

Embedding the Timed *Circus* in PVS

Kun Wei, Jim Woodcock and Alan Burns

Department of Computer Science
University of York, UK, YO10 5DD
{kun,jim,burns}@cs.york.ac.uk

Abstract. We present an embedding of a timed model of *Circus* with the reactive design miracle in the PVS theorem prover. Our work, originally built on Dutertre’s embedding of the traces model of CSP, provides a platform for the formal verification of various definitions and refinement laws as well as the correctness of systems specified by this model. The new timed model is given a denotational semantics based on *Unifying Theories of Programming* (UTP), describing each process as a reactive design and using the reactive design miracle to model some brand-new features of a system, which cannot be properly represented by the failures-divergences model of CSP.

1 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. *Circus* is a combination of CSP [13, 21, 23] and Z [24, 31], which can define both data and behavioural aspects of a system; that is, it can describe changes of states, and in the meantime naturally deal with concurrency. A well-defined syntax and a sound semantics [29, 30] have been given in *Circus*, based on unifying theories of programming (UTP) [14].

In our previous work, we propose a new timed model of *Circus* [28] by introducing the reactive design miracle to use a complete lattice with respect to the implication ordering. However our timed model of *Circus* is perhaps closer to timed CSP but preserves the ability to handle data by allowing processes to obtain a set of local variables. Compared with timed CSP, our timed *Circus* defines each process as a reactive design and retains all operators of CSP, extending with new featured operators such as guarded processes and deadline. Consequently, we are able to represent some brand-new features that have never been achieved in timed CSP.

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, *false* is a miracle (the top element) in the complete lattice of relations because it can never give rise to any observation although it can satisfy every specification. In [28] we have proved a number of algebraic laws, especially involving the miracle

by hand. However, constructing proofs of correctness by hand is arduous and error-prone. Therefore we embed the UTP-style denotational semantics of the timed model of *Circus* in the PVS theorem prover, not only for checking the soundness and consistence of the semantics, but also for increasing our confidence in correctly analysing some miraculous processes. Furthermore, such an embedding is very useful and convenient for reasoning about the properties of a complex system, and other researchers can also extend it easily to cover a wider variety of systems.

PVS [19, 7, 8], the Prototype Verification System, is an interactive theorem prover based on a form of higher-order logic. It provides an environment for constructing precise specifications, and for efficient mechanized verification. Although it is similar in many ways to other theorem provers such as Isabelle/HOL [20] and IMPS [10], it supports a richer type system, and checks semantic consistency for a PVS specification. PVS is also a natural choice for this work because of authors' previous work in [26, 27] where the authors embed the denotational semantics of the stable failures model of CSP in PVS and then apply their proof strategies to model and verify various properties of security protocols such as fairness.

There has been tool support provided for formal verification of *Circus* specifications. For example, Freitas et al. [12] develop an efficient and distinctive framework to implement model checking on *Circus* programs. They provide an operational semantics in order to represent *Circus* programs as automata, and then implement it in Java. The model checking architecture is quite complex and a number of auxiliary tools are integrated such as the Z/Eve [22] theorem prover and JCSP [11] to reduce errors during various translations. Moreover, to deal with industrial scale systems in *Circus*, Cavalcanti et al. [6, 17] propose refinement calculus to assist in constructing proofs. All refinement laws and strategies are defined and integrated in the ProofPower-Z theorem prover [1]. The application of the refinement calculus is boosted further by the new UTP semantics in [18] where each process is described as a reactive design. Apparently our timed *Circus*, based on the work in [18], takes advantage of the similar semantics to introduce the reactive design miracle and an additional variable, time.

The new timed model, as a variant of *Circus*, is supposed to adopt the same approach by means of integrating the semantics in ProofPower-Z, because Oliveira et al. [16] have done a deep embedding of part of UTP in the theorem prover containing a lot of theories such as relations, designs, reactive processes and so on. However, we determine to embed the denotational semantics in PVS, for we have completely abandoned the involvement of the Z language that plays an important role in original *Circus*. As a CSP-like language, our model is able to exploit most matured CSP refinement laws, which will be very convenient for verifying complex systems in the future. The embedding, built on the classical set theories, uses flexible expressions and abundant built-in theories in PVS and combines defined theories of CSP in previous work [9, 26] to make the mechanisation of the semantics straightforward to some extent.

Because the timed model is next of kin to CSP, we concentrate on the related work that encodes the denotational semantics of various CSP models to make it serve for their different purposes. For example, in the early stage Camilleri [4] has shown how a theorem prover can provide a natural framework for mechanising Hoare’s CSP [13]. Dutertre and Schneider [9] mechanise the traces mode of Roscoe’s CSP [21] in PVS to verify safety properties of security protocols. Tej and Woff [25] provide a platform of encoding the failures/divergences model of CSP in Isabelle/HOL. Isobe and Roggenbach [15] propose a new tool called CSP-Prover that embeds the stable failures model of CSP. Brooke [2] uses timed CSP, PVS and FDR to construct tool-supported proofs to verify properties of systems on an industrial scale.

