is not a member of Σ, ✓ appears only in the end of a trace, and ✓ is always implicit in the interface when involved in a parallel. However *SKIP* in *timed Circus* gives rise to an unexpected behaviour when we offer a choice between it and an event.

Normally one would not write a process like $P \,\Box\, SKIP$, since it does not always have well-behaved executions. It is a bit strange to provide the environment with the choice of terminating or not because termination is something that the environment observes rather than controls. In CSP, the way to deal with this process is

$$P \,\Box\, SKIP \sqsubseteq SKIP \tag{III.1}$$

This says that whenever this process can terminate, it can do so and there is nothing that the environment can do to stop it.

In our timed model, $P \,\Box\, SKIP$ has a different behaviour, e.g., the following theorem shows a strange behaviour when making a choice between *SKIP* and a simple prefix.

Theorem 4:

$$\begin{aligned}&(a \rightarrow SKIP) \,\Box\, SKIP\\ &= R(true \vdash (\neg\, wait' \wedge tr' = tr \wedge v' = v \wedge t' = t) \vee\\ &\qquad (\neg\, wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v))\end{aligned}$$

The rule III.1 can also be applied to such a process in *timed Circus*, saying that the process can terminate whenever it can do so. By comparison with Theorem 3, the result of this theorem shows even more strange behaviour. It says that either the process behaves like *SKIP* to terminate instantly or a becomes urgent. In other words, similar to the miracle, *SKIP* can make an event become urgent but also nondeterministic.

In addition, an assignment in *timed Circus* has the same influence on other operators because of its similar semantics as *SKIP*.

$$\begin{aligned}x :=_A e \mathrel{\widehat{=}} R\,(true \vdash\ & tr' = tr \wedge \neg\, wait' \wedge t' = t\\ & \wedge x' = e \wedge y' = y \wedge \ldots \wedge z' = z)\end{aligned}$$

where the set A is defined as $A = \{x, y, \ldots, z, x', y', \ldots z'\}$ and $\alpha(x :=_A e) = A$; that is, x is a member of v.

As a result, we should avoid directly using these strange processes unless necessary. For example, $P \,\Box\, SKIP$ was banned in Hoare's book [3], but it was allowed in later versions of CSP [7], [9] to make the semantics work for more elaborate processes, e.g., $(P \,\Box\, (a \rightarrow SKIP)) \setminus \{a\}$. In this section, we have explored and proved its exact behaviour when combining with the external choice, which provides us with much confidence to generate the right operational semantics.

IV. Applications

The involvement of miracles with other operators of the timed *Circus* gives rise to some very strange processes, which violate some axioms of the standard CSP failures-divergences model. However, it provides more powerful and flexible expressiveness in system specifications, so that we can define some distinct properties which cannot be properly defined in other approaches.

A. Instant and uninterrupted events

The deadline operator in our timed model is different from the deadline operator used in most other models. In timed CSP, the deadline operator is usually constructed by the timeout operator and the process *STOP*, i.e., the process will be deadlocked if the deadline is breached. By comparison, our strong deadline operator is constructed by the reactive design miracle. Due to the fact that the miracle cannot be executed, the deadline operator can push the process to the limit, or even force the process to always choose qualified paths to meet the deadline. If the deadline cannot be satisfied anyway, the whole process will not start at all.

Setting the value of the deadline as zero can make a process or an event become *instant*. For the sake of convenience, we use the following abbreviations as a shorthand

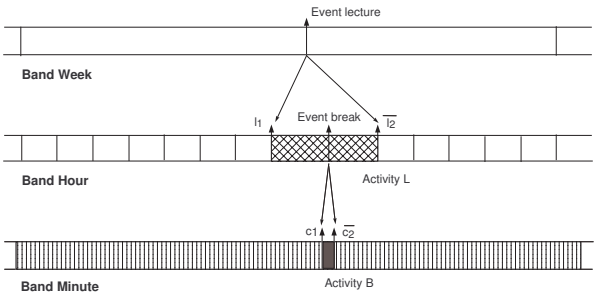

Figure 1. Mapping between different bands

to represent instant events or processes:

$$\ddagger P = P \blacktriangleright 0$$
$$P_1 \ddagger P_2 \mathrel{\widehat{=}} P_1 \,;\, (P_2 \blacktriangleright 0)$$
$$a \ddagger b \mathrel{\widehat{=}} (a \rightarrow \mathit{SKIP}) \ddagger (b \rightarrow \mathit{SKIP})$$

Here the instantaneity operator squeezes the 'distance' of events and processes to zero.