The remainder of the paper is organized as follows. We will present a brief introduction to the notation of the timed model of *Circus* and its UTP semantics. We then show how to represent the semantics in PVS including embedding the fixed point theory for defining recursive processes and proving a number of algebraic laws. Finally, we conclude and discuss future work.

2 Timed *Circus* with the reactive design miracle

In UTP, Hoare and He use the alphabetised relational calculus to give a denotational semantics which can explain a wide variety of programming paradigms. A relation P is a predicate with an alphabet αP , composed of *undashed* variables (a, b, \dots) and *dashed* variables (a', x', \dots). 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 $in\alpha P$.

2.1 Reactive Designs

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'* that it has terminated. If precondition P and postcondition Q are predicates not containing *ok* and *ok'*, a design with P and Q , written $P \vdash Q$, is defined as follows:

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

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

Healthiness conditions of a theory in UTP are a collection of some fundamental laws that must be satisfied by relations of the theory; in other words, healthiness conditions are used to characterise a set of relations that belong to the theory. They are usually expressed by means of an idempotent function ϕ . For example, a relation P , satisfying $\phi(P) = P$, is called *healthy*.

There are four healthiness conditions identified by Hoare and He in UTP. Here we discuss two of them only, which are also used in the PVS embedding. A

relation P is $H1$ healthy if and only if $H1(P) = (ok \Rightarrow P)$, which means observations can only be made after the program has started. The second healthiness condition is $[P[false/ok'] \Rightarrow P[true/ok']]$ where square brackets denote universal quantification over all variables in the alphabet. $H2$ states that if P is satisfied when ok is false, it is also satisfied when ok is true. Thus, a design is a relation that is $H1$ and $H2$ healthy.

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 $wait'$ is true, then the process is in an intermediate state. If $wait$ 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 , tr , 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.
- tr specifies the trace of timed events in which a process has engaged until it starts, and tr' records all timed events that have occurred so far, up to the end of an observation. A timed event is a pair drawn from $\mathbb{R}^+ \times \Sigma$, consisting of a time and an event.
- ref records the set of events that could be refused in the last observation; ref' contains the set of events that could be refused in the next observation.
- v represents the initial values of a process's variables, and v' records the final values until the next observation.

There are three healthiness conditions that reactive processes must satisfy. If the relation P describes a reactive process behaviour, $R1$ states that it never changes history, or the trace is always extending.

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

The second healthiness condition, $R2$, states that the undashed variable tr has no influence on the behaviour of the process, and therefore P is not changed if tr is the empty sequence.

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

where $tr' - tr$ represents the trace of events that has occurred since the last observation.

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

$$R3(P) = \Pi_{rea} \triangleleft wait \triangleright P$$

where the reactive identity, \mathbb{I}_{rea} ,¹ is defined as follows:

$$\begin{aligned} \mathbb{I}_{rea} \hat{=} & (\neg ok \wedge tr \leq tr' \wedge t \leq t') \vee \\ & (ok' \wedge tr' = tr \wedge ref' = ref \wedge v' = v \wedge wait' = wait) \end{aligned}$$

which states that if ok is false, its predecessor diverges and then the only guarantee is that tr and t are extending; if ok' is true, it keeps states unchanged except that time may 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, an extra healthiness condition $R4$ must also be satisfied: $R4(P) = P \wedge t \leq t'$ which states time always moves forward. As a result, P is a timed reactive process if and only if it is a fixed point of $R \hat{=} R1 \circ R2 \circ R3 \circ R4$. In practice, $R2$ is always the healthiness condition that we are not interested in. Therefore, we usually remove it from our embedding of the semantics.

Note that the variable ok is not included in the alphabet of reactive processes. Our timed model of *Circus* is characterised as relations resulting from applying R to designs. For a more detailed introduction to the theory of reactive designs, the reader is referred to the tutorial [5].

2.2 Process Semantics

In this time model of *Circus*, we preserve all common operators of timed CSP, introduce two operators, assignment and guarded command, from *Circus*, and provide a new operator, deadline. The reactive design semantics based on UTP has been firstly proposed for CSP processes in [5], and the similar semantics for *Circus* in [18]. Here We rewrite the semantics on account of the involvement of time.

Primitive processes The reactive design miracle \top_R is defined in terms of the design miracle made R -healthy: $\top_R \hat{=} R(true \vdash false)$.

The reactive design abort \perp_R or *CHAOS* is the bottom element which can behave like anything: $CHAOS \hat{=} R(false \vdash true)$.

The process *STOP* is the deadlocked process which can perform nothing or refuse anything, allowing time to elapse:

$$STOP \hat{=} R(true \vdash tr' = tr \wedge wait' \wedge t' = t)$$

The process *SKIP* terminates immediately without changing the trace and process variables, defined as follows:

$$SKIP \hat{=} R(true \vdash tr' = tr \wedge v' = v \wedge \neg wait')$$

¹ In fact, this is the timed reactive identity because the observational variable t has been considered in the definition.

Sequential composition The definition of sequential composition, $P ; Q$, is as same as the one in alphabetised relational calculus, which denotes P is executed firstly and then Q when P terminates, and meanwhile the final state of P is passed on as the initial state of Q . It is defined as $P ; Q \hat{=} \exists x_0 \bullet P[x_0/x'] \wedge Q[x_0/x]$ where x is a list of variables including all variables used in the model.

Prefix The prefix process in UTP, denoting the process that behaves like P after performing the event a , can be represented in terms of a simple prefix and *SKIP*: $a \rightarrow P \hat{=} a \rightarrow \text{SKIP} ; P$, and then the simple prefix can be expressed as follows:

$$a \rightarrow \text{SKIP} \hat{=} R \left(\begin{array}{c} tr' = tr \wedge a \notin \sigma(ref') \\ true \vdash \quad \triangleleft wait' \triangleright \quad \wedge v' = v \\ tr' = tr \wedge \langle (t', a) \rangle \end{array} \right)$$

Assignment 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.

$$x :=_A e \hat{=} R (true \vdash tr' = tr \wedge \neg wait' \wedge t' = t \\ \wedge x' = e \wedge y' = y \wedge \dots \wedge z' = z)$$

where the set A is defined as $A = \{x, y, \dots, z, x', y', \dots, z'\}$ 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:

$$g \& P \hat{=} R((g \Rightarrow \neg P_f^f) \vdash ((g \wedge P_f^t) \vee (\neg g \wedge tr' = tr \wedge wait')))$$

where P_b^a represents $P[a/ok'][b/wait]$, and the abbreviation follows the definitions throughout this paper. For example, P_f^f denotes that P diverges while not waiting for its predecessor to terminate.

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

$$P \square Q \hat{=} R((\neg P_f^f \wedge \neg Q_f^f) \vdash (P_f^t \wedge Q_f^t \triangleleft tr' = tr \wedge wait' \triangleright P_f^t \vee Q_f^t))$$

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

Internal choice The internal choice $P \sqcap Q$ can behave either like P or like Q , but it is out of control of its environment. Therefore, it can be simply defined as $P \sqcap Q \hat{=} P \vee Q$.

Parallel composition The process $P \parallel_A Q$ is the process where all events in the set A must be synchronised, and events outside A can perform independently. The parallel process can terminate only if both P and Q terminate, and it becomes divergent after either one of P and Q does so.

The definition of parallel composition in reactive-design style is the most complex one, in which its precondition describes the behaviour of the process when it diverges, and its postcondition represents the parallel-by-merge semantics. The idea of the merging process is to make processes become disjoint processes by labelling variables of their alphabets so that each process can execute independently, and to merge the variables to produce the real final values at the end of observation. We do not give the detailed illustration of the semantics, and the reader is referred to [28, 18].

In our embedding we do not use the reactive-design semantics of parallel composition since it is rather awkward to be integrated in PVS. In fact, we adopt the original definition of parallel composition in UTP and modify it to satisfy our requirements. Its definition is given as follows:

$$\begin{aligned}
P \parallel_A Q \hat{=} & R(ok' = (P.ok' \wedge Q.ok') \wedge wait' = (P.wait' \vee Q.wait') \wedge \\
& tr' - tr = (P.tr' - P.tr) \parallel_A (Q.tr' - Q.tr) \wedge \\
& ref' = (P.ref' \cap Q.ref') \cup ((P.ref' \cup Q.ref') \cap A) \wedge \\
& v' = M(v, P.v', Q.v') \wedge t' = \max(P.t', Q.t'); SKIP
\end{aligned}$$

where M is a function merging the local variables of P and Q and the function \max just returns the maximal time.

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 Σ is a universal set of events, and then the hiding is defined as follows:

$$\begin{aligned}
P \setminus A \\
\hat{=} & R(\exists s \bullet P[s, (ref \cup A)/tr', ref'] \wedge (tr' - tr) = (s - tr) \upharpoonright (\Sigma - A)); SKIP
\end{aligned}$$

The hiding may introduce divergence; therefore, the process $SKIP$ is used to capture the observation of possible divergences.