One possible application of instantaneity is that we can construct *uninterrupted* events in which either all events can happen or none of them can happen individually. For example, we can define a process as follows:

$$P = (a \blacktriangleright 0) \,;\, \mathit{WAIT}\ 1 \,;\, (b \blacktriangleright 0)$$

where a and b are uninterrupted events. That is to say, a can happen only if b can at an interval of one time unit. The traces of P are expressed as follows:

$$\mathcal{T}(P) = \{\langle (0, a), (1, b) \rangle\}$$

where the 'uninterrupted' property comes from the absence of the single trace $\langle (0, a) \rangle$.

B. Modelling temporal behaviours with timebands

Complex real-time systems exhibit dynamic behaviours on many different time levels. For example, circuits have nanosecond speeds for computation in a component, whereas slower functional units may take seconds to achieve their goals; moreover, the involvement of human activities related to calendar units such as days, weeks, months and even years may take more time. To overcome the weakness of traditional approaches which model dynamic temporal behaviours in a single flat time, Burns and Hayes [1] propose a timebands model in which a system is decomposed to reveal different behaviours in different time bands. Apart from defining time bands by granularities, a key aspect of the timebands framework is that events are considered to be instantaneous in a band, and then in a finer band they can be mapped into activities that have duration.

For example, to express a statement that *one week a lecturer has a lecture which takes two hours and has a five-minute break*, as illustrated in Figure 1, the timebands model specifies the lecture as an instantaneous event in a week band and subsequently maps it into an activity in an hour band. Furthermore, the break in the activity of the hour band is mapped again into an activity with five minutes in a minute band. This clearly allows dynamic temporal behaviours to be partitioned but not isolated from each other. Otherwise, we may have to state it in a very cumbersome way, e.g., within a period of 60*24*7 minutes, a lecture which takes 60*2 minutes has a five-minute break.

To deal with events and activities, we naturally choose process algebra approaches to formalise the timebands model. However, a few issues arise immediately when embedding time granularity in a process algebra approach. For example, how to maintain consistency of different time bands when mapping events to corresponding activities. Asynchronous occurrence of an event and an activity within their own bands definitely leads to inconsistency.

Our *timed Circus* provides an elegant solution to the formalism of the timebands model. In combination with the lecture example in Figure 1, we first define a signature event (e.g., l_2 is marked with an overline to denote the end of activity L) in the hour band, and then make event *lecture* in the week band *instant* to the signature event. Thus, the instantaneity of the two events guarantees coordination of the mapped event and the signature event of its corresponding activity. Intuitively, we find that l_1 in the hour band actually occurs earlier than *lecture* in the week band. Therefore, how can we restrict that l_1 simply occurs at a right time, thereby preserving integrity of activity L. For example, if we say that *lecture* occurs right at 10:00, and then the instant l_2 must occur at 10:00 as well, which in fact turns out to say that l_1 must occur at 8:00. That is, activity L should automatically locate its position in the hour band and l_1 just happens in a right time to maintain integrity of the activity. To achieve integrity of an activity, we simply require that all events in the activity are uninterrupted. For example, we can easily define that l_1 and l_2 are uninterrupted events with an interval of two hours. As a result, by means of the miracle, instant and uninterrupted events comprise the main framework of the formalism of the timebands model.

V. Conclusion

We have proposed a timed model of *Circus* involving the reactive design miracle, which can also be consider timed CSP with the miracle in UTP. Although Woodcock has given some properties of the miracle in [14], the behaviour caused by the miracle in a timed environment is more complex. In addition, successful termination and assignment also result in unexpected behaviours when connected with an external choice. The full proof of these strange behaviours given in this paper is very helpful in understanding our *timed Circus* and provides us with much confidence to generate the right operational semantics. Our *timed Circus* has the potential to tackle temporal behaviours with multiple time scales in

a complex system in terms of formalising the timebands model.

ACKNOWLEDGEMENTS

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

APPENDIX

As a matter of fact, we only need $R1, R3, R4$ in the proof of all lemmas and theorems given in this paper. Therefore, other healthiness conditions are ignored for convenience.

$$
\begin{array}{ll}
& \top_R \quad [\text{def}] \\
= & R1 \circ R3j \circ R4(true \vdash false) \quad [R3j^3] \\
= & R1 \circ R4((true \vdash \mathbb{I}) \lhd wait \rhd (true \vdash false)) \\
& \quad [\text{design-conditional}] \\
= & R1 \circ R4((true \lhd wait \rhd true) \vdash (\mathbb{I} \lhd wait \rhd false)) \\
& \quad [\text{propositional calculus}] \\
= & R1 \circ R4(true \wedge ok \Rightarrow ok' \wedge \mathbb{I} \wedge wait) \quad [\text{p. c.}] \\
= & R1 \circ R4(\neg ok \vee (wait \wedge ok' \wedge \mathbb{I})) \quad [R1\text{-}R4] \\
= & R1 \circ R4(\neg ok) \vee (wait \wedge ok' \wedge \mathbb{I})
\end{array}
$$

L1: $\top_R ; P = \top_R$

Proof:

$$
\begin{array}{ll}
& \top_R ; P \quad [\text{simplified } \top_R] \\
= & (R1 \circ R4(\neg ok) \vee (wait \wedge ok' \wedge \mathbb{I})) ; P \quad [\vee\text{-; distr}] \\
= & (R1 \circ R4(\neg ok) ; P) \vee ((wait \wedge ok' \wedge \mathbb{I}) ; P) \\
& \quad [R1\text{-}R4] \\
= & (\neg ok \wedge tr \leq tr' \wedge t \leq t' ; P) \vee ((wait \wedge ok' \wedge \mathbb{I}) ; P) \\
& \quad [\text{relational calculus}] \\
= & (\neg ok \wedge (tr \leq tr' \wedge t \leq t' ; P)) \vee ((wait \wedge ok' \wedge \mathbb{I}) ; P) \\
& \quad [\textit{Lemma } 1] \\
= & (\neg ok \wedge tr \leq tr' \wedge t \leq t') \vee ((wait \wedge ok' \wedge \mathbb{I}) ; P) \\
& \quad [\textit{lemma } 2] \\
= & R1 \circ R4(\neg ok) \vee (wait \wedge ok' \wedge \mathbb{I}) \quad [\top_R] \\
= & \top_R
\end{array}
$$

■

***Lemma** 1:* For a reactive process P [4],

$$(tr \leq tr') ; P = tr \leq tr'$$

[3]Woodcock [14] introduced $R3j$ to replace the original $R3$ in order to make a design behave like the design identity when waiting, and proved that it was equivalent to $R3$ when combined with $R1$.

[4]This lemmas has been proved in the tutorial [2].

***Lemma** 2:* For a reactive process P ,

$$(wait \wedge ok' \wedge \mathbb{I}) ; P = wait \wedge ok \wedge \mathbb{I}$$

Proof:

$$
\begin{array}{ll}
& (wait \wedge ok' \wedge \mathbb{I}) ; P \quad [\text{r. c.}] \\
= & wait \wedge ok \wedge (\mathbb{I} ; P) \quad [\mathbb{I}\text{-unit}] \\
= & wait \wedge ok \wedge P \quad [R3] \\
= & wait \wedge ok \wedge (\mathbb{I}_{rea} \lhd wait \rhd P) \quad [\text{p. c.}] \\
= & wait \wedge ok \wedge \mathbb{I}_{rea} \quad [\text{p. c.}] \\
= & wait \wedge ok' \wedge \mathbb{I}
\end{array}
$$

■

The definition of delay is as follows:

$$
\begin{array}{l}
WAIT\ d \mathrel{\widehat{=}} R(true \vdash W) \\
W = (tr' = tr \wedge v' = v \wedge ((wait' \wedge t' - t < d) \vee \\
\qquad (\neg\ wait' \wedge t' - t = d)))
\end{array}
$$