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

$$WAIT\ d \hat{=} R(true \vdash tr' = tr \wedge v' = v \wedge (wait' \triangleleft t' - t < d \triangleright \neg wait'))$$

The trace tr and the local variable v always keep unchanged.

Timeout The time-sensitive choice $P \triangleright_d Q$ resolves the choice in favour of P if P is able to execute observable events by time d , or resolves the choice in favour of Q . Such an operator can be defined in terms of the delay process, sequential composition and external choice as follows:

$$P \triangleright_d Q \hat{=} P \square (\text{WAIT } d; Q)$$

Deadline Compared with the above timeout operator, the deadline operator claims that P *must* perform observable events within d units, defined in terms of the reactive design miracle:

$$P \blacktriangleright d \hat{=} P \triangleright_d \top_R$$

This is really a very strong requirement in which there is no alternative to meet the deadline; that is, within d time units, either P performs observable events or P terminates with doing nothing, otherwise the process will never start.

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 miracle \top_R and the bottom 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 $X = F(X)$.

For the detailed introduction to the semantics of the timed model, the reader is referred to [28].

3 Embedding semantics in PVS

Relations are predicates whose alphabet consists of undashed and dashed variables, which represent the initial observation and the intermediate or final observation respectively. In PVS, we suppose that all relations are homogeneous; that is, for each undashed variable, there is always a value for its corresponding dashed variable. Hence, a relation is simply treated as a set of combinations of all variables.

With regard to all variables used in the model, we define the following record expression in PVS.

```
AP:TYPE = [# ok:bool, ok1:bool, wait: bool, wait1:bool,
           tr:trace[E], tr1:trace[E], ref:set[E], ref1:set[E],
           v:Variable, v1:Variable, t:real, t1:real #]
```

In the record type AP, the variables with a number denote dashed variables. For example, `tr1`, a record accessor, denote tr' and its type is the trace type `trace[E]`. The type of traces has been defined in another theory and `E` is a parameter of the trace type denoting the type of elements in a trace. Here we just leave `E` as an undefined type that can be redefined whenever instantiating

the theory. In addition, v and $v1$ represent the local variables of a process, and t and $t1$ denote the start and end point respectively of the time interval.

Meanwhile, in order to make generic local variables, we define a type **Variable** as a function:

```
Variable: TYPE = [ Name -> real ]
```

where **Name** is an undefined type to represent a collection of names of local variables, and every name is mapped to a real-valued number. Thus, a relation is simply defined as a set in which the type of elements is **AP**:

```
Relation: TYPE = set[AP]
```

Recall the definition of a design in Section 2.1. The top element of designs in the implication ordering is $false \vdash true$ or $\neg ok$. If ok is false, the process has not started yet, and then no observation can be found. However, we interpret it as a relation in which ok is false and the values of other variables are arbitrary.

```
D_top: Relation = { pr:AP | pr.ok = false }
```

This interpretation immediately gives rise to a fact that the healthiness condition $H1$ is replaced by the requirement that D_top is a subset of any design. $H2$ is also imposed to the definition as follows:

```
Design: TYPE = { P:Relation | subset?(D_top,P) AND
  FORALL (pr1:AP): P(pr1) AND pr1.ok1=false IMPLIES
  EXISTS (pr2:AP): P(pr2) AND pr2.ok1=true AND equal?(pr1,pr2) }
```

where the boolean function, $equal?(pr1,pr2)$, is true only if $pr1$ equals $pr2$ except for the value of ok' .

3.1 Healthiness conditions

Reactive processes are those relations that satisfy the healthiness conditions: $R1$ – $R4$. However, $R2$ is always the healthiness condition that we are not interested in. Therefore, we define $R1$, $R3$ and $R4$ only in the PVS embedding.

$R1$ guarantees traces are always extended and $R4$ has the similar guarantee for time, defined as follows:

```
R1(P): Relation = { pr | P(pr) AND pr.tr <= pr.tr1 }
R4(P): Relation = { pr | P(pr) AND pr.t <= pr.t1 }
```

$R1$ is defined as a function, taking an arbitrary relation P and returning a relation where the initial values of traces are never greater than the final values, and so is $R4$.