Theorem 1

$$
\begin{array}{l}
WAIT\ d ; \top_R = R(true \vdash tr' = tr \wedge v' = v \\
\qquad \wedge\ wait' \wedge t' - t < d)
\end{array}
$$

Proof:

$$
\begin{array}{ll}
& WAIT\ d ; \top_R \quad [\text{def-}WAIT] \\
= & R(true \vdash W) ; \top_R \quad [R3j, \text{ design-conditional}] \\
= & R1 \circ R4(true \vdash (\mathbb{I} \lhd wait \rhd W)) ; \top_R \quad [\text{def-}\top_R] \\
= & R1 \circ R4(true \vdash (\mathbb{I} \lhd wait \rhd W)) ; \quad [;\text{-}\vee \text{ distr}] \\
& (R1 \circ R4(\neg ok) \vee (wait \wedge ok' \wedge \mathbb{I})) \\
= & (R1 \circ R4(true \vdash \mathbb{I} \lhd wait \rhd W) ; R1 \circ R4(\neg ok)) \vee \\
& (R1 \circ R4(true \vdash \mathbb{I} \lhd wait \rhd W) ; (wait \wedge ok' \wedge \mathbb{I})) \\
& \quad [\text{r. c.}] \\
= & R1 \circ R4(\neg ok) \vee (R1 \circ R4(ok' \wedge \mathbb{I} \lhd wait \rhd W) ; \\
& (wait \wedge ok' \wedge \mathbb{I})) \quad [\vee\text{-; distr}] \\
= & R1 \circ R4(\neg ok) \vee \\
& R1 \circ R4(wait \wedge ok' \wedge \mathbb{I}) ; (wait \wedge ok' \wedge \mathbb{I}) \vee \\
& R1 \circ R4(ok' \wedge \neg\ wait \wedge tr' = tr \wedge v' = v \wedge wait' \wedge \\
& t' - t < d) ; (wait \wedge ok' \wedge \mathbb{I}) \vee \\
& R1 \circ R4(ok' \wedge \neg\ wait \wedge tr' = tr \wedge v' = v \wedge \neg\ wait' \wedge \\
& t' - t = d) ; (wait \wedge ok' \wedge \mathbb{I}) \quad [\text{p. c.}] \\
= & R1 \circ R4(\neg ok) \vee R1 \circ R4(wait \wedge ok' \wedge \mathbb{I}) \vee \quad [R3j] \\
& R1 \circ R4(ok' \wedge \neg\ wait \wedge tr' = tr \wedge v' = v \wedge \\
& \qquad wait' \wedge t' - t < d) \vee false \\
= & R(true \vdash tr' = tr \wedge v' = v \wedge wait' \wedge t' - t < d)
\end{array}
$$

■

The definitions of the external choice and a simple prefix are as follows:

$$P_1 \,\Box\, P_2 \mathrel{\widehat{=}} R\,((\neg\, {P_1}_f^f \wedge \neg\, {P_2}_f^f) \vdash (({P_1}_f^t \wedge {P_2}_f^t) \lhd tr' = tr \wedge wait' \rhd ({P_1}_f^t \vee {P_2}_f^t))$$

where $P_f^t = P[true, false/ok', wait]$.

$$a \to SKIP \mathrel{\widehat{=}} R \left(true \vdash \begin{array}{c} tr' = tr \wedge a \notin ref' \\ \lhd\, wait' \,\rhd \\ tr' = tr \frown \langle (t', a) \rangle \end{array} \quad \wedge\, v' = v \right)$$