The definition of $R3$ is not as simple as that of $R1$. It actually modifies relations, defined as follows:

```
R3(P):Relation = { pr | IF pr.wait=true THEN RI(pr) ELSE P(pr) ENDIF }
```

$R3$ means that for each element pr in a relation P , if $wait$ is true, pr can be any element of the reactive identity RI ; otherwise it keeps unchanged. The definition of RI as given in Section 2.1 is described as follows:

```
RI:Relation = {pr | (pr'ok=false AND pr'tr<=pr'tr1 AND pr't<=pr't1) OR
                  (pr'ok1=true AND pr'wait1=pr'wait AND pr'tr1=pr'tr
                   AND pr'ref1=pr'ref AND pr'v1=pr'v AND pr't1=pr't) }
```

Finally, processes in our time model of *Circus* are defined by applying R on designs.

```
R(P):Relation = R3(R1(R4(P)))
process: TYPE = { D:Design | R(D) = D }
```

Before getting in the PVS definitions of various operators of the timed *Circus*, we first define sequence composition that is not described as a reactive design.

```
** (P,Q):Relation = {pr | EXISTS p,q: P(p) AND Q(q) AND assign_middle(p,q)
                    AND assign_in(pr,p) AND assign_out(pr,q) }
```

which merges the elements of P and Q ; for example, the record expression pr takes the values of undashed variables from p and the values of dashed variable from q by $assign_in$ and $assign_out$, as long as the output of p equals the input of q in terms of the boolean function $assign_middle$.

3.2 Basic processes and operators

The PVS definitions for primitive processes are very straightforward, described as follows:

```
RM: process = R({ pr | pr'ok = false })
CHAOS: process = R({ pr | true })

STOP:process = R({ pr | pr'ok=false OR
                  (pr'ok=true AND pr'ok1=true AND pr'tr1=pr'tr
                   AND pr'wait1=true) })
SKIP:process = R({ pr | pr'ok=false OR
                  (pr'ok=true AND pr'ok1=true AND pr'tr1=pr'tr
                   AND pr'wait1=false AND pr't1=pr't AND pr'v1=pr'v) });
```

To formalise the prefix in PVS, we define the simple prefix and then concatenate it with *SKIP*.

```
P,Q: VAR process
SKIP(a):process=R({ pr | pr'ok=false OR
                  (pr'ok=true AND pr'ok1=true AND pr'v1=pr'v AND
                   ( (pr'wait1=true AND pr'tr1=pr'tr AND NOT pr'ref1(a)) OR
                     (pr'wait1=false AND pr'tr1= add(pr'tr,(pr't1,a)))) ) })

>>(a,P): process = (SKIP(a) ** P)
```

where the function `add` simply appends a timed event to the end of a trace.

To represent the form of P_f^f , we define the following function:

```

sup: VAR bool % to denote ok'
sub: VAR bool % to denote wait

OW(P,sup,sub):process={ pr | P(pr) AND
                        (IF sup=true THEN pr'ok=true AND pr'ok1=true
                         ELSE pr'ok1=false ENDIF)
                        AND pr'wait=sub }

```

Note that there is a little bit complex when assigning the value to ok' , which results from different views of no observation in UTP and PVS. If ok' is true, we can deduce that ok must be true too in UTP; however in this embedding we have to assign *true* to ok explicitly, because in previous embedding ok' could be either true or false when ok is false.

Now it is convenient to define a guarded process as follows:

```

&(g,P):process =R({ pr | pr'ok=false OR
                    (g=true AND OW(P,false,false)(pr)) OR
                    (g=true AND OW(P,true,false)(pr)) OR
                    (pr'ok=true AND pr'ok1=true AND g=false
                     AND pr'tr1=pr'tr AND pr'wait1=true) } );

```

For external choice of two processes, for example P and Q , if neither of them terminates and no event occurs, it behaves like the conjunction of P and Q , otherwise it behaves like the disjunction.

```

/(P,Q):process=R({pr | pr'ok=false OR
                  OW(P,false,false)(pr) OR
                  OW(Q,false,false)(pr) OR
                  IF (pr'wait1=true AND pr'tr1=pr'tr)
                  THEN (OW(P,true,false)(pr) AND OW(Q,true,false)(pr))
                  ELSE (OW(P,true,false)(pr) OR OW(Q,true,false)(pr)) ENDIF});

```

The internal choice $P \sqcap Q$ can behave either like P or like Q , defined as follows:

```

\/(P,Q): process = union(P,Q)

```

Other than taking the reactive-design semantics, we adopt the ordinary UTP semantics for parallel composition, since it is more convenient to be represented in PVS.

```

A: VAR set[E]
pr,pr1,pr2: VAR AP

```

```

Par(A)(P,Q):process = R({pr | pr'ok=false OR
                          EXISTS pr1,pr2: P(pr1) AND Q(pr2) AND
                          assign_in(pr,pr1) AND assign_in(pr,pr2) AND
                          pr'ok1=(pr1'ok1 AND pr2'ok1) AND
                          pr'wait1=(pr1'wait1 OR pr2'wait1) AND

```

```

pod(A)(sub(pr1'tr1,pr1'tr),sub(pr2'tr1,pr2'tr),sub(pr'tr1,pr'tr)) AND
pr'ref1=union(inter(pr1'ref1,pr2'ref1), inter(union(pr1'ref1,pr2'ref1),A))
AND pr'v1=merge(pr'v,pr1'v1,pr2'v1) AND pr't1=max(pr1't1,pr2't1} )

```

where $prod(A)(t1, t2, t)$, or $t = t1 \parallel^A t2$, is one of key components in the definition, which is true if the trace t is the combination of $t1$ and $t2$ synchronised on the set A . Another issue is how to deal with the final observation of local variables in parallel composition, or in other words, how to handle shared variables? No *Circus* process in parallel can know the change of variables of other participants, unless it is notified by communicating events. Of course, the worst case is that every process keeps the change in secret and finally all of them are going to update the variables for the whole process. To avoid this dangerous situation, the compulsory requirement for each process is usually that their local variables are disjoint. However such a requirement gives rise to enormous *type-correctness conditions* (TCCs) that PVS has to discharge. Therefore, we use a different strategy to reduce TTCs by allowing processes to obtain shared variables and simply taking the maximal value if a variable is changed in more than one process. Such an execution is fulfilled by the function `merge`. In addition, the function `sub(s, t)` simply returns the result of moving an initial copy of t from s .

The definition of the hiding operator is straightforward, defined as follows:

```

/(S,A):process = R({pr| pr'ok=false OR
                    EXISTS p: P(p) AND assign_in(pr,p) AND
                    pr'ok1=p'ok1 AND pr'wait1=p'wait1 AND
                    pr'tr1=append(pr'tr,leak(sub(p'tr1,p'tr),A)) AND
                    pr'ref1=diff(p'ref1,A) AND pr'v1=p'v1 AND pr't1=p't1} )

```

where the function $leak(t, A)$ returns a trace from t in which all events included in the set A have been removed, and $diff(A, B)$ simply keeps the elements of A that are not in B .

The timed operators, *Delay*, *Timeout* and *deadline* are defined just as their semantics represented in Section 2.2:

```

WAIT(d):process=R({pr| pr'ok=false OR
                    (pr'ok=true AND pr'ok1=true AND pr'tr1=pr'tr AND pr'v1=pr'v AND
                    IF pr't1-pr't < d THEN pr'wait1=true
                    ELSE pr'wait1=false ENDIF} )

```

```

Timeout(P,Q,d): process = P /\ (WAIT(d)**Q)
Deadline(P,d): process = Timeout(P,RM,d)

```

3.3 Algebraic laws and refinement

We have also mechanically proved a number of algebraic laws that are essential in the verification of properties of processes. Additionally, proving such algebraic laws may verify and underpin the consistency of the semantics of our timed *Circus*, and help us check the translation of the semantics from the paper to the PVS embedding. For detailed general CSP algebraic laws, the reader is referred to [26]. Here we illustrate some unique laws in our timed model.

The combination of the reactive design miracle and ordinary operators gives rise to many miraculous processes that break some primitive assumptions of the standard model of CSP. For example, the process $a \rightarrow \top_R$ exhibits an extraordinary behaviour:

```
prefix_miracle: LEMMA a>>RM = R({p| p'ok=false OR
    (p'ok1=true AND p'tr1=p'tr AND NOT p'ref1(a) AND
    p'wait1=true AND p'v1=p'v )})
```

which states that although the event a is not in the refusal set, the process is always waiting for the interaction with its environment but never performs it. This obviously violates one of important axioms of CSP.

Another strange process is the external choice of the simple prefix and the miracle, $a \rightarrow SKIP \sqcap \top_R$, defined in PVS as follows:

```
extchoice_miracle: LEMMA
(SKIP(a) /\ RM) = R({p|(p'ok=false) OR
    (p'ok1=true AND p'wait1=false AND
    p'tr1=add(p'tr,(p't1,a)) AND p'v1=p'v )})
```

which states, if its predecessor successfully terminates, it may perform the event a and then terminate. However there is no state between the start point of the process and the time point when a is executed, because *wait'* is always false. Therefore, for example, if a is a tick of a clock and continuous time is used in the system, this process surprisingly translates continuous time to discrete time when the occurrence of a is periodic. There are a lots of strange processes on account of the involvement of the reactive design miracle, nevertheless the mechanisation of the semantics in PVS increases our confidence in analysing their behaviours correctly.

In UTP, if we suppose a program P with $\alpha P = \{x, x'\}$, then the *universal closure* of P is simply $\forall x, x' \bullet P$, or $[P]$. Let S be a specification, composed as a collection of various requirements placed on the behaviour of the program. The correctness of the program P is defined as follows:

$$S \sqsubseteq P \text{ iff } [P \Rightarrow S]$$

which denotes that S is refined by P , or no observation of P could ever violate the specification S . In PVS, we interpret the correctness as a subset relation.

```
|>(P,S): bool = subset?(P,S)
```

We choose the subset order rather than the refinement order for the purpose of proving properties of recursively defined processes with ease. We will detailed discuss it in next section.