***Lemma** 3:*

$$(a \to SKIP)_f^f = \top_{R_f^f} = \top_{R_f^t} = R1 \circ R4(\neg\, ok)$$

Proof:

$$\begin{array}{ll} & (a \to SKIP)_f^f \qquad [\text{def}] \\ =& R(true \vdash tr' = tr \wedge a \notin ref' \wedge wait' \wedge v' = v)_f^f \qquad [\text{R3j}] \\ =& R1 \circ R4(true \vdash \mathbb{I} \lhd wait \rhd tr' = tr \qquad [wait{=}\text{false}] \\ & \qquad \wedge\, a \notin ref' \wedge wait' \wedge v' = v)_f^f \\ =& R1 \circ R4(true \vdash tr' = tr \wedge a \notin ref' \wedge wait' \wedge v' = v)^f \\ =& R1 \circ R4(\neg\, ok \vee false) \qquad [\text{p. c.}] \\ =& R1 \circ R4(\neg\, ok) \end{array}$$

■

***Lemma** 4:*

$$\begin{array}{l} (a \to SKIP)_f^t \\ = R1 \circ R4 \left(true \vdash \begin{array}{c} tr' = tr \wedge a \notin ref' \\ \lhd\, wait' \,\rhd \\ tr' = tr \frown \langle (t', a) \rangle \end{array} \quad \wedge\, v' = v \right) \end{array}$$

***Lemma** 5:*

$$\top_{R_f^t} \wedge (a \to SKIP)_f^t = R1 \circ R4(\neg\, ok)$$

Theorem 2

$$\begin{array}{l} (a \to SKIP) \,\Box\, \top_R \\ = R(true \vdash \neg\, wait' \wedge tr = tr \frown \langle (t', a) \rangle \wedge v' = v) \end{array}$$

Proof:

$$\begin{array}{ll} & (a \to SKIP) \,\Box\, \top_R \qquad [\text{def-}\Box] \\ =& R(\neg\, \top_{R_f^f} \wedge \neg\, (a \to SKIP)_f^f \vdash \left(\begin{array}{c} \top_{R_f^t} \wedge (a \to SKIP)_f^t \\ \lhd\, tr' = tr \wedge wait' \,\rhd \\ \top_{R_f^t} \vee (a \to SKIP)_f^t \end{array} \right)) \qquad [\textit{Lemma } 3\text{-}5] \\ =& R(\neg\, R1 \circ R4(\neg\, ok) \vdash \left(\begin{array}{c} R1 \circ R4(\neg\, ok) \\ \lhd\, tr' = tr \wedge wait' \,\rhd \\ R1 \circ R4(\neg\, ok) \vee (a \to SKIP)_f^t \end{array} \right)) \qquad [\text{def-design}] \\ =& R(\neg\, R1 \circ R4(\neg\, ok) \wedge ok \Rightarrow ok' \wedge \left(\begin{array}{c} R1 \circ R4(\neg\, ok) \\ \lhd\, tr' = tr \wedge wait' \,\rhd \\ R1 \circ R4(\neg\, ok) \vee (a \to SKIP)_f^t \end{array} \right)) \qquad [\text{p. c.}] \\ =& R(R1 \circ R4(\neg\, ok) \vee \neg\, ok \vee (ok' \wedge R1 \circ R4(\neg\, ok) \\ & \wedge\, tr' = tr \wedge wait') \vee (ok' \wedge \neg\, (tr' = tr \wedge wait') \wedge \\ & (R1 \circ R4(\neg\, ok) \vee (a \to SKIP)_f^t))) \\ & \qquad [\text{absorption-}\vee \text{ and p.c.}] \\ =& R(R1 \circ R4(\neg\, ok) \vee \neg\, ok \qquad [\text{absorption-}\vee \text{ and p.c.}] \\ & \vee (ok' \wedge \neg\, (tr' = tr \wedge wait') \wedge R1 \circ R4(\neg\, ok)) \\ & \vee (ok' \wedge \neg\, (tr' = tr \wedge wait') \wedge (a \to SKIP)_f^t)) \\ =& R(\neg\, ok \vee (ok' \wedge \neg\, (tr' = tr \wedge wait') \wedge (a \to SKIP)_f^t)) \\ & \qquad [\textit{Lemma } 4] \\ =& R(\neg\, ok \vee (ok' \wedge \neg\, (tr' = tr \wedge wait') \wedge \left(\begin{array}{c} tr' = tr \wedge a \notin ref' \\ true \vdash \lhd\, wait' \,\rhd \quad \wedge\, v' = v \\ tr' = tr \frown \langle (t', a) \rangle \end{array} \right)) \\ & \qquad [\text{absorption-}\vee \text{ and p.c.}] \\ =& R(\neg\, ok \vee (ok' \wedge \neg\, (tr' = tr \wedge wait') \qquad [\text{p.c.}] \\ & \wedge\, (\neg\, wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v))) \\ =& R(\neg\, ok \vee (ok' \wedge (\neg\, wait' \wedge tr' = tr \frown \langle (t', a) \rangle \\ & \wedge\, v' = v))) \qquad [\text{def-}\vdash] \\ =& R(true \vdash \neg\, wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v) \end{array}$$