3.4 Recursive processes and fixed points

In UTP, Hoare and He use weakest fixed points to define recursive processes. Suppose that a function F is *monotonic* if $P \sqsubseteq Q \Rightarrow F(P) \sqsubseteq F(Q)$, then the weakest fixed point of F , usually written as μF , is simply the greatest lower bound of all the fixed points of F , defined as follows:

```

SX: VAR set[process[E,Name]]
glb(SX): process[E,Name]= { pr | EXISTS (X:(SX)): X(pr) }

F: VAR [process[E,Name]->process[E,Name]]
monotonic?(F):bool = FORALL X,Y: X |> Y IMPLIES F(X) |> F(Y)

G: VAR (monotonic?)
mu(G): process[E,Name] = glb({ X | X |> G(X) })

```

where $\text{glb}(SX)$ denotes the greatest lower bound, $\prod SX$; E and $Name$ are the parameters for the process type to denote events and names of local variables respectively.

To prove that μF is itself a fixed point, we first formalise two laws of the greatest lower bound as follows:

```

glb_is_bound: LEMMA FORALL (X:(SX)): X |> glb(SX)
glb_is_sup: LEMMA (FORALL (X:(SX)): X|>Y ) IMPLIES glb(SX) |> Y

```

which actually are the laws, **L1A** and **L1B**, listed in Chapter 2 of the UTP book [14]. Thereupon, the following laws about the least fixed point can be easily proved.

```

closure_mu: LEMMA mu(G) |> G(mu(G))
smallest_closed: LEMMA X |> G(X) IMPLIES X |> mu(G)
fixed_point: LEMMA G(mu(G)) = mu(G)

```

Furthermore, we have also mechanically proved a general fixed point induction theorem, which is crucial in analysing refinement of recursive processes:

```

fix_induction: THOEREM RM |> S AND (FORALL X: X|>S IMPLIES G(X)|>S)
                IMPLIES mu(G) |> S

```

For example, for proving a recursive process P deadlock-free, we may construct a deadlock-free specification S , then check whether P refines S , or P is a subset of S . Firstly, we prove S contains the reactive design miracle which is the bottom element with respect to the subset order; then we unwind P as its definition to verify whether it is still included by S ; if it is, we then finish the proof.

4 Discussion and conclusion

We provide a shallow embedding of the reactive design semantics of the timed *Circus* in the PVS theorem prover. Our timed model of *Circus* introduces the reactive design miracle to use a complete lattice with respect to the implication ordering. The miracle is highly unexplored in UTP, as it can never be implemented in engineering practice. However, it has been proved extremely useful as a mathematical abstraction to specify and reason about properties of a system. For example, using the reactive design miracle, we have defined the strict deadline operator, which can specify that events must occur within certain time units. The involvement of the reactive design miracle also gives rise to a number of very strange processes that violate many fundamental assumptions of the standard CSP model. The mechanisation of the semantics in PVS can help us increase our confidence in analysing their extraordinary behaviours correctly, and also help us understand the timed model better.

Compared with the deep embedding of UTP in Proofpower-Z by Oliveira et al. [16], our embedding is easier and more convenient to use for analysing this timed model of *Circus*. We have proved a lot of algebraic laws in PVS, which can greatly reduce the efforts on constructing proofs of properties of a complex system. However proving those laws is non-trivial and time-consuming. This work² takes authors nearly one month to finish the proof.

There are a number of directions that future work can take. We will first model and verify a simple system to demonstrate the power of the timed model. We may also explore the feasibility of embedding the semantics of UTP in PVS for supporting the development of other languages whose semantics is based on UTP. Another direction in developing this model is to try to find an approach to formalise the timebands model [3]. For example, using the reactive miracle, we are able to address the difference between two events occur simultaneously and the ones at same time, which is one of peculiar features of the time band model. Obviously, the mechanisation of the timed *Circus* will help us check the correctness and consistency of the potential new model for the timebands model.

Acknowledgements We would like to thank Ana Cavalcanti, Leo Freitas, Andrew Butterfiel 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

1. R. D. Arthan. On formal specification of a proof tool. In *VDM '91 - Formal Software Development, 4th International Symposium of VDM Europe, Noordwijk-erhout, The Netherlands, October 21-25, 1991, Proceedings, Volume 1: Conference Contributions*.
2. P. Brooke. *A Timed Semantics for a Hierarchical Design Notation*. PhD thesis, University of York, 1999.
3. A. Burns, I. Hayes, G. Baxter, and C. Fidge. Modelling temporal behaviour in complex socio-technical systems. Technical Report YCS 390, University of York, 2005.
4. A. J. Camilleri. Higher order logic mechanization of the CSP failure-divergence semantics. Technical report, HP Lab Bristol, 1990.
5. 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.
6. A. L. C. Cavalcanti, A. C. A. Sampaio, and J. C. P. Woodcock. A Refinement Strategy for *Circus*. *Formal Aspects of Computing*, 15(2 - 3):146 — 181, 2003.
7. J. Crow, S. Owre, J. Rushby, and N. Shankar. A tutorial introduction to PVS. In *Workshop on Industrial-Strength Formal Specification Techniques*, Boca Raton, Florida, Apr. 1995.
8. J. Crow, S. Owre, J. Rushby, and N. Shankar. *PVS Prover Guide, PVS Language Reference, PVS System Guide*. SRI International, 2001.

² All PVS source code of the model can be downloaded from the first author's personal webpage.

9. B. Dutertre and S. A. Schneider. Embedding CSP in PVS: an application to authentication protocols. In E. Gunter and A. Felty, editors, *Theorem Proving in Higher-Order Logics: 10th International Conference, TPHOLs '97*, volume 1275. Springer-Verlag, 1997.
10. W. M. Farmer, J. D. Guttman, and J. F. Thayer. IMPS: An Interactive Mathematical Proof System. *Journal of Automated Reasoning*, 11:213–218, 1993.
11. A. F. Freitas and A. L. C. Cavalcanti. Automatic Translation from *Circus* to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors, *FM 2006: Formal Methods*, volume 4085 of *Lecture Notes in Computer Science*, pages 115 – 130. Springer-Verlag, 2006.
12. L. Freitas, A. L. C. Cavalcanti, and J. C. P. Woodcock. Taking our own medicine: Applying the refinement calculus to state-rich refinement model checking. In Z. Liu and J. He, editors, *Formal Methods and Software Engineering, 8th International Conference on Formal Engineering Methods, ICFEM 2006*, volume 4260 of *Lecture Notes in Computer Science*, pages 697–716, Guiyang, China, November 2006. Springer.
13. C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall International, 1985.
14. C. A. R. Hoare and H. Jifeng. *Unifying Theories of Programming*. Prentice-Hall International, 1998.
15. Y. Isobe and M. Roggenbach. A generic theorem prover of CSP refinement. *TACAS 2005*, LNCS 3440, 2005.
16. M. Oliveira, A. Cavalcanti, and J. Woodcock. Unifying theories in proofpowerz. In S. Dunne and B. Stoddart, editors, *UTP*, volume 4010 of *Lecture Notes in Computer Science*, pages 123–140. Springer, 2006.
17. M. V. M. Oliveira and A. L. C. Cavalcanti. ArcAngelC: a Refinement Tactic Language for *Circus*. *Electronic Notes in Theoretical Computer Science*, **214C**:203 – 229, 2008.
18. M. V. M. Oliveira, A. L. C. Cavalcanti, and J. C. P. Woodcock. A UTP Semantics for *Circus*. *Formal Aspects of Computing*, **21**(1):3 – 32, 2007. The original publication is available at www.springerlink.com.
19. S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In D. Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752. Springer-Verlag, June 1992.
20. L. C. Paulson. Isabelle : A Generic Theorem Prover. 828, 1994.
21. A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice-Hall International, 1998.
22. 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.
23. S. A. Schneider. *Concurrent and real-time systems: the CSP approach*. John Wiley & Sons, 1999.
24. J. M. Spivey. *The Z notation: a reference manual*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, UK, 1992.
25. H. Tej and B. Wolff. A corrected failure divergence model for csp in isabelle/hol. In *FME '97: Proceedings of the 4th International Symposium of Formal Methods Europe on Industrial Applications and Strengthened Foundations of Formal Methods*, pages 318–337, London, UK, 1997. Springer-Verlag.

26. K. Wei and J. Heather. Embedding the stable failures model of csp in pvs. In *Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, November 29 - December 2, 2005, Proceedings*.
27. K. Wei and J. Heather. A theorem-proving approach to verification of fair non-repudiation protocols. In *Formal Aspects in Security and Trust, Fourth International Workshop, FAST 2006, Hamilton, Ontario, Canada, August 26-27, 2006, Revised Selected Papers*.
28. K. Wei, J. Woodcock, and A. Burns. A timed model of Circus with the reactive miracle. In *Technical report. In submitting to ICTAC 09*.
29. J. Woodcock and A. Cavalcanti. Circus: a concurrent refinement language. Technical Report Oxford OX1 3QD UK, Oxford University Computing Laboratory, Wolfson Building, Parks Road, July 2001.
30. 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.
31. J. Woodcock and J. Davies. *Using Z: specification, refinement, and proof*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.