■

The reactive design semantics for parallel composition 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. Here, we will not give a detailed introduction to the semantics, and the interested reader is referred to the technical

report [12].

$$P_1 \underset{A}{\parallel} P_2 \mathrel{\widehat{=}}$$

$$R\left(\begin{array}{c}(\neg\ \exists 1.tr', 2.tr' \bullet (P_{1_f}^{f}; (1.tr' = tr)) \wedge (P_{2_f}; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \restriction A = 2.tr' - tr \restriction A) \\ \wedge \\ (\neg\ \exists 1.tr', 2.tr' \bullet (P_{1_f}; (1.tr' = tr)) \wedge (P_{2_f}^{f}; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \restriction A = 2.tr' - tr \restriction A) \\ \vdash \\ ((P_{1_f}^{t}; U1(out\alpha P_1)) \wedge (P_{2_f}^{t}; U2(out\alpha P_2)))_{+\{v,tr\}}\ ;\ M_{\parallel}(A)\end{array}\right)$$

***Lemma** 6:* For a reactive design process P,

$$R1 \circ R4(\neg\ ok) \wedge P = R1 \circ R4(\neg\ ok)$$

Proof:

$$\begin{array}{ll} & R1 \circ R4(\neg\ ok) \wedge P & [CSP1\text{-healthy}] \\ = & R1 \circ R4(\neg\ ok) \wedge CSP1(P) & [\text{def-}CSP1] \\ = & R1 \circ R4(\neg\ ok) \wedge (P \vee R1 \circ R4(\neg\ ok)) & [\text{absorption-}\wedge] \\ = & R1 \circ R4(\neg\ ok) & \end{array}$$

■

***Lemma** 7:* For a reactive design process P,

$$\left(\begin{array}{c}\exists 1.tr', 2.tr' \bullet (P_f^{f}; (1.tr' = tr)) \wedge (\top_{Rf}; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \restriction A = 2.tr' - tr \restriction A)\end{array}\right) = R1 \circ R4(\neg\ ok)$$

Proof:

$$\begin{array}{ll} & \left(\begin{array}{c}\exists 1.tr', 2.tr' \bullet (P_f^{f}; (1.tr' = tr)) \wedge (\top_{Rf}; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \restriction A = 2.tr' - tr \restriction A)\end{array}\right) & [\textit{Lemma}\ 3] \\ = & \exists 1.tr', 2.tr' \bullet (P_f^{f}; (1.tr' = tr)) \wedge & [R1 \text{ and p. c.}] \\ & (R1 \circ R4(\neg\ ok); (2.tr' = tr)) \wedge & \\ & 1.tr' - tr \restriction A = 2.tr' - tr \restriction A) & \\ = & (tr \leq tr' \wedge R4(P)_f^{f}; (tr' = tr)) \wedge & [\textit{Lemma}\ 6, \text{ r. c.}] \\ & (tr \leq tr' \wedge R4(\neg\ ok); (tr' = tr)) \wedge & \\ & tr' - tr \restriction A = tr' - tr \restriction A) & \\ = & R1 \circ R4(\neg\ ok) & \end{array}$$

■

***Lemma** 8:* For a reactive design process P,

$$\left(\begin{array}{c}\exists 1.tr', 2.tr' \bullet (P_f; (1.tr' = tr)) \wedge (\top_{Rf}^{f}; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \restriction A = 2.tr' - tr \restriction A)\end{array}\right) = R1 \circ R4(\neg\ ok)$$

Proof:

$$\begin{array}{ll} & \left(\begin{array}{c}\exists 1.tr', 2.tr' \bullet (P_f; (1.tr' = tr)) \wedge (\top_{Rf}^{f}; (2.tr' = tr)) \\ \wedge\ 1.tr' - tr \restriction A = 2.tr' - tr \restriction A)\end{array}\right) & [\textit{Lemma}\ 3] \\ = & \exists 1.tr', 2.tr' \bullet (P_f; (1.tr' = tr)) \wedge & [R1] \\ & (R1 \circ R4(\neg\ ok); (2.tr' = tr)) \wedge & \\ & 1.tr' - tr \restriction A = 2.tr' - tr \restriction A) & \\ = & (tr \leq tr' \wedge P_f; (tr' = tr)) \wedge & [\textit{Lemma}\ 6, \text{ r. c.}] \\ & (tr \leq tr' \wedge R4(\neg\ ok); (tr' = tr)) \wedge & \\ & tr' - tr \restriction A = tr' - tr \restriction A) & \\ = & R1 \circ R4(\neg\ ok) & \end{array}$$

■

***Lemma** 9:* For a reactive design process P,

$$((P_f^{t}; U1(out\alpha P)) \wedge (\top_{Rf}^{t}; U2(out\alpha \top_R)))_{+\{v,tr\}}\ ;\ M_{\parallel}(A) = R1 \circ R4(\neg\ ok)$$

Proof:

$$\begin{array}{ll} & ((P_f^{t}; U1(out\alpha P)) \wedge (\top_{Rf}^{t}; U2(out\alpha \top_R)))_{+\{v,tr\}}\ ;\ M_{\parallel}(A) & [\textit{Lemma}\ 3] \\ = & ((P_f^{t}; U1(out\alpha P)) \wedge & \\ & (R1 \circ R4(\neg\ ok); U2(out\alpha \top_R)))_{+\{v,tr\}}\ ;\ M_{\parallel}(A) & [\textit{Lemma}\ 1] \\ = & ((P_f^{t}; U1(out\alpha P)) \wedge R1 \circ R4(\neg\ ok))_{+\{v,tr\}}\ ;\ M_{\parallel}(A) & [\textit{Lemma}\ 6] \\ = & (R1 \circ R4(\neg\ ok))_{+\{v,tr\}}\ ;\ M_{\parallel}(A) & [\textit{Lemma}\ 1, \text{ p.c.}] \\ = & R1 \circ R4(\neg\ ok) & \end{array}$$

■

L4: $P \underset{A}{\parallel} \top_R = \top_R$

Proof:

$$\begin{array}{ll} & P \underset{A}{\parallel} \top_R & [\textit{Lemma}\ 7,8,9] \\ = & R((\neg\ R1 \circ R4(\neg\ ok) \wedge \neg\ R1 \circ R4(\neg\ ok)) & \\ & \quad \vdash R1 \circ R4(\neg\ ok)) & [\text{def-}\vdash] \\ = & R(R1 \circ R4(\neg\ ok) \vee \neg\ ok \vee (ok' \wedge R1 \circ R4(\neg\ ok))) & [\text{absorption-}\vee] \\ = & R(\neg\ ok) & \end{array}$$

■

Theorem 4

$$\begin{array}{ll} & (a \rightarrow SKIP)\ \Box\ SKIP & \\ = & R(true \vdash (\neg\ wait' \wedge tr' = tr \wedge v' = v \wedge t' = t) \vee & \\ & \quad (\neg\ wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v)) & \end{array}$$

Proof:

$$
\begin{aligned}
& (a \rightarrow SKIP) \,\Box\, SKIP && \text{[def-}\Box\text{]}\\
=& R(\neg\, (a \rightarrow SKIP)^f_f \wedge \neg\, SKIP^f_f \\
& \qquad \vdash \left(\begin{array}{c} (a \rightarrow SKIP)^t_f \wedge SKIP^t_f \\ \lhd\, tr' = tr \wedge wait' \,\rhd \\ (a \rightarrow SKIP)^t_f \vee SKIP^t_f \end{array} \right)) && \text{[}\textit{Lemma}\text{ 3]}\\
=& R(\neg\, R1 \circ R4(\neg\, ok) \\
& \qquad \vdash \left(\begin{array}{c} (a \rightarrow SKIP)^t_f \wedge SKIP^t_f \\ \lhd\, tr' = tr \wedge wait' \,\rhd \\ (a \rightarrow SKIP)^t_f \vee SKIP^t_f \end{array} \right)) && \text{[def-}\vdash\text{]}\\
=& R(R1 \circ R4(\neg\, ok) \vee \neg\, ok \vee ((a \rightarrow SKIP)^t_f \wedge SKIP^t_f \\
& \wedge\, tr' = tr \wedge wait') \vee (((a \rightarrow SKIP)^t_f \vee SKIP^t_f) \\
& \wedge \neg\, (tr' = tr \wedge wait'))) && \text{[absorption-}\vee\text{]}\\
=& R(\neg\, ok \vee ((a \rightarrow SKIP)^t_f \wedge SKIP^t_f \wedge tr' = tr \wedge wait') \\
& \vee (((a \rightarrow SKIP)^t_f \vee SKIP^t_f) \wedge \neg\, (tr' = tr \wedge wait'))) \\
& && \text{[def-SKIP]}\\
=& R(\neg\, ok \vee ((a \rightarrow SKIP)^t_f && \text{[}\textit{Lemma}\text{ 4, p. c.]}\\
& \qquad \wedge\, R1 \circ R4(true \vdash tr' = tr \wedge v' = v \wedge \neg\, wait' \\
& \qquad \wedge\, t' = t) \wedge tr' = tr \wedge wait') \\
& \vee (((a \rightarrow SKIP)^t_f \vee SKIP^t_f) \wedge \neg\, (tr' = tr \wedge wait'))) \\
=& R(\neg\, ok \vee R1 \circ R4(\neg\, ok) && \text{[p. c.]}\\
& \vee (((a \rightarrow SKIP)^t_f \vee SKIP^t_f) \wedge \neg\, (tr' = tr \wedge wait'))) \\
=& R(\neg\, ok \vee (ok' \wedge (\neg\, wait' \wedge tr' = tr \wedge v' = v \wedge t' = t) \\
& \vee (\neg\, wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v))) && \text{[def-}\vdash\text{]}\\
=& R(true \vdash (\neg\, wait' \wedge tr' = tr \wedge v' = v \wedge t' = t) \vee \\
& \qquad (\neg\, wait' \wedge tr' = tr \frown \langle (t', a) \rangle \wedge v' = v))
\end{aligned}
$$

■

REFERENCES

[1] A. Burns and I. Hayes. A timeband framework for modelling real-time systems. *Real-Time Systems Journal*, 45(1–2):106–142, June 2010.

[2] A. Cavalcanti and J. Woodcock. A tutorial introduction to CSP in Unifying Theories of Programming. In *Refinement Techniques in Software Engineering*, volume 3167 of *LNCS*, pages 220–268. Springer, 2006.

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

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

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

[6] M. Oliveira, A. Cavalcanti, and J. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, 21(1):3 – 32, 2007.

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

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

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

[10] A. Sherif and J. He. Towards a time model for *Circus*. In *ICFEM '02: Proceedings of the 4th International Conference on Formal Engineering Methods*, pages 613–624, London, UK, 2002. Springer-Verlag.

[11] J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1992.

[12] K. Wei, J. Woodcock, and A. Burns. Formalising the timebands model in timed *Circus*. Technical Report available at http://www-users.cs.york.ac.uk/~kun/, University of York, 2010.

[13] K. Wei, J. Woodcock, and A. Burns. A timed model of *Circus* with the reactive design miracle. In *8th International Conference on Software Engineering and Formal Methods (SEFM)*, pages 315–319, Pisa, Italy, September 2010. IEEE Computer Society.

[14] J. Woodcock. The miracle of reactive programming. In *Unifying Theories of Programming 2008: 2nd International Symposium*, Dublin, Ireland, 2008. Springer-Verlag.

[15] J. Woodcock and A. Cavalcanti. The semantics of *Circus*. In *ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 184–203, London, UK, 2002. Springer-Verlag.

[16] J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.